Visualization (Matplotlib)

Introduction

- Matplotlib is the oldest library for data visualization in Python.
- It was created to replicate MatLab's plotting capabilities.
- It is an excellent 2D and 3D graphics library for generating scientific figures.

Some of the major Pros of Matplotlib are:

- Generally easy to get started for simple plots
- Support for custom labels and texts
- Great control of every element in a figure
- High-quality output in many formats

Before continuing explore the official Matplotlib web page: http://matplotlib.org/ (http://matplotlib.org/)

Installation

You'll need to install matplotlib first with either:

```
conda install matplotlib
```

or

```
pip install matplotlib
```

Importing

Import the `matplotlib.pyplot` module as `plt`

In [1]:
```
import matplotlib.pyplot as plt
```

Let's tell jupyter we want to see the plots as outputs of the cells

In [2]:
```
%matplotlib inline
```

If you are using another editor, use: `plt.show()` to display your figure in another window.

Example

In [3]:
```
import numpy as np
x = np.linspace(0, 5, 11)
y = x ** 2
```

In [4]:
```
x
```

Out[4]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

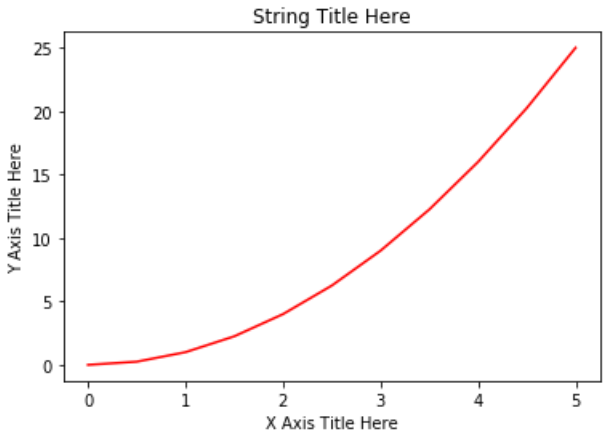
In [5]:
```
y
```

Out[5]:
```
array([ 0.  ,  0.25,  1.  ,  2.25,  4.  ,  6.25,  9.  , 12.25, 16.  ,
       20.25, 25.  ])
```

Basic Matplotlib Commands

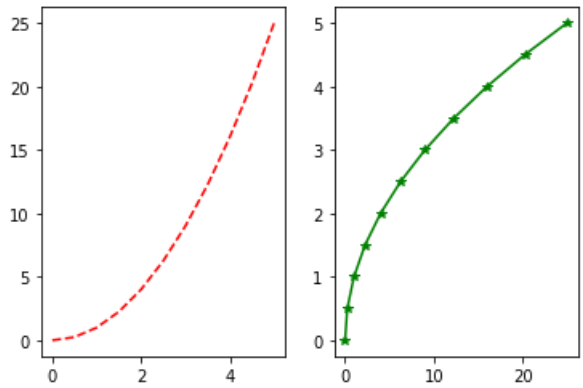
Use `plot` to generate a line plot.

In [6]:
```
plt.plot(x, y, 'r')  # 'r' is the color red
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show()
```



Creating Multiplots on Same Canvas

In [7]:
```
# plt.subplot(nrows, ncols, plot_number)
plt.subplot(1, 2, 1)
plt.plot(x, y, 'r--')  # More on color options later
plt.subplot(1, 2, 2)
plt.plot(y, x, 'g*-');
```

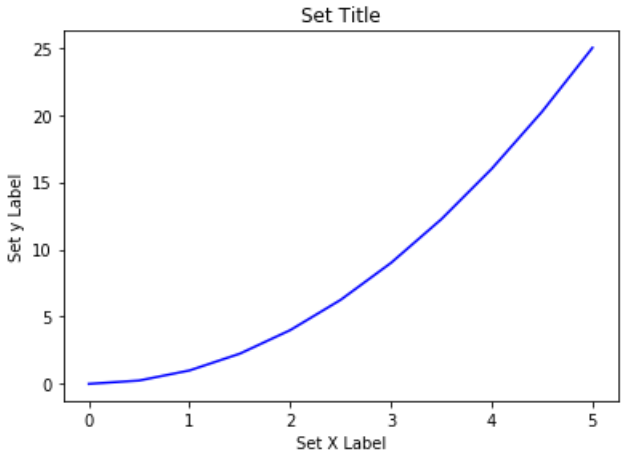


Matplotlib Object Oriented Method

The main idea is to create figure objects and then call methods or attributes off of that object.

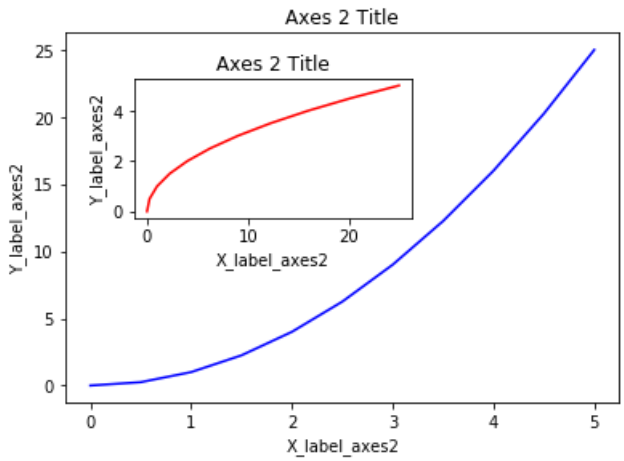
In [8]:
```
# Create Figure (empty canvas)
fig = plt.figure()

# Add set of axes to figure
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])  # left, bottom, width, height (range 0 to 1)

# Plot on that set of axes
axes.plot(x, y, 'b')
axes.set_xlabel('Set X Label') # Notice the use of set_ to begin methods
axes.set_ylabel('Set y Label')
axes.set_title('Set Title')
```

Out[8]:
```
Text(0.5, 1.0, 'Set Title')
```



We now have full control of where the plot axes are placed:

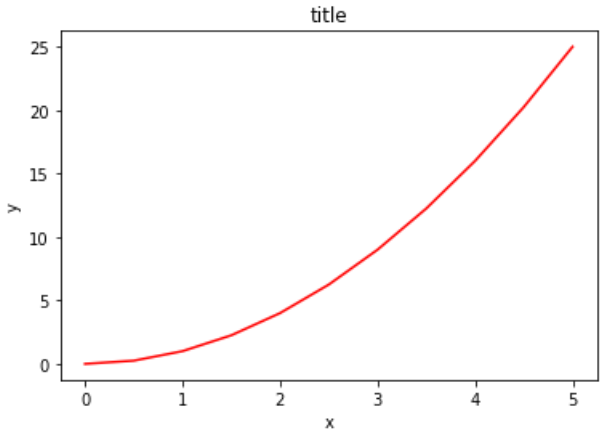
In [9]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# Larger Figure Axes 1
axes1.plot(x, y, 'b')
axes1.set_xlabel('X_label_axes2')
axes1.set_ylabel('Y_label_axes2')
axes1.set_title('Axes 2 Title')

# Insert Figure Axes 2
axes2.plot(y, x, 'r')
axes2.set_xlabel('X_label_axes2')
axes2.set_ylabel('Y_label_axes2')
axes2.set_title('Axes 2 Title');
```



subplots()

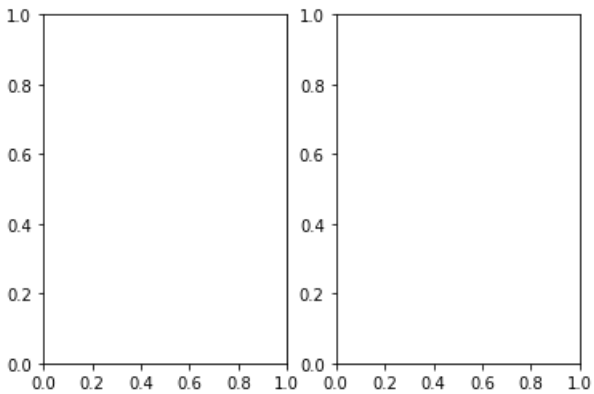
The `plt.subplots()` object acts as an automatic axis manager.

In [10]:
```
# Use similar to plt.figure() except use tuple unpacking to grab fig and axes
fig, axes = plt.subplots()

# Now use the axes object to add stuff to plot
axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



We can specify the number of rows and columns in the `subplots()` object:

In [11]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



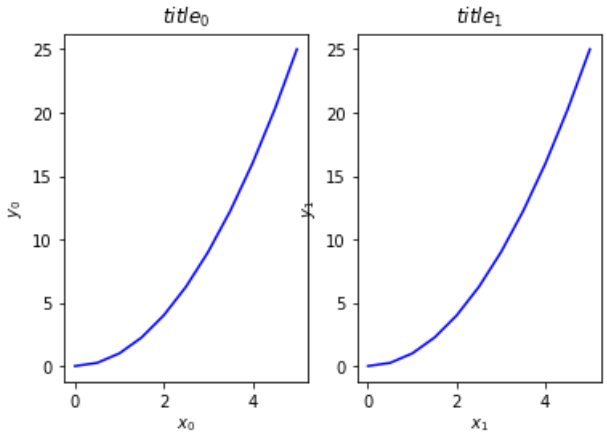
In [12]:
```
# Axes is an array of axes to plot on
axes
```

Out[12]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11b879250>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11b8903d0>],
      dtype=object)
```

We can iterate through this array:

In [13]:
```
for idx, ax in enumerate(axes):
    ax.plot(x, y, 'b')
    ax.set_xlabel('$x_{}$'.format(idx))
    ax.set_ylabel('$y_{}$'.format(idx))
    ax.set_title('$title_{}$'.format(idx))

# Display the figure object
fig
```

Out[13]:



To avoid overlapping subplots, we can use `fig.tight_layout()` or `plt.tight_layout()` which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

In [14]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

axes[0].plot(x, y, 'g')
axes[0].set_ylabel('y')

axes[1].plot(y, x, 'r')
axes[1].set_ylabel('y')
plt.tight_layout()
```

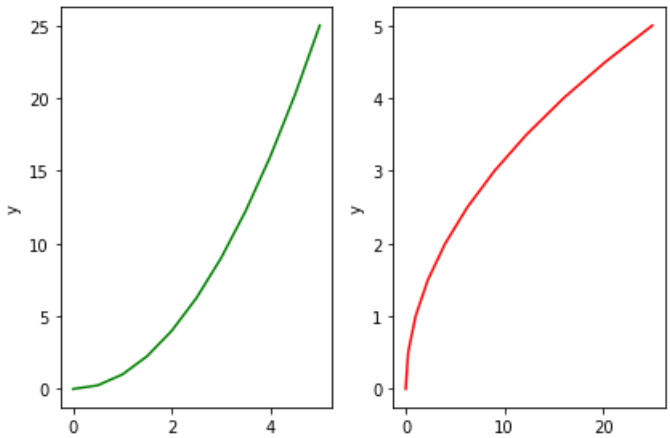


Figure size, aspect ratio and DPI

Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created.

- `figsize` is a tuple of the width and height of the figure in inches
- `dpi` is the dots-per-inch (pixel per inch).

For example:

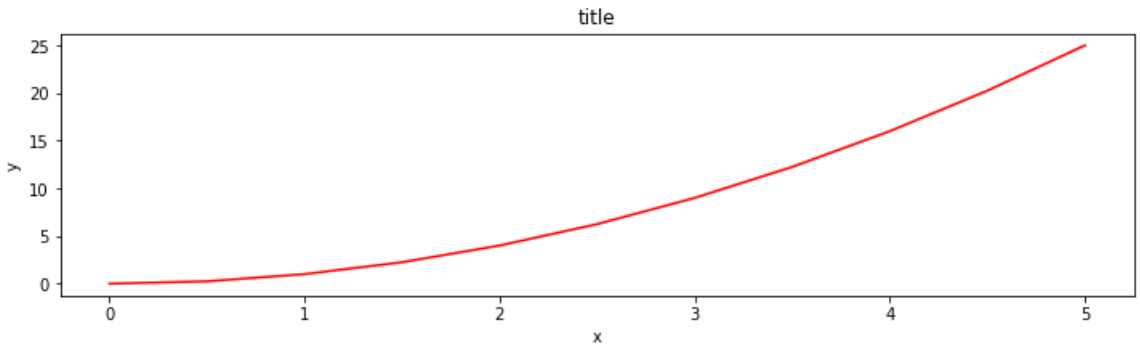
In [15]:
```
fig = plt.figure(figsize=(8,4), dpi=100)
```

```
<Figure size 800x400 with 0 Axes>
```

The same arguments can also be passed to layout managers, such as the `subplots` function:

In [16]:
```
fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title')
```

Out[16]:
```
Text(0.5, 1.0, 'title')
```



Saving figures

Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF.

To save a figure to a file we can use the `savefig` method in the `Figure` class:

In [17]:
```
fig.savefig("filename.png")
```

We can also optionally specify the DPI and choose between different output formats:

In [18]:
```
fig.savefig("filename.png", dpi=200)
```

Legends, labels and titles

Figure titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

In [19]:
```
ax.set_title("title");
```

Axis labels

Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

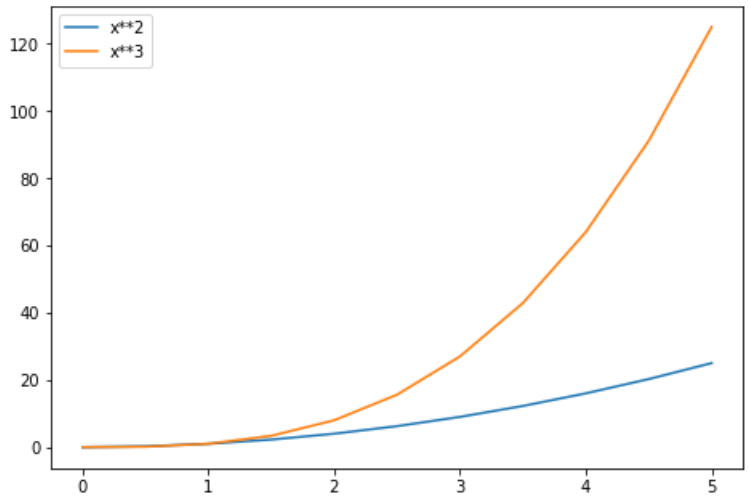
In [20]:
```
ax.set_xlabel("x")
ax.set_ylabel("y");
```

Legends

You can use the `label="label text"` keyword argument when plots or other objects are added to the figure, and then using the **legend** method without arguments to add the legend to the figure:

In [21]:
```
fig = plt.figure()

ax = fig.add_axes([0,0,1,1])

ax.plot(x, x**2, label="x**2")
ax.plot(x, x**3, label="x**3")
ax.legend()
```

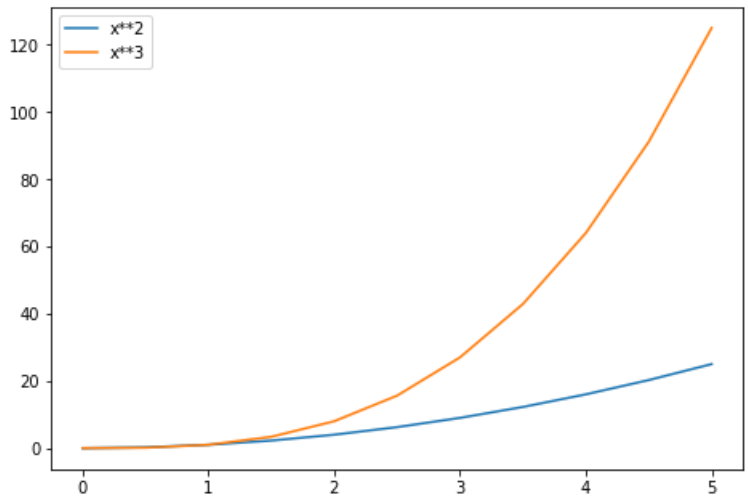
Out[21]:
```
<matplotlib.legend.Legend at 0x11b410fd0>
```



The **legend** function takes an optional keyword argument `loc` that can be used to specify where in the figure the legend is to be drawn. The allowed values of **loc** are numerical codes for the various places the legend can be drawn. See the documentation page (http://matplotlib.org/users/legend_guide.html#legend-location) for details. Some of the most common **loc** values are:

In [22]:
```
# Lots of options....

ax.legend(loc="upper right") # upper right corner
ax.legend(loc="upper left") # upper left corner
ax.legend(loc="lower left") # lower left corner
ax.legend(loc="lower right") # lower right corner

# Most common to choose
ax.legend(loc="best") # let matplotlib decide the optimal location
fig
```

Out[22]:



Setting colors, linewidths, linetypes

Matplotlib gives you *a lot* of options for customizing colors, linewidths, and linetypes.

There is the basic MATLAB-like syntax and a more sane version.

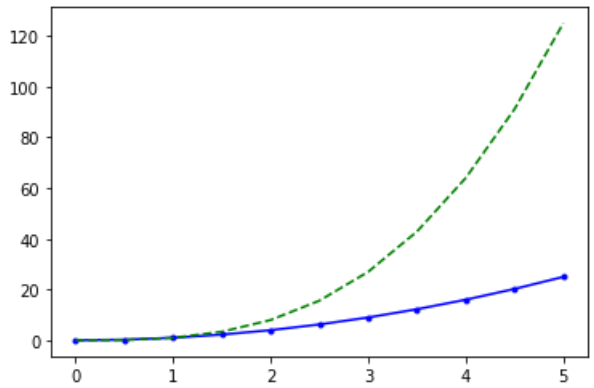
Colors with MatLab like syntax

First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc.

The MATLAB API for selecting line styles are also supported: where, for example, `'b.-'` means a blue line with dots:

In [23]:
```
# MATLAB style line color and style
fig, ax = plt.subplots()
ax.plot(x, x**2, 'b.-')  # blue line with dots
ax.plot(x, x**3, 'g--')  # green dashed line
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x11ac791d0>]
```

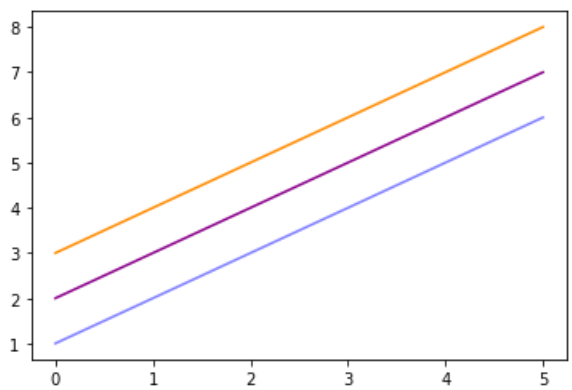


Colors with the `color=` parameter

We can also define colors by their names or RGB hex codes in the `color` parameter. `alpha` indicates opacity.

In [24]:
```
fig, ax = plt.subplots()

ax.plot(x, x+1, color="blue", alpha=0.5) # half-transparant
ax.plot(x, x+2, color="#8B008B")        # RGB hex code
ax.plot(x, x+3, color="#FF8C00")        # RGB hex code
```

Out[24]:
```
[<matplotlib.lines.Line2D at 0x11b412650>]
```

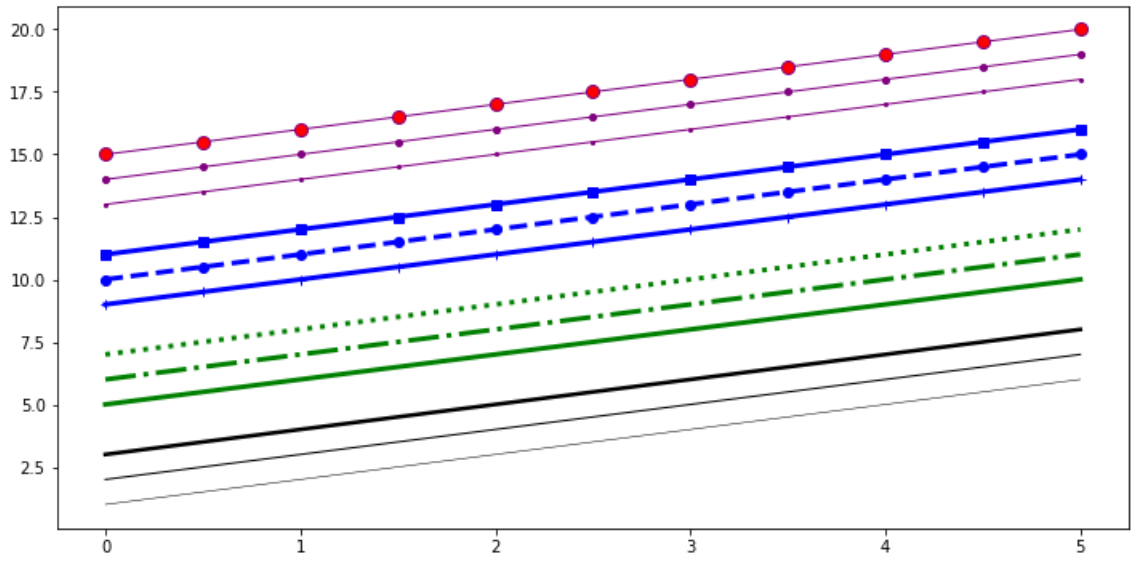


Line and marker styles

To change the line width, we can use the `linewidth` or `lw` keyword argument. The line style can be selected using the `linestyle` or `ls` keyword arguments:

In [25]:
```
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="black", linewidth=0.5)
ax.plot(x, x+2, color="black", linewidth=1.00)
ax.plot(x, x+3, color="black", linewidth=2.50)

# possible linestype options '-', '--', '-.', ':', 'steps'
ax.plot(x, x+5, color="green", lw=3, linestyle='-')
ax.plot(x, x+6, color="green", lw=3, ls='-.')
ax.plot(x, x+7, color="green", lw=3, ls=':')

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3', '4', ...
ax.plot(x, x+9, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, markerfacecolor="r
ed")
```

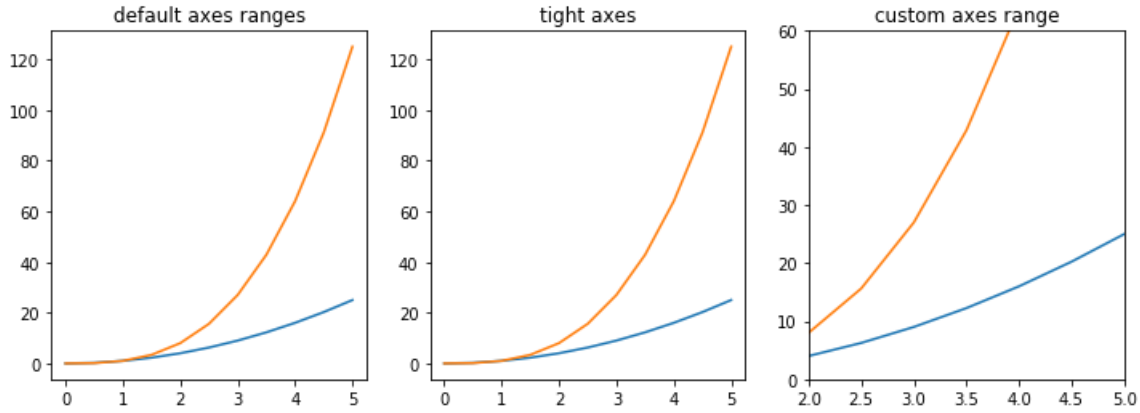
Out[25]:
```
[<matplotlib.lines.Line2D at 0x11b4f6cd0>]
```



Plot range

We can configure the ranges of the axes using the `set_ylim` and `set_xlim` methods in the axis object, or `axis('tight')` for automatically getting "tightly fitted" axes ranges:

In [26]:
```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```

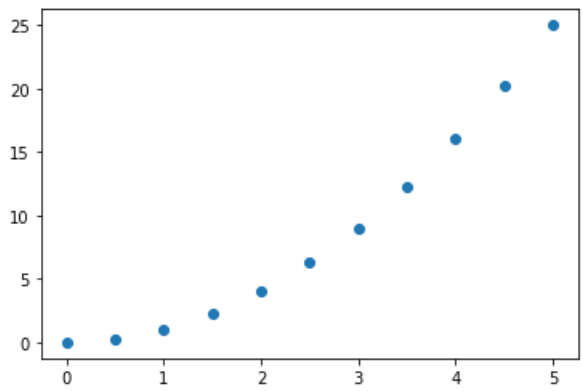


Special Plot Types

There are many specialized plots that we can create in matplotlib. For most of these we will use another similar library, but here a few examples:

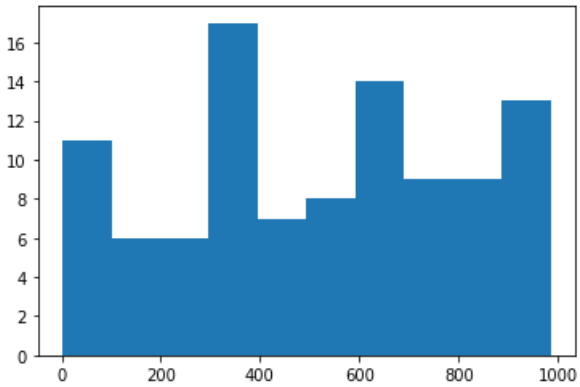
In [27]:
```
plt.scatter(x, y)
```

Out[27]:
```
<matplotlib.collections.PathCollection at 0x11b4f6e50>
```



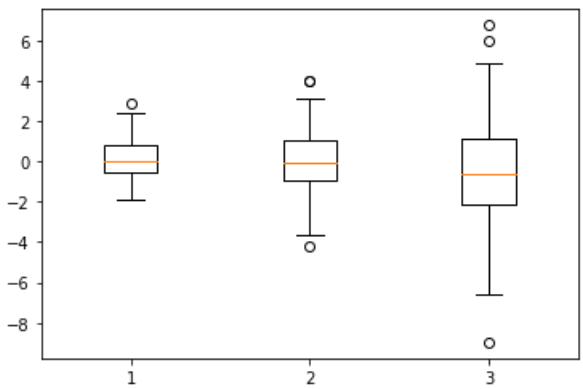
In [28]:
```
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

Out[28]:
```
(array([11.,  6.,  6., 17.,  7.,  8., 14.,  9.,  9., 13.]),
 array([  2. , 100.3, 198.6, 296.9, 395.2, 493.5, 591.8, 690.1, 788.4,
        886.7, 985. ]),
 <a list of 10 Patch objects>)
```



In [29]:
```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data)
```

Out[29]:
```
{'whiskers': [<matplotlib.lines.Line2D at 0x11cbb3450>,
  <matplotlib.lines.Line2D at 0x11cbb3a10>,
  <matplotlib.lines.Line2D at 0x11cbc38d0>,
  <matplotlib.lines.Line2D at 0x11cbbcf10>,
  <matplotlib.lines.Line2D at 0x11cbe5c10>,
  <matplotlib.lines.Line2D at 0x11cbe5c90>],
 'caps': [<matplotlib.lines.Line2D at 0x11cbb3f10>,
  <matplotlib.lines.Line2D at 0x11cba8810>,
  <matplotlib.lines.Line2D at 0x11cbbc950>,
  <matplotlib.lines.Line2D at 0x11cbca7d0>,
  <matplotlib.lines.Line2D at 0x11cbec690>,
  <matplotlib.lines.Line2D at 0x11cbecb90>],
 'boxes': [<matplotlib.lines.Line2D at 0x11cba87d0>,
  <matplotlib.lines.Line2D at 0x11cbb3f90>,
  <matplotlib.lines.Line2D at 0x11cbe5710>],
 'medians': [<matplotlib.lines.Line2D at 0x11cbbc990>,
  <matplotlib.lines.Line2D at 0x11cbcacd0>,
  <matplotlib.lines.Line2D at 0x11cbecc50>],
 'fliers': [<matplotlib.lines.Line2D at 0x11cbbce90>,
  <matplotlib.lines.Line2D at 0x11cbcad50>,
  <matplotlib.lines.Line2D at 0x11cbf7610>],
 'means': []}
```



Further reading

- http://www.matplotlib.org (http://www.matplotlib.org) - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib (https://github.com/matplotlib/matplotlib) - The source code for matplotlib.
- http://matplotlib.org/gallery.html (http://matplotlib.org/gallery.html) - A large gallery showcaseing various types of plots matplotlib can create. Highly recommended!
- http://www.loria.fr/~rougier/teaching/matplotlib (http://www.loria.fr/~rougier/teaching/matplotlib) - A good matplotlib tutorial.
- http://scipy-lectures.github.io/matplotlib/matplotlib.html (http://scipy-lectures.github.io/matplotlib/matplotlib.html) - Another good matplotlib reference.

Matplotlib Exercises - Solutions

NOTE: ALL THE COMMANDS FOR PLOTTING A FIGURE SHOULD ALL GO IN THE SAME CELL. SEPARATING THEM OUT INTO MULTIPLE CELLS MAY CAUSE NOTHING TO SHOW UP.

Exercises

Follow the instructions to recreate the plots using this data.

Data

In [1]:
```
import numpy as np
x = np.arange(0, 100)
y = x * 2
z = x ** 2
```

Import `matplotlib.pyplot` as `plt` and set `%matplotlib inline` if you are using the jupyter notebook.

In [2]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

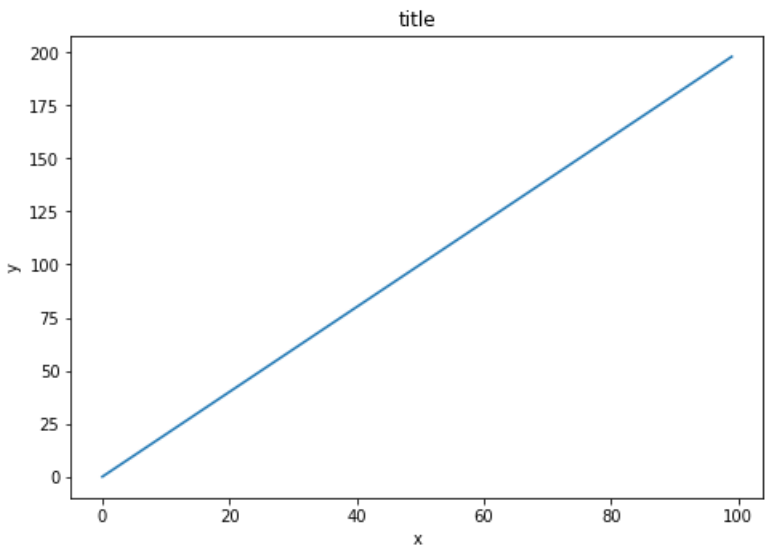
Exercise 1

Follow along with these steps:

- **Create a figure object called fig using `plt.figure()`**
- **Use `add_axes` to add an axis to the figure canvas at `[0,0,1,1]`. Call this new axis `ax`.**
- **Plot (x,y) on that axes and set the labels and titles to match the plot below:**

In [3]:
```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.plot(x,y)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('title')
```

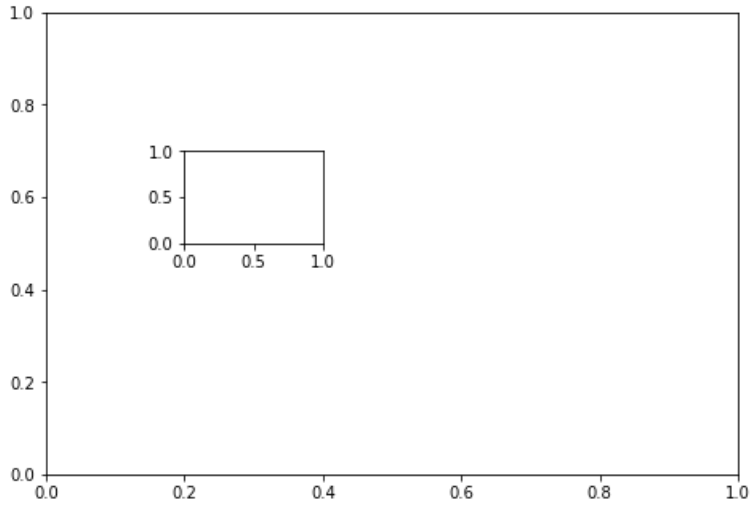
Out[3]:
```
Text(0.5, 1.0, 'title')
```



Exercise 2

Create a figure object and put two axes on it, `ax1` and `ax2` . Located at `[0,0,1,1]` and `[0.2,0.5,.2,.2]` respectively.

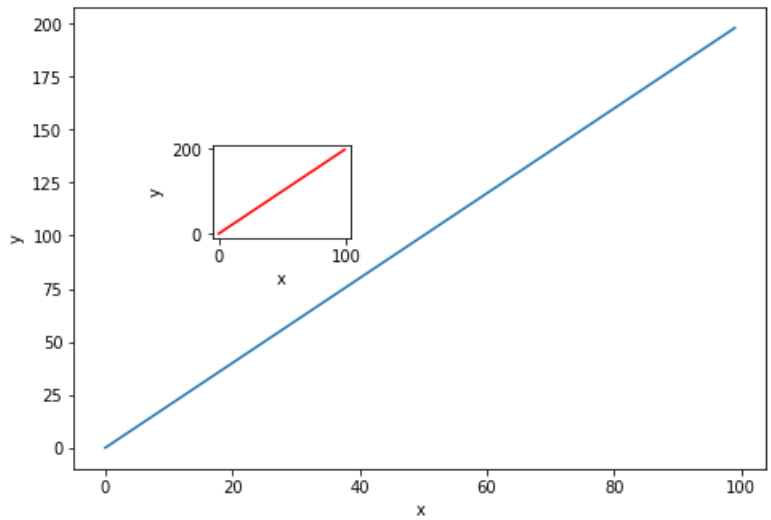
In [4]:
```
fig = plt.figure()

ax1 = fig.add_axes([0,0,1,1])
ax2 = fig.add_axes([0.2,0.5,.2,.2])
```



Now plot (x,y) on both axes. And call your figure object to show it.

In [5]:
```
ax1.plot(x, y)
ax1.set_xlabel('x')
ax1.set_ylabel('y')

ax2.plot(x, y, color='red')
ax2.set_xlabel('x')
ax2.set_ylabel('y')

fig # Show figure object
```

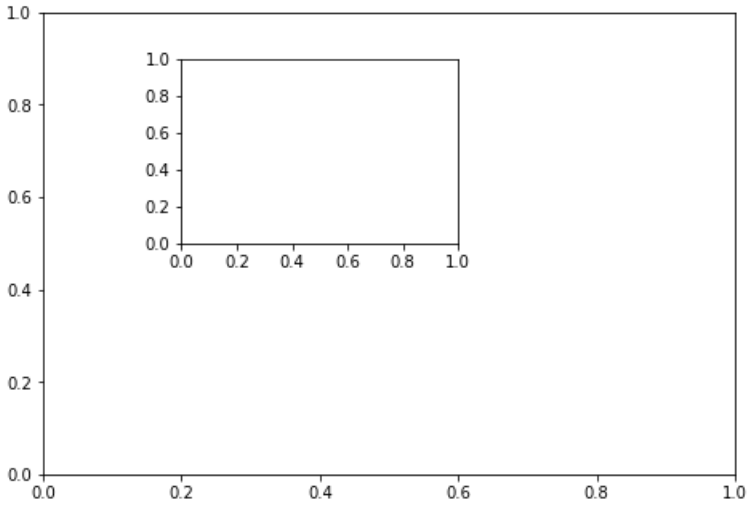
Out[5]:



Exercise 3

Create the plot below by adding two axes to a figure object at `[0,0,1,1]` and `[0.2,0.5,.4,.4]`

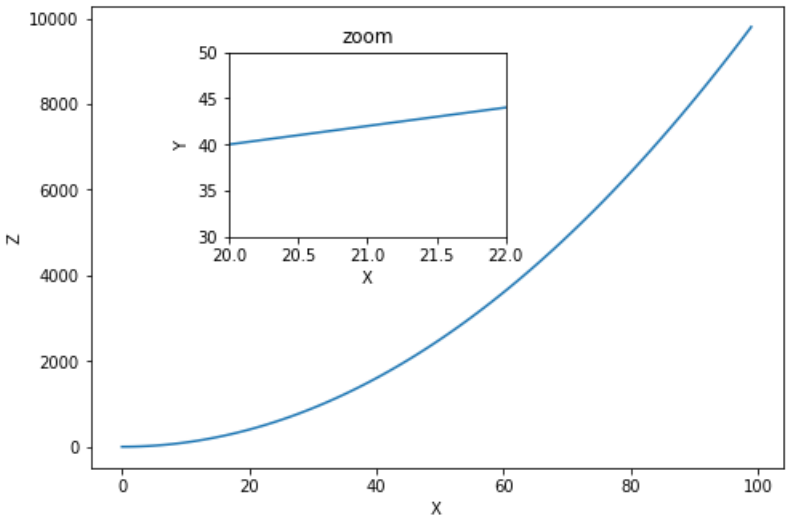
In [6]:
```
fig = plt.figure()

ax = fig.add_axes([0,0,1,1])
ax2 = fig.add_axes([0.2,0.5,.4,.4])
```



Now use x, y, and z arrays to recreate the plot below. Notice the x limits and y limits on the inserted plot:

In [7]:
```
ax.plot(x,z)
ax.set_xlabel('X')
ax.set_ylabel('Z')

ax2.plot(x,y)
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_title('zoom')
ax2.set_xlim(20, 22)
ax2.set_ylim(30, 50)

fig
```

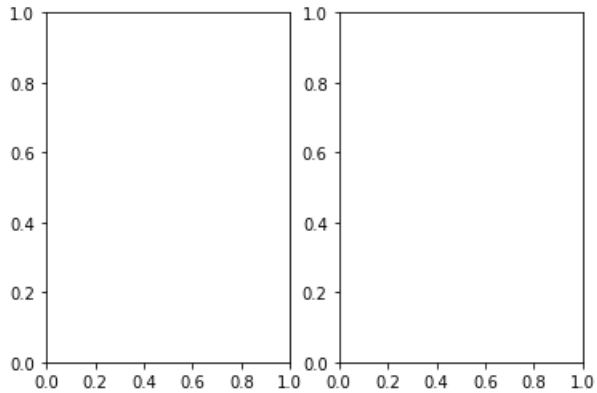
Out[7]:



Exercise 4

Use `plt.subplots(nrows=1, ncols=2)` to create the plot below.

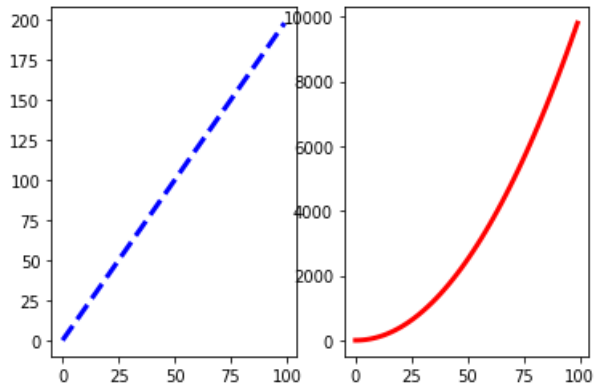
In [8]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



Now plot (x,y) and (x,z) on the axes. Play around with the linewidth and style

In [9]:
```
axes[0].plot(x, y, color="blue", lw=3, ls='--')
axes[1].plot(x, z, color="red", lw=3, ls='-')
fig
```

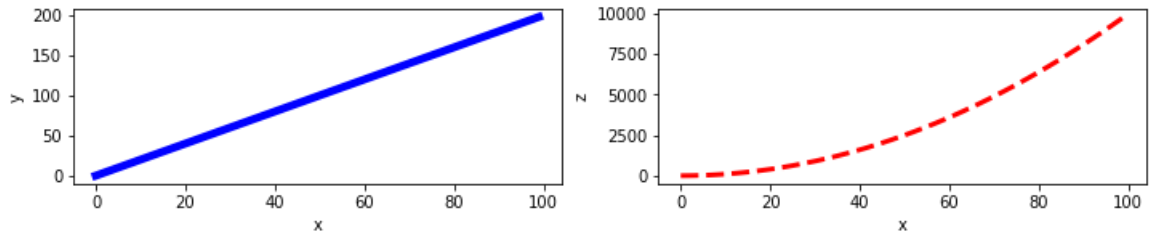
Out[9]:



See if you can resize the plot by adding the `figsize()` argument in `plt.subplots()` are copying and pasting your previous code.

In [10]:
```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12,2))

axes[0].plot(x, y, color="blue", lw=5)
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')

axes[1].plot(x, z, color="red", lw=3, ls='--')
axes[1].set_xlabel('x')
axes[1].set_ylabel('z')
```

Out[10]:
```
Text(0, 0.5, 'z')
```



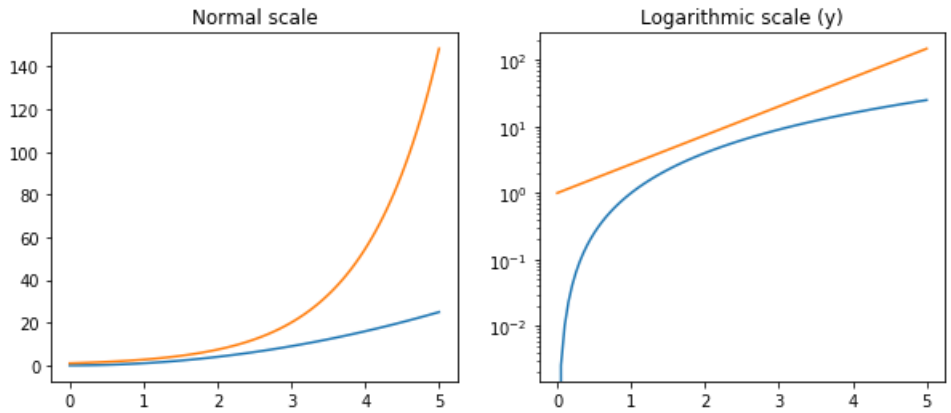
Advanced Matplotlib Concepts Lecture

In [2]:
```
import matplotlib.pyplot as plt
import matplotlib
import numpy as np

%matplotlib inline

x = np.linspace(0, 5, 100)
y = x ** 2
```

Logarithmic scale

It is possible to set logarithmic scale for one or both axes using `set_xscale` and `set_yscale` methods which accept one parameter (with the value `"log"` in this case):

In [3]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

axes[0].plot(x, x**2, x, np.exp(x))
axes[0].set_title("Normal scale")

axes[1].plot(x, x**2, x, np.exp(x))
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)");
```

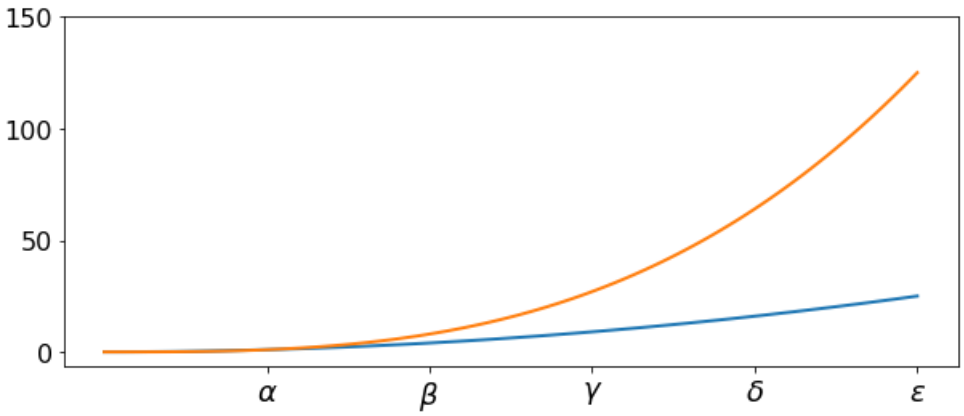


Placement of ticks and custom tick labels

We can specify where to put the axis ticks with `set_xticks` and `set_yticks`, and the labels with `set_xticklabels` and `set_yticklabels`:

In [4]:
```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, x**2, x, x**3, lw=2)

ax.set_xticks([1, 2, 3, 4, 5])
ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$', r'$\epsilon$'], font
size=18)

yticks = [0, 50, 100, 150]
ax.set_yticks(yticks)
ax.set_yticklabels(yticks, fontsize=16)
```

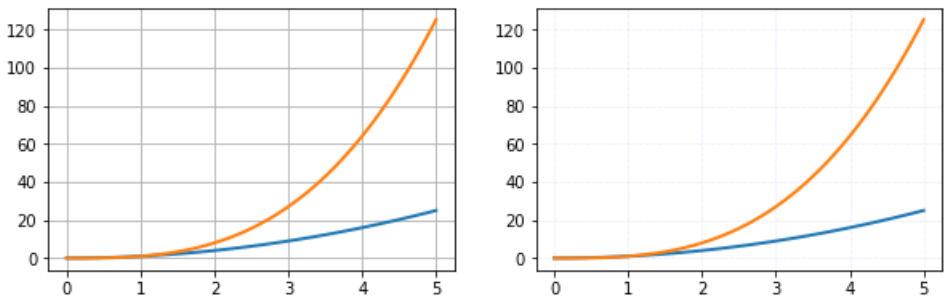
Out[4]:
```
[Text(0, 0, '0'), Text(0, 0, '50'), Text(0, 0, '100'), Text(0, 0, '150')]
```



Axis grid

With the `grid` method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the `plot` function:

In [5]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.2, linestyle='dotted', linewidth=0.5)
```

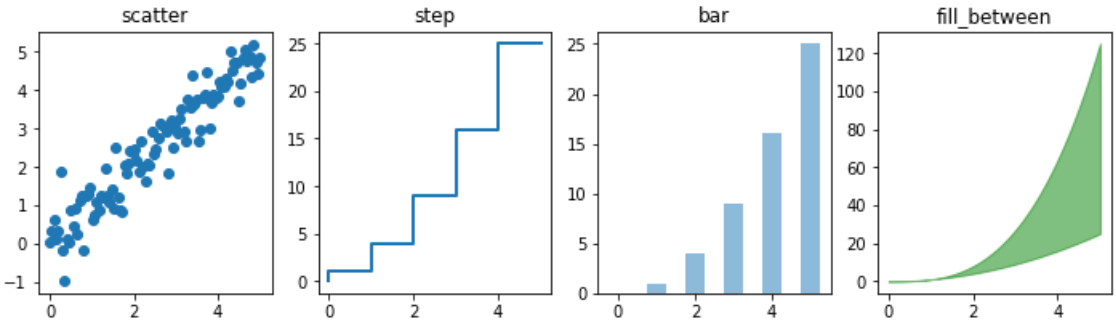


Other 2D plot styles

In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html (http://matplotlib.org/gallery.html). Some of the more useful ones are show below:

In [6]:
```
n = np.array([0,1,2,3,4,5])
```

In [11]:
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(x, x + 0.5*np.random.randn(len(x)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")

axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between");
```



Colormap and contour figures

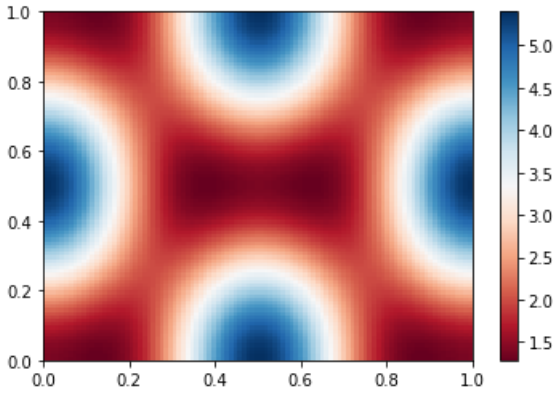
Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see: http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps (http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps)

In [15]:
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha * np.cos(phi_ext - 2*phi_p
)
```

In [16]:
```
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

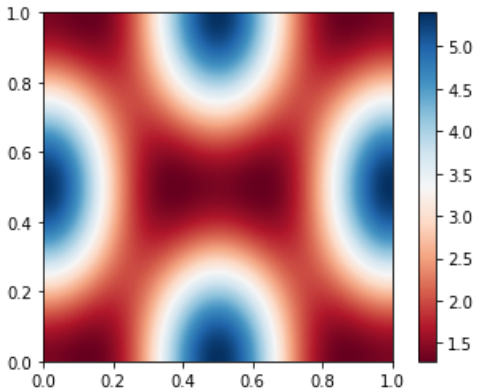
pcolor

In [17]:
```
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax
=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```



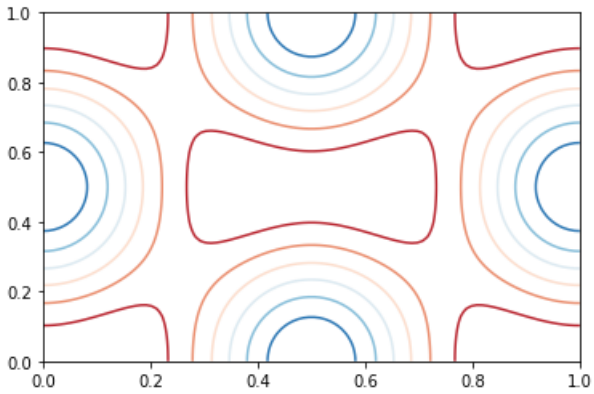
imshow

In [18]:
```
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), extent=[0,
1, 0, 1])
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)
```



contour

In [19]:
```
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), extent=[
0, 1, 0, 1])
```



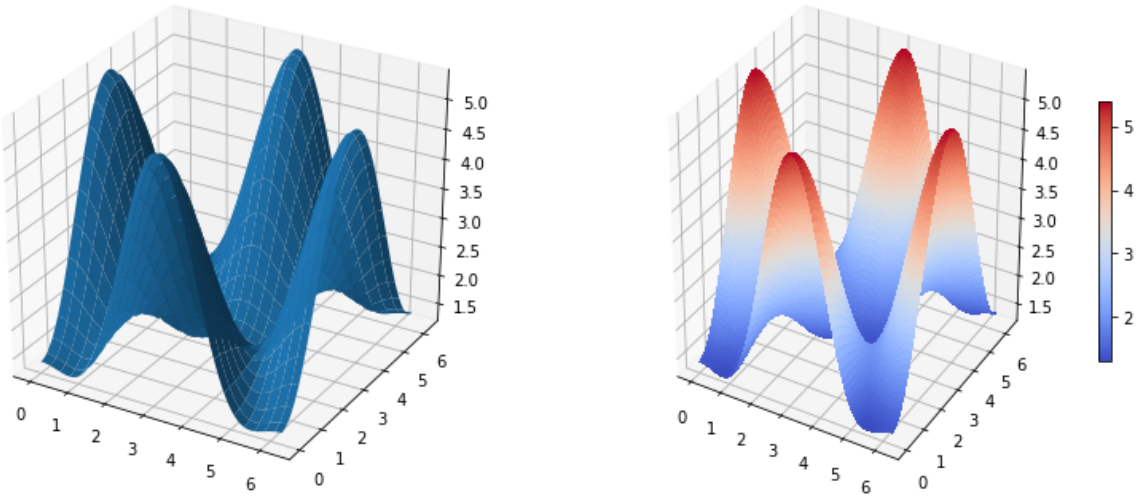
3D figures

To use 3D graphics in matplotlib, we first need to create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

In [20]:
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

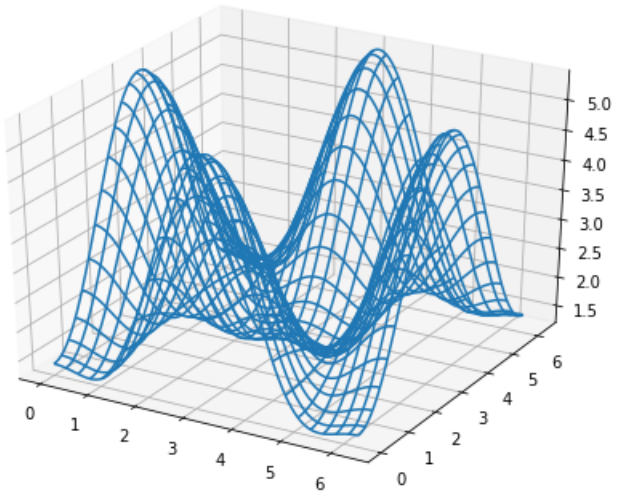
Surface plots

In [21]:
```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```



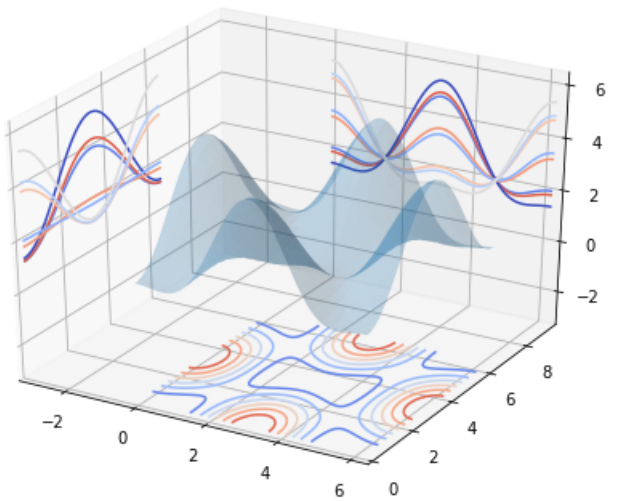
Wire-frame plot

In [22]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```



Coutour plots with projections

In [23]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwarm)

ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);
```



Distribution Plots

Let's discuss some plots that allow us to visualize the distribution of a data set. These plots are:

- distplot
- jointplot
- pairplot
- rugplot
- kdeplot

Imports

In [1]:
```
import seaborn as sns
%matplotlib inline
```

Data

Seaborn comes with built-in data sets!

In [2]:
```
tips = sns.load_dataset('tips')
```

In [3]:
```
tips.head()
```

Out[3]:

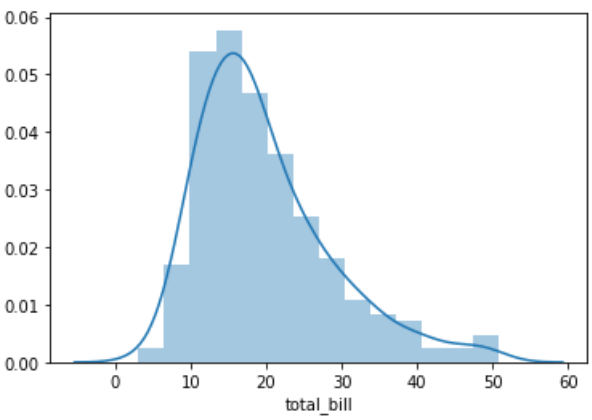
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

distplot

The distplot shows the distribution of a univariate set of observations.

In [4]:
```
sns.distplot(tips['total_bill'])
```

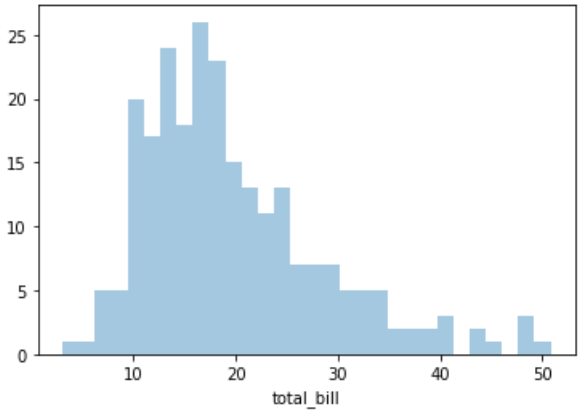
Out[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a23c62610>
```



To remove the `kde` (gaussian kernel density estimate) layer and just have the histogram use:

```
In [5]: sns.distplot(tips['total_bill'], kde=False, bins=30)
```

```
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x1a24436f50>
```



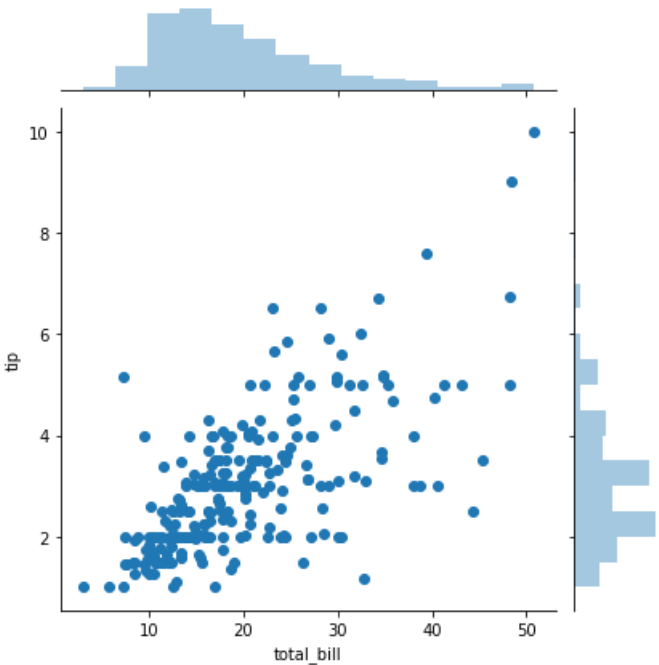
jointplot

`jointplot()` allows you to basically match up two distplots for bivariate data. With your choice of what **kind** parameter to compare with:

- “scatter”
- “reg”
- “resid”
- “kde”
- “hex”

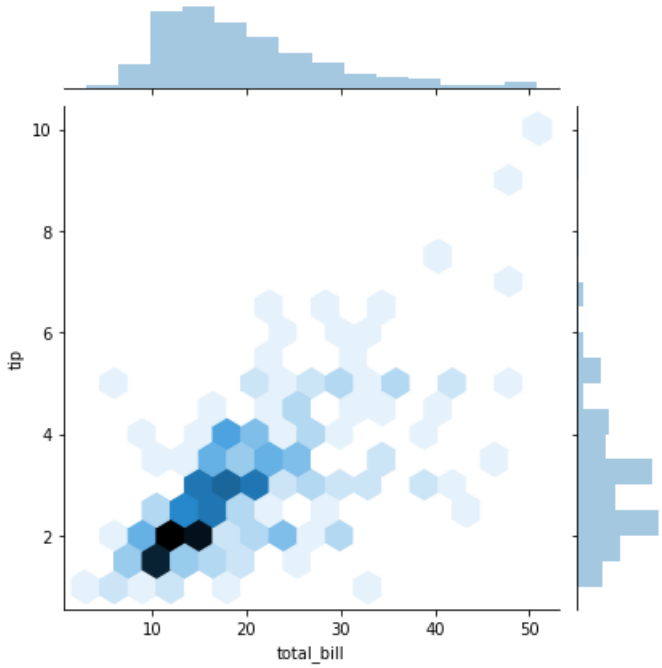
```
In [6]: sns.jointplot(x='total_bill', y='tip', data=tips, kind='scatter')
```

```
Out[6]: <seaborn.axisgrid.JointGrid at 0x1a2459c990>
```



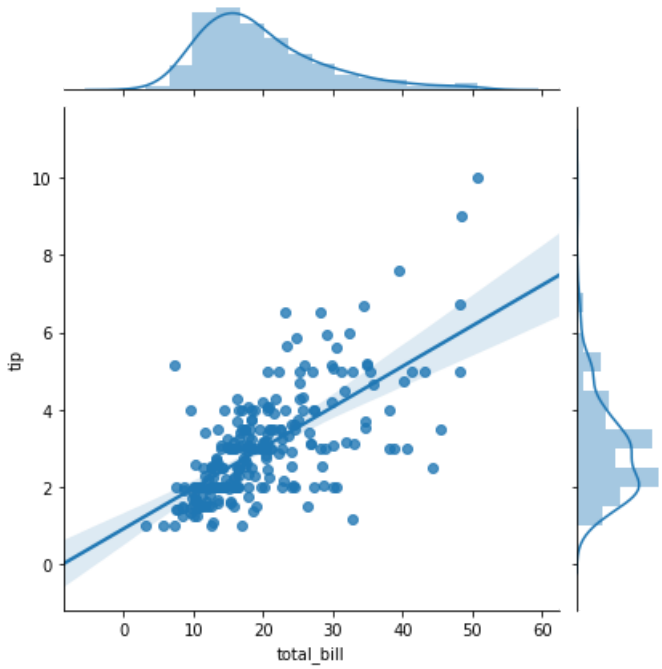
In [7]:
```
sns.jointplot(x='total_bill', y='tip', data=tips, kind='hex')
```

Out[7]:
```
<seaborn.axisgrid.JointGrid at 0x1a24799450>
```



In [8]:
```
sns.jointplot(x='total_bill', y='tip', data=tips, kind='reg')
```

Out[8]:
```
<seaborn.axisgrid.JointGrid at 0x1a244a0a10>
```

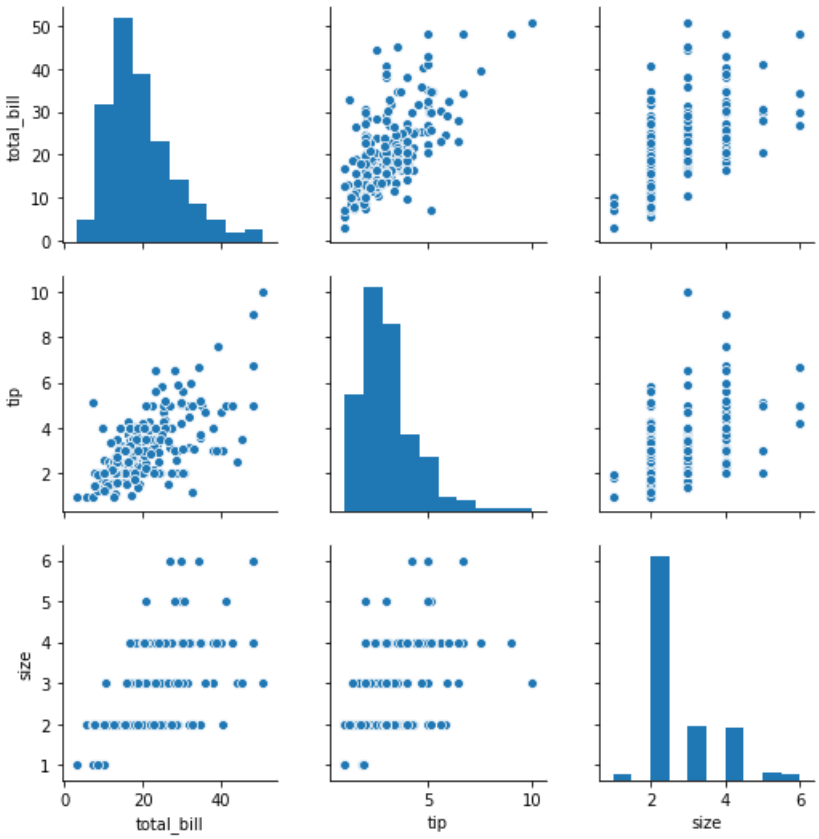


pairplot

`pairplot` will plot pairwise relationships across an entire dataframe (for the numerical columns) and supports a color hue argument (for categorical columns).

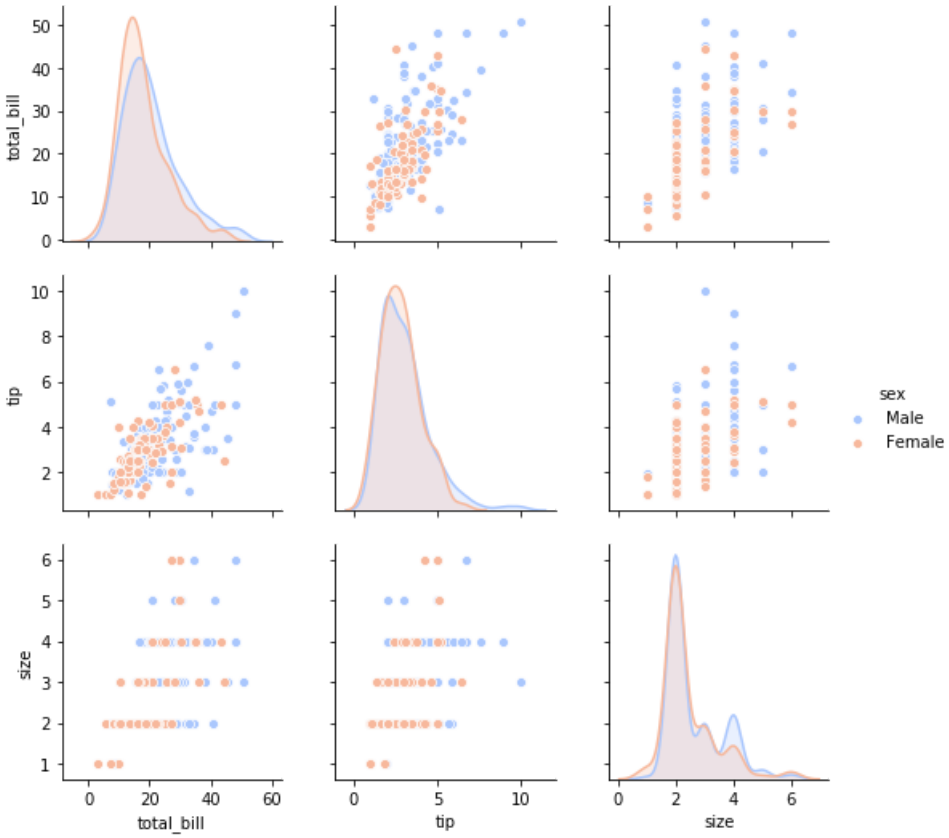
In [9]:
```
sns.pairplot(tips)
```

Out[9]: <seaborn.axisgrid.PairGrid at 0x110d69450>



In [10]:
```
sns.pairplot(tips, hue='sex', palette='coolwarm')
```

Out[10]: <seaborn.axisgrid.PairGrid at 0x1a253eb290>

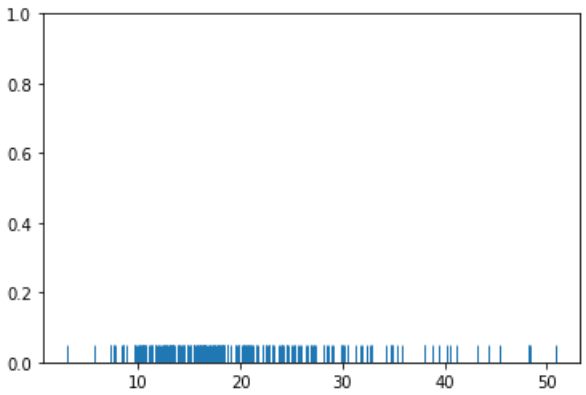


rugplot

`rugplots` are a very simple concept, they just draw a dash mark for every point on a univariate distribution.

In [11]:
```
sns.rugplot(tips['total_bill'])
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a253dbd90>
```

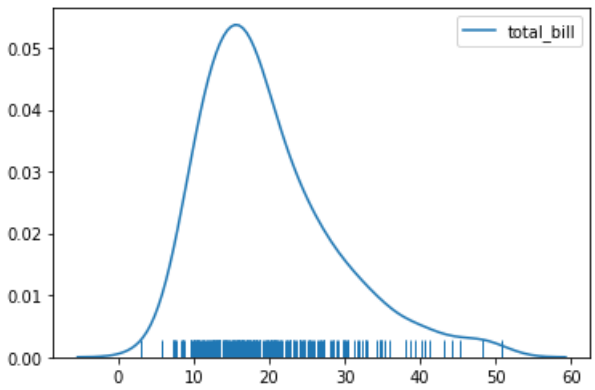


kdeplot

kdeplots are Kernel Density Estimation plots (http://en.wikipedia.org/wiki/Kernel_density_estimation#Practical_estimation_of_the_bandwidth). These KDE plots replace every single observation with a Gaussian (Normal) distribution centered around that value.

In [12]:
```
sns.kdeplot(tips['total_bill'])
sns.rugplot(tips['total_bill'])
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a25dd9d50>
```



Categorical Data Plots

Now let's discuss using seaborn to plot categorical data! There are a few main plot types for this:

- factorplot
- boxplot
- stripplot
- swarmplot
- barplot
- countplot

In [1]:
```
import seaborn as sns
%matplotlib inline
```

In [2]:
```
tips = sns.load_dataset('tips')
tips.head()
```

Out[2]:

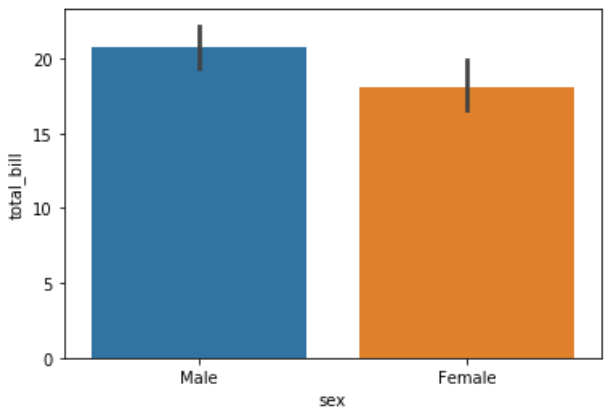
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

barplot and countplot

These very similar plots allow you to get aggregate data off a categorical feature in your data. **barplot** is a general plot that allows you to aggregate the categorical data based off some function, by default the mean:

In [3]:
```
sns.barplot(x='sex', y='total_bill', data=tips)
```

Out[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1ca2bb50>
```

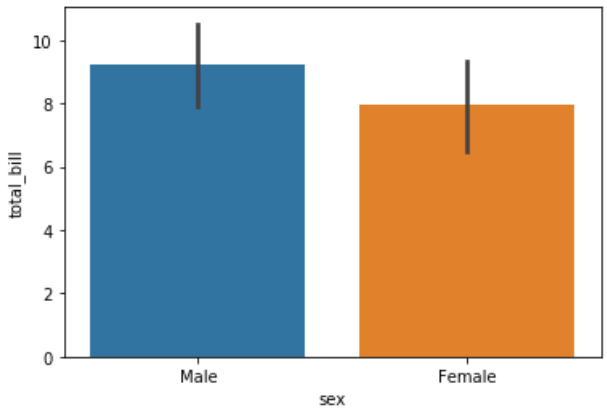


In [4]:
```
import numpy as np
```

You can change the estimator object to your own function, that converts a vector to a scalar:

```
In [5]: sns.barplot(x='sex', y='total_bill', data=tips, estimator=np.std)
```

```
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1d8eb310>
```

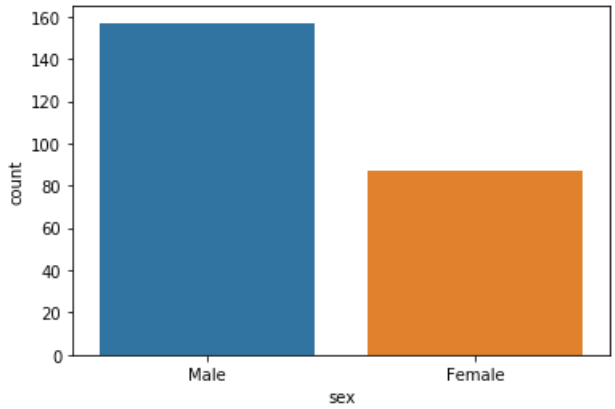


countplot

This is essentially the same as barplot except the estimator is explicitly counting the number of occurrences. Which is why we only pass the x value:

```
In [6]: sns.countplot(x='sex', data=tips)
```

```
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1d9d5050>
```

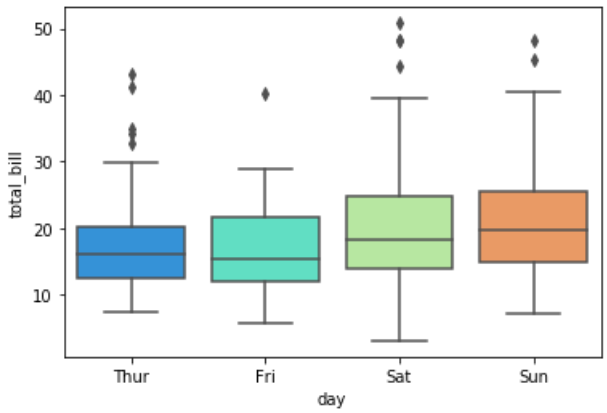


boxplot and violinplot

boxplots and violinplots are used to shown the distribution of categorical data. A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be “outliers” using a method that is a function of the inter-quartile range.

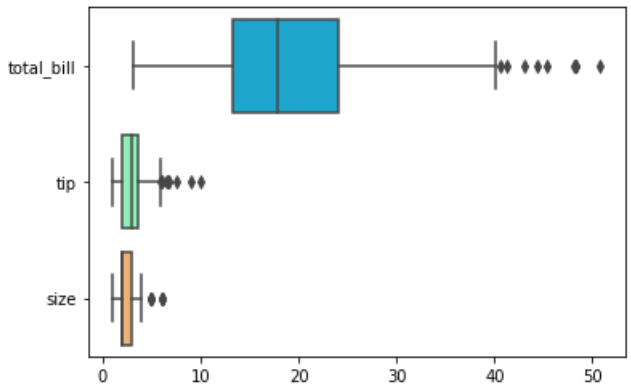
In [7]:
```
sns.boxplot(x="day", y="total_bill", data=tips, palette='rainbow')
```

Out[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1daa6310>
```



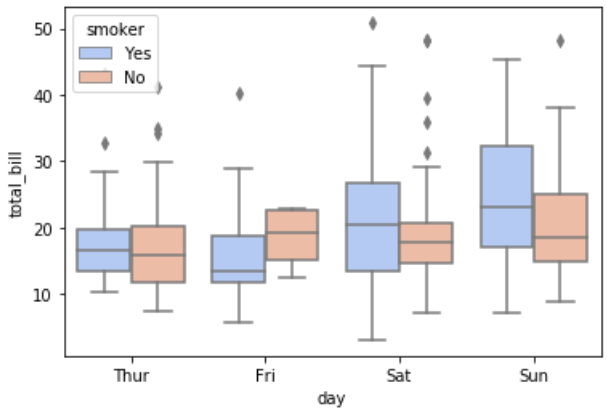
In [8]:
```
# Can do entire dataframe with orient='h'
sns.boxplot(data=tips, palette='rainbow', orient='h')
```

Out[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1dbc20d0>
```



In [9]:
```
sns.boxplot(x="day", y="total_bill", hue="smoker", data=tips, palette="coolwarm")
```

Out[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1dcda950>
```



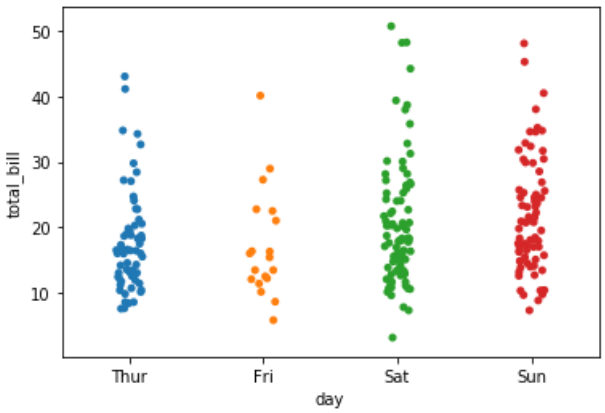
stripplot and swarmplot

The `stripplot` will draw a scatterplot where one variable is categorical. A strip plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.

The `swarmplot` is similar to `stripplot()`, but the points are adjusted (only along the categorical axis) so that they don't overlap. This gives a better representation of the distribution of values, although it does not scale as well to large numbers of observations (both in terms of the ability to show all the points and in terms of the computation needed to arrange them).

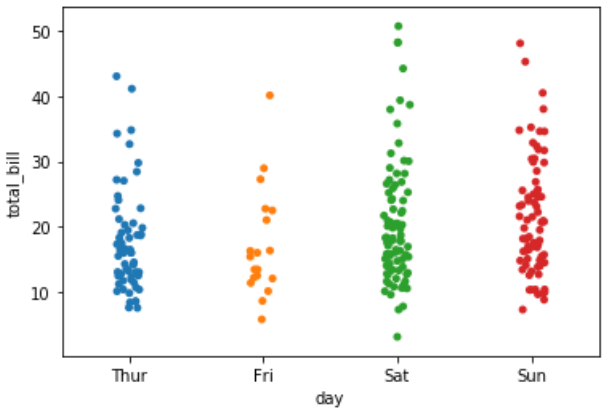
In [10]:
```
sns.stripplot(x="day", y="total_bill", data=tips)
```

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1de7dd10>



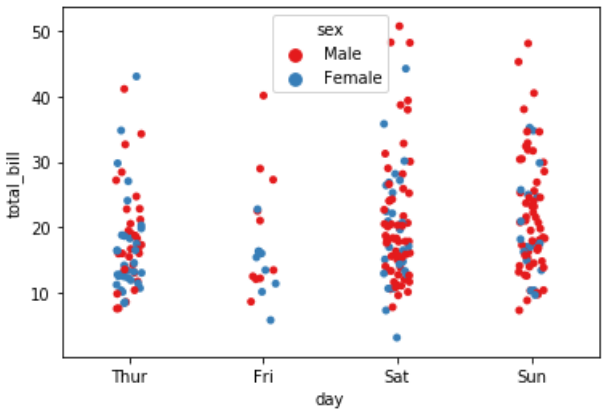
In [11]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=True)
```

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1df50590>



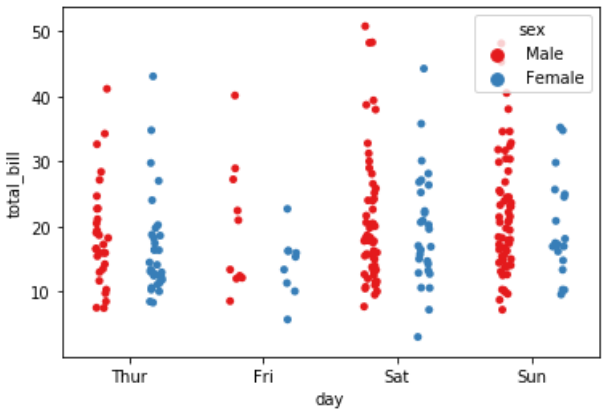
In [12]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=True, hue='sex', palette='Set1')
```

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e02fe50>



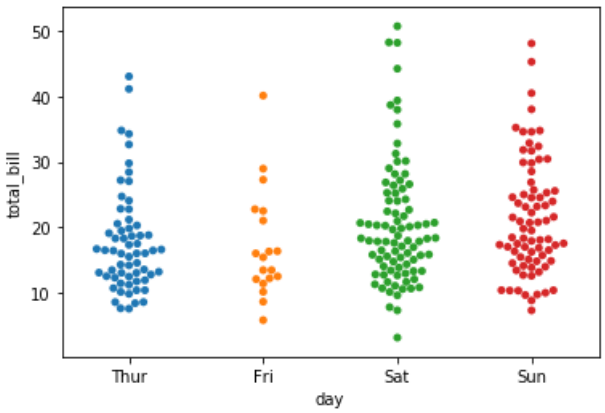
In [13]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=True, hue='sex', palette='Set1', dodge=True)
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e01bb90>



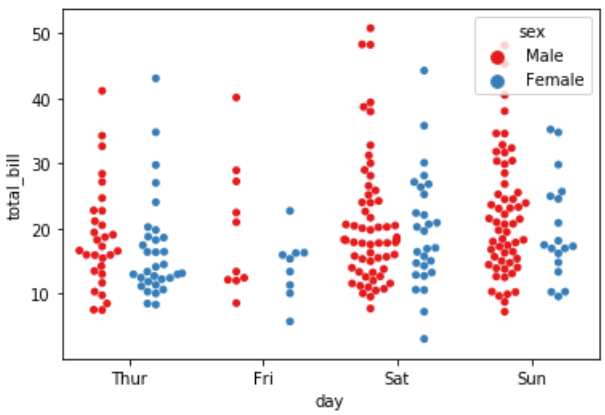
In [14]:
```
sns.swarmplot(x="day", y="total_bill", data=tips)
```

Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e1f3050>



In [15]:
```
sns.swarmplot(x="day", y="total_bill", hue='sex', data=tips, palette="Set1", dodge=True)
```

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e2c6710>



catplot

catplot is the most general form of a categorical plot. It can take in a **kind** parameter to adjust the plot type:

In [16]:
```
sns.catplot(x='sex', y='total_bill', data=tips, kind='bar')
```

Out[16]: <seaborn.axisgrid.FacetGrid at 0x1a1ca2bbd0>