
(http://www.pieriandata.com)

NumPy

NumPy (or Numpy) is a Linear Algebra Library for Python, the reason it is so important for Data Science with Python is that almost all of the libraries in the PyData Ecosystem rely on NumPy as one of their main building blocks.

Numpy is also incredibly fast, as it has bindings to C libraries. For more info on why you would want to use Arrays instead of lists, check out this great StackOverflow post (http://stackoverflow.com/questions/993984/why-numpy-instead-of-python-lists).

We will only learn the basics of NumPy, to get started we need to install it!

Installation Instructions

It is highly recommended you install Python using the Anaconda distribution to make sure all underlying dependencies (such as Linear Algebra libraries) all sync up with the use of a conda install. If you have Anaconda, install NumPy by going to your terminal or command prompt and typing:

```
conda install numpy
```

If you do not have Anaconda and can not install it, please refer to Numpy's official documentation on various installation instructions. (http://docs.scipy.org/doc/numpy-1.10.1/user/install.html)

Using NumPy

Once you've installed NumPy you can import it as a library:

In [1]:
```
import numpy as np
```

Numpy has many built-in functions and capabilities. We won't cover them all but instead we will focus on some of the most important aspects of Numpy: vectors,arrays,matrices, and number generation. Let's start by discussing arrays.

Numpy Arrays

NumPy arrays are the main way we will use Numpy throughout the course. Numpy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-d arrays and matrices are 2-d (but you should note a matrix can still have only one row or one column).

Let's begin our introduction by exploring how to create NumPy arrays.

Creating NumPy Arrays

From a Python List

In [19]:
```
my_list = [1,2,3]
my_list
```

Out[19]:
```
[1, 2, 3]
```

In [16]:
```
np.array(my_list)
```

Out[16]:
```
array([1, 2, 3])
```

In [20]:
```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
my_matrix
```

Out[20]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [21]:
```
np.array(my_matrix)
```

Out[21]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Built-in Methods

There are lots of built-in ways to generate Arrays

arange

Return evenly spaced values within a given interval.

In [22]:
```
np.arange(0,10)
```

Out[22]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [23]:
```
np.arange(0,11,2)
```

Out[23]:
```
array([ 0,  2,  4,  6,  8, 10])
```

zeros and ones

In [24]:
```
np.zeros(3)
```

Out[24]:
```
array([ 0.,  0.,  0.])
```

In [26]:
```
np.zeros((5,5))
```

Out[26]:
```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

In [27]:
```
np.ones(3)
```

Out[27]:
```
array([ 1.,  1.,  1.])
```

In [28]:
```
np.ones((3,3))
```

Out[28]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

linspace

Return evenly spaced numbers over a specified interval.

In [29]:
```
np.linspace(0,10,3)
```

Out[29]:
```
array([  0.,   5.,  10.])
```

In [31]:
```
np.linspace(0,10,50)
```

Out[31]:
```
array([  0.        ,   0.20408163,   0.40816327,   0.6122449 ,
         0.81632653,   1.02040816,   1.2244898 ,   1.42857143,
         1.63265306,   1.83673469,   2.04081633,   2.24489796,
         2.44897959,   2.65306122,   2.85714286,   3.06122449,
         3.26530612,   3.46938776,   3.67346939,   3.87755102,
         4.08163265,   4.28571429,   4.48979592,   4.69387755,
         4.89795918,   5.10204082,   5.30612245,   5.51020408,
         5.71428571,   5.91836735,   6.12244898,   6.32653061,
         6.53061224,   6.73469388,   6.93877551,   7.14285714,
         7.34693878,   7.55102041,   7.75510204,   7.95918367,
         8.16326531,   8.36734694,   8.57142857,   8.7755102 ,
         8.97959184,   9.18367347,   9.3877551 ,   9.59183673,
         9.79591837,  10.        ])
```

eye

Creates an identity matrix

In [37]:
```
np.eye(4)
```

Out[37]:
```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Random

Numpy also has lots of ways to create random number arrays:

rand

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)` .

In [47]:
```
np.random.rand(2)
```

Out[47]:
```
array([ 0.11570539,  0.35279769])
```

In [46]:
```
np.random.rand(5,5)
```

Out[46]:
```
array([[ 0.66660768,  0.87589888,  0.12421056,  0.65074126,  0.60260888],
       [ 0.70027668,  0.85572434,  0.8464595 ,  0.2735416 ,  0.10955384],
       [ 0.0670566 ,  0.83267738,  0.9082729 ,  0.58249129,  0.12305748],
       [ 0.27948423,  0.66422017,  0.95639833,  0.34238788,  0.9578872 ],
       [ 0.72155386,  0.3035422 ,  0.85249683,  0.30414307,  0.79718816]])
```

randn

Return a sample (or samples) from the "standard normal" distribution. Unlike rand which is uniform:

In [48]:
```
np.random.randn(2)
```

Out[48]:
```
array([-0.27954018,  0.90078368])
```

In [45]:
```
np.random.randn(5,5)
```

Out[45]:
```
array([[ 0.70154515,  0.22441999,  1.33563186,  0.82872577, -0.28247509],
       [ 0.64489788,  0.61815094, -0.81693168, -0.30102424, -0.29030574],
       [ 0.8695976 ,  0.413755  ,  2.20047208,  0.17955692, -0.82159344],
       [ 0.59264235,  1.29869894, -1.18870241,  0.11590888, -0.09181687],
       [-0.96924265, -1.62888685, -2.05787102, -0.29705576,  0.68915542]])
```

randint

Return random integers from `low` (inclusive) to `high` (exclusive).

In [50]:
```
np.random.randint(1,100)
```

Out[50]:
```
44
```

In [51]:
```
np.random.randint(1,100,10)
```

Out[51]:
```
array([13, 64, 27, 63, 46, 68, 92, 10, 58, 24])
```

Array Attributes and Methods

Let's discuss some useful attributes and methods or an array:

In [55]:
```
arr = np.arange(25)
ranarr = np.random.randint(0,50,10)
```

In [56]:
```
arr
```

Out[56]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

In [57]:
```
ranarr
```

Out[57]:
```
array([10, 12, 41, 17, 49,  2, 46,  3, 19, 39])
```

Reshape

Returns an array containing the same data with a new shape.

In [54]:
```
arr.reshape(5,5)
```

Out[54]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

max,min,argmax,argmin

These are useful methods for finding max or min values. Or to find their index locations using argmin or argmax

In [64]:
```
ranarr
```

Out[64]:
```
array([10, 12, 41, 17, 49,  2, 46,  3, 19, 39])
```

In [61]:
```
ranarr.max()
```

Out[61]:
```
49
```

In [62]:
```
ranarr.argmax()
```

Out[62]:
```
4
```

In [63]:
```
ranarr.min()
```

Out[63]:
```
2
```

In [60]:
```
ranarr.argmin()
```

Out[60]:
```
5
```

Shape

Shape is an attribute that arrays have (not a method):

In [65]:
```
# Vector
arr.shape
```

Out[65]:
```
(25,)
```

In [66]:
```
# Notice the two sets of brackets
arr.reshape(1,25)
```

Out[66]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24]])
```

In [69]:
```
arr.reshape(1,25).shape
```

Out[69]:
```
(1, 25)
```

In [70]:
```
arr.reshape(25,1)
```

Out[70]:
```
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10],
       [11],
       [12],
       [13],
       [14],
       [15],
       [16],
       [17],
       [18],
       [19],
       [20],
       [21],
       [22],
       [23],
       [24]])
```

In [76]:
```
arr.reshape(25,1).shape
```

Out[76]:
```
(25, 1)
```

dtype

You can also grab the data type of the object in the array:

In [75]:
```
arr.dtype
```

Out[75]:
```
dtype('int64')
```

Great Job!


(http://www.pieriandata.com)

NumPy Indexing and Selection

In this lecture we will discuss how to select elements or groups of elements from an array.

In [2]:
```
import numpy as np
```

In [3]:
```
#Creating sample array
arr = np.arange(0,11)
```

In [4]:
```
#Show
arr
```

Out[4]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Bracket Indexing and Selection

The simplest way to pick one or some elements of an array looks very similar to python lists:

In [5]:
```
#Get a value at an index
arr[8]
```

Out[5]:
```
8
```

In [6]:
```
#Get values in a range
arr[1:5]
```

Out[6]:
```
array([1, 2, 3, 4])
```

In [7]:
```
#Get values in a range
arr[0:5]
```

Out[7]:
```
array([0, 1, 2, 3, 4])
```

Broadcasting

Numpy arrays differ from a normal Python list because of their ability to broadcast:

In [8]:
```
#Setting a value with index range (Broadcasting)
arr[0:5]=100

#Show
arr
```

Out[8]:
```
array([100, 100, 100, 100, 100,   5,   6,   7,   8,   9,  10])
```

In [9]:
```
# Reset array, we'll see why I had to reset in  a moment
arr = np.arange(0,11)

#Show
arr
```

Out[9]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [10]:
```
#Important notes on Slices
slice_of_arr = arr[0:6]

#Show slice
slice_of_arr
```

Out[10]:
```
array([0, 1, 2, 3, 4, 5])
```

In [11]:
```
#Change Slice
slice_of_arr[:]=99

#Show Slice again
slice_of_arr
```

Out[11]:
```
array([99, 99, 99, 99, 99, 99])
```

Now note the changes also occur in our original array!

In [12]:
```
arr
```

Out[12]:
```
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Data is not copied, it's a view of the original array! This avoids memory problems!

In [13]:
```
#To get a copy, need to be explicit
arr_copy = arr.copy()

arr_copy
```

Out[13]:
```
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Indexing a 2D array (matrices)

The general format is **arr_2d[row][col]** or **arr_2d[row,col]**. I recommend usually using the comma notation for clarity.

In [14]:
```
arr_2d = np.array(([5,10,15],[20,25,30],[35,40,45]))

#Show
arr_2d
```

Out[14]:
```
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])
```

In [15]:
```
#Indexing row
arr_2d[1]
```

Out[15]:
```
array([20, 25, 30])
```

In [16]:
```
# Format is arr_2d[row][col] or arr_2d[row,col]

# Getting individual element value
arr_2d[1][0]
```

Out[16]:
```
20
```

In [17]:
```
# Getting individual element value
arr_2d[1,0]
```

Out[17]:
```
20
```

In [18]:
```
# 2D array slicing

#Shape (2,2) from top right corner
arr_2d[:2,1:]
```

Out[18]:
```
array([[10, 15],
       [25, 30]])
```

In [19]:
```
#Shape bottom row
arr_2d[2]
```

Out[19]:
```
array([35, 40, 45])
```

In [20]:
```
#Shape bottom row
arr_2d[2,:]
```

Out[20]:
```
array([35, 40, 45])
```

Fancy Indexing

Fancy indexing allows you to select entire rows or columns out of order,to show this, let's quickly build out a numpy array:

In [21]:
```
#Set up matrix
arr2d = np.zeros((10,10))
```

In [22]:
```
#Length of array
arr_length = arr2d.shape[1]
```

In [23]:
```
#Set up array

for i in range(arr_length):
    arr2d[i] = i

arr2d
```

Out[23]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.],
       [ 8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.],
       [ 9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.]])
```

Fancy indexing allows the following

In [24]:
```
arr2d[[2,4,6,8]]
```

Out[24]:
```
array([[ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.],
       [ 6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.],
       [ 8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.]])
```

In [25]:
```
#Allows in any order
arr2d[[6,4,2,7]]
```

Out[25]:
```
array([[ 6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.],
       [ 4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.]])
```

More Indexing Help

Indexing a 2d matrix can be a bit confusing at first, especially when you start to add in step size. Try google image searching NumPy indexing to fins useful images, like this one:

Selection

Let's briefly go over how to use brackets for selection based off of comparison operators.

In [28]:
```
arr = np.arange(1,11)
arr
```

Out[28]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [30]:
```
arr > 4
```

Out[30]:
```
array([False, False, False, False,  True,  True,  True,  True,  True,  True],
dtype=bool)
```

In [31]:
```
bool_arr = arr>4
```

In [32]:
```
bool_arr
```

Out[32]:
```
array([False, False, False, False,  True,  True,  True,  True,  True,  True],
dtype=bool)
```

In [33]:
```
arr[bool_arr]
```

Out[33]:
```
array([ 5,  6,  7,  8,  9, 10])
```

In [34]:
```
arr[arr>2]
```

Out[34]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

In [37]:
```
x = 2
arr[arr>x]
```

Out[37]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

Great Job!


(http://www.pieriandata.com)

NumPy Operations

Arithmetic

You can easily perform array with array arithmetic, or scalar with array arithmetic. Let's see some examples:

In [1]:
```
import numpy as np
arr = np.arange(0,10)
```

In [2]:
```
arr + arr
```

Out[2]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [3]:
```
arr * arr
```

Out[3]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

In [4]:
```
arr - arr
```

Out[4]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [5]:
```
# Warning on division by zero, but not an error!
# Just replaced with nan
arr/arr
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/ipykernel/__main__.py:1: Ru
ntimeWarning: invalid value encountered in true_divide
  if __name__ == '__main__':
```

Out[5]:
```
array([ nan,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.])
```

In [6]:
```
# Also warning, but not an error instead infinity
1/arr
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/ipykernel/__main__.py:1: Ru
ntimeWarning: divide by zero encountered in true_divide
  if __name__ == '__main__':
```

Out[6]:
```
array([        inf,  1.        ,  0.5       ,  0.33333333,  0.25      ,
        0.2       ,  0.16666667,  0.14285714,  0.125     ,  0.11111111])
```

In [10]:
```
arr**3
```

Out[10]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

Universal Array Functions ¶

Numpy comes with many [universal array functions (http://docs.scipy.org/doc/numpy/reference/ufuncs.html)](), which are essentially just mathematical operations you can use to perform the operation across the array. Let's show some common ones:

In [12]:
```
#Taking Square Roots
np.sqrt(arr)
```

Out[12]:
```
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ])
```

In [13]:
```
#Calcualting exponential (e^)
np.exp(arr)
```

Out[13]:
```
array([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,
         8.10308393e+03])
```

In [14]:
```
np.max(arr) #same as arr.max()
```

Out[14]:
```
9
```

In [15]:
```
np.sin(arr)
```

Out[15]:
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

In [16]:
```
np.log(arr)
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/ipykernel/__main__.py:1: Ru
ntimeWarning: divide by zero encountered in log
  if __name__ == '__main__':
```

Out[16]:
```
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436,
        1.60943791,  1.79175947,  1.94591015,  2.07944154,  2.19722458])
```

Great Job!

That's all we need to know for now!


(http://www.pieriandata.com)

NumPy Exercises - Solutions

Now that we've learned about NumPy let's test your knowledge. We'll start off with a few simple tasks and then you'll be asked some more complicated questions.

Import NumPy as np

In [1]:
```
import numpy as np
```

Create an array of 10 zeros ¶

In [2]:
```
np.zeros(10)
```

Out[2]:
```
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

Create an array of 10 ones

In [3]:
```
np.ones(10)
```

Out[3]:
```
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Create an array of 10 fives

In [4]:
```
np.ones(10) * 5
```

Out[4]:
```
array([ 5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.])
```

Create an array of the integers from 10 to 50

In [5]:
```
np.arange(10,51)
```

Out[5]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
       44, 45, 46, 47, 48, 49, 50])
```

Create an array of all the even integers from 10 to 50

In [6]:
```
np.arange(10,51,2)
```

Out[6]:
```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
       44, 46, 48, 50])
```

Create a 3x3 matrix with values ranging from 0 to 8

In [7]:
```
np.arange(9).reshape(3,3)
```

Out[7]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Create a 3x3 identity matrix

In [8]:
```
np.eye(3)
```

Out[8]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Use NumPy to generate a random number between 0 and 1

In [15]:
```
np.random.rand(1)
```

Out[15]:
```
array([ 0.42829726])
```

Use NumPy to generate an array of 25 random numbers sampled from a standard normal distribution

In [33]:
```
np.random.randn(25)
```

Out[33]:
```
array([ 1.32031013,  1.6798602 , -0.42985892, -1.53116655,  0.85753232,
        0.87339938,  0.35668636, -1.47491157,  0.15349697,  0.99530727,
       -0.94865451, -1.69174783,  1.57525349, -0.70615234,  0.10991879,
       -0.49478947,  1.08279872,  0.76488333, -2.3039931 ,  0.35401124,
       -0.45454399, -0.64754649, -0.29391671,  0.02339861,  0.38272124])
```

Create the following matrix:

In [35]:
```
np.arange(1,101).reshape(10,10) / 100
```

Out[35]:
```
array([[ 0.01,  0.02,  0.03,  0.04,  0.05,  0.06,  0.07,  0.08,  0.09,  0.1
],
       [ 0.11,  0.12,  0.13,  0.14,  0.15,  0.16,  0.17,  0.18,  0.19,  0.2
],
       [ 0.21,  0.22,  0.23,  0.24,  0.25,  0.26,  0.27,  0.28,  0.29,  0.3
],
       [ 0.31,  0.32,  0.33,  0.34,  0.35,  0.36,  0.37,  0.38,  0.39,  0.4
],
       [ 0.41,  0.42,  0.43,  0.44,  0.45,  0.46,  0.47,  0.48,  0.49,  0.5
],
       [ 0.51,  0.52,  0.53,  0.54,  0.55,  0.56,  0.57,  0.58,  0.59,  0.6
],
       [ 0.61,  0.62,  0.63,  0.64,  0.65,  0.66,  0.67,  0.68,  0.69,  0.7
],
       [ 0.71,  0.72,  0.73,  0.74,  0.75,  0.76,  0.77,  0.78,  0.79,  0.8
],
       [ 0.81,  0.82,  0.83,  0.84,  0.85,  0.86,  0.87,  0.88,  0.89,  0.9
],
       [ 0.91,  0.92,  0.93,  0.94,  0.95,  0.96,  0.97,  0.98,  0.99,  1.
]])
```

Create an array of 20 linearly spaced points between 0 and 1:

In [36]:
```
np.linspace(0,1,20)
```

Out[36]:
```
array([ 0.        ,  0.05263158,  0.10526316,  0.15789474,  0.21052632,
        0.26315789,  0.31578947,  0.36842105,  0.42105263,  0.47368421,
        0.52631579,  0.57894737,  0.63157895,  0.68421053,  0.73684211,
        0.78947368,  0.84210526,  0.89473684,  0.94736842,  1.        ])
```

Numpy Indexing and Selection

Now you will be given a few matrices, and be asked to replicate the resulting matrix outputs:

In [38]:
```
mat = np.arange(1,26).reshape(5,5)
mat
```

Out[38]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

In [39]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [40]:
```
mat[2:,1:]
```

Out[40]:
```
array([[12, 13, 14, 15],
       [17, 18, 19, 20],
       [22, 23, 24, 25]])
```

In [29]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [41]:
```
mat[3,4]
```

Out[41]:
```
20
```

In [30]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [42]:
```
mat[:3,1:2]
```

Out[42]:
```
array([[ 2],
       [ 7],
       [12]])
```

In [31]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [46]:
```
mat[4,:]
```

Out[46]:
```
array([21, 22, 23, 24, 25])
```

In [32]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [49]:
```
mat[3:5,:]
```

Out[49]:
```
array([[16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

Now do the following

Get the sum of all the values in mat

In [50]:
```
mat.sum()
```

Out[50]:
```
325
```

Get the standard deviation of the values in mat

In [51]:
```
mat.std()
```

Out[51]:
```
7.2111025509279782
```

Get the sum of all the columns in mat

In [53]:
```
mat.sum(axis=0)
```

Out[53]:
```
array([55, 60, 65, 70, 75])
```

Great Job!


(http://www.pieriandata.com)

Introduction to Pandas

In this section of the course we will learn how to use pandas for data analysis. You can think of pandas as an extremely powerful version of Excel, with a lot more features. In this section of the course, you should go through the notebooks in this order:

- Introduction to Pandas
- Series
- DataFrames
- Missing Data
- GroupBy
- Merging,Joining,and Concatenating
- Operations
- Data Input and Output


(http://www.pieriandata.com)

Series

The first main data type we will learn about for pandas is the Series data type. Let's import Pandas and explore the Series object.

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object.

Let's explore this concept through some examples:

In [2]:
```
import numpy as np
import pandas as pd
```

Creating a Series

You can convert a list,numpy array, or dictionary to a Series:

In [3]:
```
labels = ['a','b','c']
my_list = [10,20,30]
arr = np.array([10,20,30])
d = {'a':10,'b':20,'c':30}
```

** Using Lists**

In [4]:
```
pd.Series(data=my_list)
```

Out[4]:
```
0    10
1    20
2    30
dtype: int64
```

In [5]:
```
pd.Series(data=my_list,index=labels)
```

Out[5]:
```
a    10
b    20
c    30
dtype: int64
```

In [6]:
```
pd.Series(my_list,labels)
```

Out[6]:
```
a    10
b    20
c    30
dtype: int64
```

** NumPy Arrays **

In [7]:
```
pd.Series(arr)
```

Out[7]:
```
0    10
1    20
2    30
dtype: int64
```

In [8]:
```
pd.Series(arr,labels)
```

Out[8]:
```
a    10
b    20
c    30
dtype: int64
```

** Dictionary**

In [9]:
```
pd.Series(d)
```

Out[9]:
```
a    10
b    20
c    30
dtype: int64
```

Data in a Series

A pandas Series can hold a variety of object types:

In [10]:
```
pd.Series(data=labels)
```

Out[10]:
```
0    a
1    b
2    c
dtype: object
```

In [11]:
```
# Even functions (although unlikely that you will use this)
pd.Series([sum,print,len])
```

Out[11]:
```
0      <built-in function sum>
1    <built-in function print>
2      <built-in function len>
dtype: object
```

Using an Index

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast look ups of information (works like a hash table or dictionary).

Let's see some examples of how to grab information from a Series. Let us create two sereis, ser1 and ser2:

In [12]:
```
ser1 = pd.Series([1,2,3,4],index = ['USA', 'Germany','USSR', 'Japan'])
```

In [13]:
```
ser1
```

Out[13]:
```
USA        1
Germany    2
USSR       3
Japan      4
dtype: int64
```

In [14]:
```
ser2 = pd.Series([1,2,5,4],index = ['USA', 'Germany','Italy', 'Japan'])
```

In [15]:
```
ser2
```

Out[15]:
```
USA        1
Germany    2
Italy      5
Japan      4
dtype: int64
```

In [16]:
```
ser1['USA']
```

Out[16]:
```
1
```

Operations are then also done based off of index:

In [17]:
```
ser1 + ser2
```

Out[17]:
```
Germany    4.0
Italy      NaN
Japan      8.0
USA        2.0
USSR       NaN
dtype: float64
```

Let's stop here for now and move on to DataFrames, which will expand on the concept of Series!

Great Job!


(http://www.pieriandata.com)

DataFrames

DataFrames are the workhorse of pandas and are directly inspired by the R programming language. We can think of a DataFrame as a bunch of Series objects put together to share the same index. Let's use pandas to explore this topic!

In [183]:
```
import pandas as pd
import numpy as np
```

In [184]:
```
from numpy.random import randn
np.random.seed(101)
```

In [185]:
```
df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y Z'.split
```

In [186]:
```
df
```

Out[186]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Selection and Indexing

Let's learn the various methods to grab data from a DataFrame

In [187]:
```
df['W']
```

Out[187]:
```
A    2.706850
B    0.651118
C   -2.018168
D    0.188695
E    0.190794
Name: W, dtype: float64
```

In [188]:
```
# Pass a list of column names
df[['W','Z']]
```

Out[188]:

	W	Z
A	2.706850	0.503826
B	0.651118	0.605965
C	-2.018168	-0.589001
D	0.188695	0.955057
E	0.190794	0.683509

In [189]:
```
# SQL Syntax (NOT RECOMMENDED!)
df.W
```

Out[189]:
```
A    2.706850
B    0.651118
C   -2.018168
D    0.188695
E    0.190794
Name: W, dtype: float64
```

DataFrame Columns are just Series

In [190]:
```
type(df['W'])
```

Out[190]:
```
pandas.core.series.Series
```

Creating a new column:

In [191]:
```
df['new'] = df['W'] + df['Y']
```

In [192]:
```
df
```

Out[192]:

	W	X	Y	Z	new
A	2.706850	0.628133	0.907969	0.503826	3.614819
B	0.651118	-0.319318	-0.848077	0.605965	-0.196959
C	-2.018168	0.740122	0.528813	-0.589001	-1.489355
D	0.188695	-0.758872	-0.933237	0.955057	-0.744542
E	0.190794	1.978757	2.605967	0.683509	2.796762

** Removing Columns**

In [193]:
```
df.drop('new',axis=1)
```

Out[193]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [194]:
```
# Not inplace unless specified!
df
```

Out[194]:

	W	X	Y	Z	new
A	2.706850	0.628133	0.907969	0.503826	3.614819
B	0.651118	-0.319318	-0.848077	0.605965	-0.196959
C	-2.018168	0.740122	0.528813	-0.589001	-1.489355
D	0.188695	-0.758872	-0.933237	0.955057	-0.744542
E	0.190794	1.978757	2.605967	0.683509	2.796762

In [195]:
```
df.drop('new',axis=1,inplace=True)
```

In [196]:
```
df
```

Out[196]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Can also drop rows this way:

In [197]:
```
df.drop('E',axis=0)
```

Out[197]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057

** Selecting Rows**

In [198]:
```
df.loc['A']
```

Out[198]:
```
W    2.706850
X    0.628133
Y    0.907969
Z    0.503826
Name: A, dtype: float64
```

Or select based off of position instead of label

In [199]:
```
df.iloc[2]
```

Out[199]:
```
W   -2.018168
X    0.740122
Y    0.528813
Z   -0.589001
Name: C, dtype: float64
```

** Selecting subset of rows and columns **

In [200]:
```
df.loc['B','Y']
```

Out[200]:
```
-0.84807698340363147
```

In [201]:
```
df.loc[['A','B'],['W','Y']]
```

Out[201]:

	W	Y
A	2.706850	0.907969
B	0.651118	-0.848077

Conditional Selection

An important feature of pandas is conditional selection using bracket notation, very similar to numpy:

In [202]:
```
df
```

Out[202]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [203]:
```
df>0
```

Out[203]:

	W	X	Y	Z
A	True	True	True	True
B	True	False	False	True
C	False	True	True	False
D	True	False	False	True
E	True	True	True	True

In [204]:
```
df[df>0]
```

Out[204]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	NaN	NaN	0.605965
C	NaN	0.740122	0.528813	NaN
D	0.188695	NaN	NaN	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [205]:
```
df[df['W']>0]
```

Out[205]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [206]:
```
df[df['W']>0]['Y']
```

Out[206]:
```
A    0.907969
B   -0.848077
D   -0.933237
E    2.605967
Name: Y, dtype: float64
```

In [207]:
```
df[df['W']>0][['Y','X']]
```

Out[207]:

	Y	X
A	0.907969	0.628133
B	-0.848077	-0.319318
D	-0.933237	-0.758872
E	2.605967	1.978757

For two conditions you can use | and & with parenthesis:

In [208]:
```
df[(df['W']>0) & (df['Y'] > 1)]
```

Out[208]:

	W	X	Y	Z
E	0.190794	1.978757	2.605967	0.683509

More Index Details

Let's discuss some more features of indexing, including resetting the index or setting it something else. We'll also talk about index hierarchy!

In [209]:
```
df
```

Out[209]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [210]:
```
# Reset to default 0,1...n index
df.reset_index()
```

Out[210]:

	index	W	X	Y	Z
0	A	2.706850	0.628133	0.907969	0.503826
1	B	0.651118	-0.319318	-0.848077	0.605965
2	C	-2.018168	0.740122	0.528813	-0.589001
3	D	0.188695	-0.758872	-0.933237	0.955057
4	E	0.190794	1.978757	2.605967	0.683509

In [211]:
```
newind = 'CA NY WY OR CO'.split()
```

In [212]:
```
df['States'] = newind
```

In [213]:
```
df
```

Out[213]:

	W	X	Y	Z	States
A	2.706850	0.628133	0.907969	0.503826	CA
B	0.651118	-0.319318	-0.848077	0.605965	NY
C	-2.018168	0.740122	0.528813	-0.589001	WY
D	0.188695	-0.758872	-0.933237	0.955057	OR
E	0.190794	1.978757	2.605967	0.683509	CO

In [214]:
```
df.set_index('States')
```

Out[214]:

States	W	X	Y	Z
CA	2.706850	0.628133	0.907969	0.503826
NY	0.651118	-0.319318	-0.848077	0.605965
WY	-2.018168	0.740122	0.528813	-0.589001
OR	0.188695	-0.758872	-0.933237	0.955057
CO	0.190794	1.978757	2.605967	0.683509

In [215]:
```
df
```

Out[215]:

	W	X	Y	Z	States
A	2.706850	0.628133	0.907969	0.503826	CA
B	0.651118	-0.319318	-0.848077	0.605965	NY
C	-2.018168	0.740122	0.528813	-0.589001	WY
D	0.188695	-0.758872	-0.933237	0.955057	OR
E	0.190794	1.978757	2.605967	0.683509	CO

In [216]:
```
df.set_index('States',inplace=True)
```

In [218]:
```
df
```

Out[218]:

States	W	X	Y	Z
CA	2.706850	0.628133	0.907969	0.503826
NY	0.651118	-0.319318	-0.848077	0.605965
WY	-2.018168	0.740122	0.528813	-0.589001
OR	0.188695	-0.758872	-0.933237	0.955057
CO	0.190794	1.978757	2.605967	0.683509

Multi-Index and Index Hierarchy

Let us go over how to work with Multi-Index, first we'll create a quick example of what a Multi-Indexed DataFrame would look like:

In [253]:
```
# Index Levels
outside = ['G1','G1','G1','G2','G2','G2']
inside = [1,2,3,1,2,3]
hier_index = list(zip(outside,inside))
hier_index = pd.MultiIndex.from_tuples(hier_index)
```

In [254]:
```
hier_index
```

Out[254]:
```
MultiIndex(levels=[['G1', 'G2'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

In [257]:
```
df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
```

Out[257]:

		A	B
G1	1	0.153661	0.167638
	2	-0.765930	0.962299
	3	0.902826	-0.537909
G2	1	-1.549671	0.435253
	2	1.259904	-0.447898
	3	0.266207	0.412580

Now let's show how to index this! For index hierarchy we use df.loc[], if this was on the columns axis, you would just use normal bracket notation df[]. Calling one level of the index returns the sub-dataframe:

In [260]:
```
df.loc['G1']
```

Out[260]:

	A	B
1	0.153661	0.167638
2	-0.765930	0.962299
3	0.902826	-0.537909

In [263]:
```
df.loc['G1'].loc[1]
```

Out[263]:
```
A    0.153661
B    0.167638
Name: 1, dtype: float64
```

In [265]:
```
df.index.names
```

Out[265]:
```
FrozenList([None, None])
```

In [266]:
```
df.index.names = ['Group','Num']
```

In [267]:
```
df
```

Out[267]:

		A	B
Group	**Num**		
G1	**1**	0.153661	0.167638
	2	-0.765930	0.962299
	3	0.902826	-0.537909
G2	**1**	-1.549671	0.435253
	2	1.259904	-0.447898
	3	0.266207	0.412580

In [270]:
```
df.xs('G1')
```

Out[270]:

	A	B
Num		
1	0.153661	0.167638
2	-0.765930	0.962299
3	0.902826	-0.537909

In [271]:
```
df.xs(['G1',1])
```

Out[271]:
```
A    0.153661
B    0.167638
Name: (G1, 1), dtype: float64
```

In [273]:
```
df.xs(1,level='Num')
```

Out[273]:

	A	B
Group		
G1	0.153661	0.167638
G2	-1.549671	0.435253

Great Job!


(http://www.pieriandata.com)

Missing Data

Let's show a few convenient methods to deal with Missing Data in pandas:

In [1]:
```
import numpy as np
import pandas as pd
```

In [9]:
```
df = pd.DataFrame({'A':[1,2,np.nan],
                   'B':[5,np.nan,np.nan],
                   'C':[1,2,3]})
```

In [10]:
```
df
```

Out[10]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

In [12]:
```
df.dropna()
```

Out[12]:

	A	B	C
0	1.0	5.0	1

In [13]:
```
df.dropna(axis=1)
```

Out[13]:

	C
0	1
1	2
2	3

In [14]:
```
df.dropna(thresh=2)
```

Out[14]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2

In [15]:
```
df.fillna(value='FILL VALUE')
```

Out[15]:

	A	B	C
0	1	5	1
1	2	FILL VALUE	2
2	FILL VALUE	FILL VALUE	3

In [17]:
```
df['A'].fillna(value=df['A'].mean())
```

Out[17]:
```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

Great Job!


(http://www.pieriandata.com)

Groupby

The groupby method allows you to group rows of data together and call aggregate functions

In [31]:
```
import pandas as pd
# Create dataframe
data = {'Company':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
       'Person':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
       'Sales':[200,120,340,124,243,350]}
```

In [32]:
```
df = pd.DataFrame(data)
```

In [33]:
```
df
```

Out[33]:

	Company	Person	Sales
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350

** Now you can use the .groupby() method to group rows together based off of a column name. For instance let's group based off of Company. This will create a DataFrameGroupBy object:**

In [34]:
```
df.groupby('Company')
```

Out[34]:
```
<pandas.core.groupby.DataFrameGroupBy object at 0x113014128>
```

You can save this object as a new variable:

In [35]:
```
by_comp = df.groupby("Company")
```

And then call aggregate methods off the object:

In [36]:
```
by_comp.mean()
```

Out[36]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

In [37]:
```
df.groupby('Company').mean()
```

Out[37]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

More examples of aggregate methods:

In [38]:
```
by_comp.std()
```

Out[38]:

Company	Sales
FB	75.660426
GOOG	56.568542
MSFT	152.735065

In [39]:
```
by_comp.min()
```

Out[39]:

Company	Person	Sales
FB	Carl	243
GOOG	Charlie	120
MSFT	Amy	124

In [40]:
```
by_comp.max()
```

Out[40]:

Company	Person	Sales
FB	Sarah	350
GOOG	Sam	200
MSFT	Vanessa	340

In [41]:
```
by_comp.count()
```

Out[41]:

Company	Person	Sales
FB	2	2
GOOG	2	2
MSFT	2	2

In [42]:
```
by_comp.describe()
```

Out[42]:

Company		Sales
FB	count	2.000000
	mean	296.500000
	std	75.660426
	min	243.000000
	25%	269.750000
	50%	296.500000
	75%	323.250000
	max	350.000000
GOOG	count	2.000000
	mean	160.000000
	std	56.568542
	min	120.000000
	25%	140.000000
	50%	160.000000
	75%	180.000000
	max	200.000000
MSFT	count	2.000000
	mean	232.000000
	std	152.735065
	min	124.000000
	25%	178.000000
	50%	232.000000
	75%	286.000000
	max	340.000000

In [43]:
```
by_comp.describe().transpose()
```

Out[43]:

Company	FB								C			
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%
Sales	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0	2.0	160.0	...	180.0

1 rows × 24 columns

In [44]:
```
by_comp.describe().transpose()['GOOG']
```

Out[44]:

	count	mean	std	min	25%	50%	75%	max
Sales	2.0	160.0	56.568542	120.0	140.0	160.0	180.0	200.0

Great Job!


(http://www.pieriandata.com)

Merging, Joining, and Concatenating

There are 3 main ways of combining DataFrames together: Merging, Joining and Concatenating. In this lecture we will discuss these 3 methods with examples.

Example DataFrames

In [3]:
```
import pandas as pd
```

In [4]:
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])
```

In [5]:
```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                     index=[4, 5, 6, 7])
```

In [6]:
```
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                   index=[8, 9, 10, 11])
```

In [7]:
```
df1
```

Out[7]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

In [8]:
```
df2
```

Out[8]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In [12]:
```
df3
```

Out[12]:

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Concatenation

Concatenation basically glues together DataFrames. Keep in mind that dimensions should match along the axis you are concatenating on. You can use **pd.concat** and pass in a list of DataFrames to concatenate together:

In [10]:
```
pd.concat([df1,df2,df3])
```

Out[10]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

In [18]:
```
pd.concat([df1,df2,df3],axis=1)
```

Out[18]:

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

Example DataFrames

In [28]:
```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})
```

In [29]:
```
left
```

Out[29]:

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	B3	K3

In [30]:
```
right
```

Out[30]:

	C	D	key
0	C0	D0	K0
1	C1	D1	K1
2	C2	D2	K2
3	C3	D3	K3

Merging

The **merge** function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. For example:

In [35]:
```
pd.merge(left,right,how='inner',on='key')
```

Out[35]:

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K2	C2	D2
3	A3	B3	K3	C3	D3

Or to show a more complicated example:

In [37]:
```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                        'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                                'key2': ['K0', 'K0', 'K0', 'K0'],
                                   'C': ['C0', 'C1', 'C2', 'C3'],
                                   'D': ['D0', 'D1', 'D2', 'D3']})
```

In [39]:
```
pd.merge(left, right, on=['key1', 'key2'])
```

Out[39]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2

In [40]:
```
pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

Out[40]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A2	B2	K1	K0	C2	D2
4	A3	B3	K2	K1	NaN	NaN
5	NaN	NaN	K2	K0	C3	D3

In [41]:
```
pd.merge(left, right, how='right', on=['key1', 'key2'])
```

Out[41]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2
3	NaN	NaN	K2	K0	C3	D3

In [42]:
```
pd.merge(left, right, how='left', on=['key1', 'key2'])
```

Out[42]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A2	B2	K1	K0	C2	D2
4	A3	B3	K2	K1	NaN	NaN

Joining

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

In [46]:
```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                     index=['K0', 'K1', 'K2'])

right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                      'D': ['D0', 'D2', 'D3']},
                     index=['K0', 'K2', 'K3'])
```

In [47]:
```
left.join(right)
```

Out[47]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

In [48]:
```
left.join(right, how='outer')
```

Out[48]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

Great Job!


(http://www.pieriandata.com)

Operations

There are lots of operations with pandas that will be really useful to you, but don't fall into any distinct category. Let's show them here in this lecture:

In [52]:
```
import pandas as pd
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[444,555,666,444],'col3':['abc','de
df.head()
```

Out[52]:

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

Info on Unique Values

In [53]:
```
df['col2'].unique()
```

Out[53]:
```
array([444, 555, 666])
```

In [54]:
```
df['col2'].nunique()
```

Out[54]:
```
3
```

In [55]:
```
df['col2'].value_counts()
```

Out[55]:
```
444    2
555    1
666    1
Name: col2, dtype: int64
```

Selecting Data

In [56]:
```
#Select from DataFrame using criteria from multiple columns
newdf = df[(df['col1']>2) & (df['col2']==444)]
```

In [57]:
```
newdf
```

Out[57]:

	col1	col2	col3
3	4	444	xyz

Applying Functions

In [58]:
```
def times2(x):
    return x*2
```

In [59]:
```
df['col1'].apply(times2)
```

Out[59]:
```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

In [60]:
```
df['col3'].apply(len)
```

Out[60]:
```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

In [61]:
```
df['col1'].sum()
```

Out[61]:
```
10
```

** Permanently Removing a Column**

In [62]:
```
del df['col1']
```

In [63]:
```
df
```

Out[63]:

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

** Get column and index names: **

In [64]:
```
df.columns
```

Out[64]:
```
Index(['col2', 'col3'], dtype='object')
```

In [65]:
```
df.index
```

Out[65]:
```
RangeIndex(start=0, stop=4, step=1)
```

** Sorting and Ordering a DataFrame:**

In [66]:
```
df
```

Out[66]:

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

In [67]:
```
df.sort_values(by='col2') #inplace=False by default
```

Out[67]:

	col2	col3
0	444	abc
3	444	xyz
1	555	def
2	666	ghi

** Find Null Values or Check for Null Values**

In [68]:
```
df.isnull()
```

Out[68]:

	col2	col3
0	False	False
1	False	False
2	False	False
3	False	False

In [69]:
```
# Drop rows with NaN Values
df.dropna()
```

Out[69]:

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

** Filling in NaN values with something else: **

In [71]:
```
import numpy as np
```

In [72]:
```
df = pd.DataFrame({'col1':[1,2,3,np.nan],
                   'col2':[np.nan,555,666,444],
                   'col3':['abc','def','ghi','xyz']})
df.head()
```

Out[72]:

	col1	col2	col3
0	1.0	NaN	abc
1	2.0	555.0	def
2	3.0	666.0	ghi
3	NaN	444.0	xyz

In [75]:
```
df.fillna('FILL')
```

Out[75]:

	col1	col2	col3
0	1	FILL	abc
1	2	555	def
2	3	666	ghi
3	FILL	444	xyz

In [89]:
```
data = {'A':['foo','foo','foo','bar','bar','bar'],
     'B':['one','one','two','two','one','one'],
       'C':['x','y','x','y','x','y'],
       'D':[1,3,2,5,4,1]}

df = pd.DataFrame(data)
```

In [90]:
```
df
```

Out[90]:

	A	B	C	D
0	foo	one	x	1
1	foo	one	y	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	4
5	bar	one	y	1

In [91]:
```
df.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

Out[91]:

		C	x	y
A	**B**			
bar	**one**	4.0	1.0
	two	NaN	5.0
foo	**one**	1.0	3.0
	two	2.0	NaN

Great Job!


(http://www.pieriandata.com)

Data Input and Output

This notebook is the reference code for getting input and output, pandas can read a variety of file types using its pd.read_ methods. Let's take a look at the most common data types:

In [1]:
```
import numpy as np
import pandas as pd
```

CSV

CSV Input

In [25]:
```
df = pd.read_csv('example')
df
```

Out[25]:

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

CSV Output

In [24]:
```
df.to_csv('example',index=False)
```

Excel

Pandas can read and write excel files, keep in mind, this only imports data. Not formulas or images, having images or macros may cause this read_excel method to crash.

Excel Input

In [35]:
```
pd.read_excel('Excel_Sample.xlsx',sheetname='Sheet1')
```

Out[35]:

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Excel Output

In [33]:
```
df.to_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
```

HTML

You may need to install htmllib5,lxml, and BeautifulSoup4. In your terminal/command prompt run:

```
conda install lxml
conda install html5lib
conda install BeautifulSoup4
```

Then restart Jupyter Notebook. (or use pip install if you aren't using the Anaconda Distribution)

Pandas can read table tabs off of html. For example:

HTML Input

Pandas read_html function will read tables off of a webpage and return a list of DataFrame objects:

In [5]:
```
df = pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')
```

In [7]:
```
df[0]
```

Out[7]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date	Loss Share Type	Ac Te
0	First CornerStone Bank	King of Prussia	PA	35312	First-Citizens Bank & Trust Company	May 6, 2016	July 12, 2016	none	
1	Trust Company Bank	Memphis	TN	9956	The Bank of Fayette County	April 29, 2016	August 4, 2016	none	
2	North Milwaukee State Bank	Milwaukee	WI	20364	First-Citizens Bank & Trust Company	March 11, 2016	June 16, 2016	none	
3	Hometown National Bank	Longview	WA	35156	Twin City Bank	October 2, 2015	April 13, 2016	none	
4	The Bank of Georgia	Peachtree City	GA	35259	Fidelity Bank	October 2, 2015	April 13, 2016	none	
5					United Fidelity	July 10,	July 12,		

SQL (Optional)

- Note: If you are completely unfamiliar with SQL you can check out my other course: "Complete SQL Bootcamp" to learn SQL.

The pandas.io.sql module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Python's standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some cookbook examples for some advanced strategies.

The key functions are:

- read_sql_table(table_name, con[, schema, ...])
  - Read SQL database table into a DataFrame.
- read_sql_query(sql, con[, index_col, ...])

    - Read SQL query into a DataFrame.
- read_sql(sql, con[, index_col, ...])
    - Read SQL query or database table into a DataFrame.
- DataFrame.to_sql(name, con[, flavor, ...])
    - Write records stored in a DataFrame to a SQL database.

In [36]:
```
from sqlalchemy import create_engine
```

In [37]:
```
engine = create_engine('sqlite:///:memory:')
```

In [40]:
```
df.to_sql('data', engine)
```

In [42]:
```
sql_df = pd.read_sql('data',con=engine)
```

In [43]:
```
sql_df
```

Out[43]:

	index	a	b	c	d
0	0	0	1	2	3
1	1	4	5	6	7
2	2	8	9	10	11
3	3	12	13	14	15

Great Job!


(http://www.pieriandata.com)

SF Salaries Exercise - Solutions

Welcome to a quick exercise for you to practice your pandas skills! We will be using the SF Salaries Dataset (https://www.kaggle.com/kaggle/sf-salaries) from Kaggle! Just follow along and complete the tasks outlined in bold below. The tasks will get harder and harder as you go along.

** Import pandas as pd.**

In [6]:
```
import pandas as pd
```

** Read Salaries.csv as a dataframe called sal.**

In [7]:
```
sal = pd.read_csv('Salaries.csv')
```

** Check the head of the DataFrame. **

In [8]:
```
sal.head()
```

Out[8]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay
0	1	NATHANIEL FORD	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.00	400184.25	NaN	567595.43
1	2	GARY JIMENEZ	CAPTAIN III (POLICE DEPARTMENT)	155966.02	245131.88	137811.38	NaN	538909.28
2	3	ALBERT PARDINI	CAPTAIN III (POLICE DEPARTMENT)	212739.13	106088.18	16452.60	NaN	335279.91
3	4	CHRISTOPHER CHONG	WIRE ROPE CABLE MAINTENANCE MECHANIC	77916.00	56120.71	198306.90	NaN	332343.61
4	5	PATRICK GARDNER	DEPUTY CHIEF OF DEPARTMENT, (FIRE DEPARTMENT)	134401.60	9737.00	182234.59	NaN	326373.19

** Use the .info() method to find out how many entries there are.**

In [9]:
```
sal.info() # 148654 Entries
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 148654 entries, 0 to 148653
Data columns (total 13 columns):
Id                  148654 non-null int64
EmployeeName        148654 non-null object
JobTitle            148654 non-null object
BasePay             148045 non-null float64
OvertimePay         148650 non-null float64
OtherPay            148650 non-null float64
Benefits            112491 non-null float64
TotalPay            148654 non-null float64
TotalPayBenefits    148654 non-null float64
Year                148654 non-null int64
Notes               0 non-null float64
Agency              148654 non-null object
Status              0 non-null float64
dtypes: float64(8), int64(2), object(3)
memory usage: 14.7+ MB
```

What is the average BasePay ?

In [10]:
```
sal['BasePay'].mean()
```

Out[10]:
```
66325.44884050643
```

** What is the highest amount of OvertimePay in the dataset ? **

In [11]:
```
sal['OvertimePay'].max()
```

Out[11]:
```
245131.88
```

** What is the job title of JOSEPH DRISCOLL ? Note: Use all caps, otherwise you may get an answer that doesn't match up (there is also a lowercase Joseph Driscoll). **

In [12]:
```
sal[sal['EmployeeName']=='JOSEPH DRISCOLL']['JobTitle']
```

Out[12]:
```
24    CAPTAIN, FIRE SUPPRESSION
Name: JobTitle, dtype: object
```

** How much does JOSEPH DRISCOLL make (including benefits)? **

In [13]:
```
sal[sal['EmployeeName']=='JOSEPH DRISCOLL']['TotalPayBenefits']
```

Out[13]:
```
24    270324.91
Name: TotalPayBenefits, dtype: float64
```

** What is the name of highest paid person (including benefits)?**

In [14]:
```
sal[sal['TotalPayBenefits']== sal['TotalPayBenefits'].max()] #['EmployeeName']
# or
# sal.loc[sal['TotalPayBenefits'].idxmax()]
```

Out[14]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay
0	1	NATHANIEL FORD	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.0	400184.25	NaN	567595.43

** What is the name of lowest paid person (including benefits)? Do you notice something strange about how much he or she is paid?**

In [15]:
```
sal[sal['TotalPayBenefits']== sal['TotalPayBenefits'].min()] #['EmployeeName']
# or
# sal.loc[sal['TotalPayBenefits'].idxmax()]['EmployeeName']

## ITS NEGATIVE!! VERY STRANGE
```

Out[15]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay
148653	148654	Joe Lopez	Counselor, Log Cabin Ranch	0.0	0.0	-618.13	0.0	-618.13

** What was the average (mean) BasePay of all employees per year? (2011-2014) ? **

In [16]:
```
sal.groupby('Year').mean()['BasePay']
```

Out[16]:
```
Year
2011    63595.956517
2012    65436.406857
2013    69630.030216
2014    66564.421924
Name: BasePay, dtype: float64
```

** How many unique job titles are there? **

In [17]:
```
sal['JobTitle'].nunique()
```

Out[17]:
```
2159
```

** What are the top 5 most common jobs? **

In [18]:
```
sal['JobTitle'].value_counts().head(5)
```

Out[18]:
```
Transit Operator                7036
Special Nurse                   4389
Registered Nurse                3736
Public Svc Aide-Public Works    2518
Police Officer 3                2421
Name: JobTitle, dtype: int64
```

** How many Job Titles were represented by only one person in 2013? (e.g. Job Titles with only one occurence in 2013?) **

In [19]:
```
sum(sal[sal['Year']==2013]['JobTitle'].value_counts() == 1) # pretty tricky wa
```

Out[19]:
```
202
```

** How many people have the word Chief in their job title? (This is pretty tricky) **

In [20]:
```
def chief_string(title):
    if 'chief' in title.lower():
        return True
    else:
        return False
```

In [21]:
```
sum(sal['JobTitle'].apply(lambda x: chief_string(x)))
```

Out[21]:
```
477
```

** Bonus: Is there a correlation between length of the Job Title string and Salary? **

In [22]:
```
sal['title_len'] = sal['JobTitle'].apply(len)
```

In [23]:
```
sal[['title_len','TotalPayBenefits']].corr() # No correlation.
```

Out[23]:

	title_len	TotalPayBenefits
title_len	1.000000	-0.036878
TotalPayBenefits	-0.036878	1.000000

Great Job!


(http://www.pieriandata.com)

Ecommerce Purchases Exercise - Solutions

In this Exercise you will be given some Fake Data about some purchases done through Amazon! Just go ahead and follow the directions and try your best to answer the questions and complete the tasks. Feel free to reference the solutions. Most of the tasks can be solved in different ways. For the most part, the questions get progressively harder.

Please excuse anything that doesn't make "Real-World" sense in the dataframe, all the data is fake and made-up.

Also note that all of these questions can be answered with one line of code.

** Import pandas and read in the Ecommerce Purchases csv file and set it to a DataFrame called ecom. **

In [84]:
```
import pandas as pd
```

In [86]:
```
ecom = pd.read_csv('Ecommerce Purchases')
```

Check the head of the DataFrame.

In [87]:
```
ecom.head()
```

Out[87]:

Lot	AM or PM	Browser Info	Company	Credit Card	CC Exp Date	CC Security Code	CC Provider	
46 in	PM	Opera/9.56. (X11; Linux x86_64; sl-SI) Presto/2...	Martinez-Herman	6011929061123406	02/20	900	JCB 16 digit	pdunlap@ya
28 rn	PM	Opera/8.93. (Windows 98; Win 9x 4.90; en-US) Pr...	Fletcher, Richards and Whitaker	3337758169645356	11/18	561	Mastercard	anthony41@r
94 vE	PM	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT ...	Simpson, Williams and Pham	675957666125	08/19	699	JCB 16 digit	amymiller@ harri
36 vm	PM	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_0 ...	Williams, Marshall and Buchanan	6011578504430710	02/24	384	Discover	brent16@olson-robi
20 IE	AM	Opera/9.58. (X11; Linux x86_64; it-IT) Presto/2...	Brown, Watson and Andrews	6011456623207998	10/25	678	Diners Club / Carte Blanche	christopherwright@g

** How many rows and columns are there? **

In [88]:
```
ecom.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
Address             10000 non-null object
Lot                 10000 non-null object
AM or PM            10000 non-null object
Browser Info        10000 non-null object
Company             10000 non-null object
Credit Card         10000 non-null int64
CC Exp Date         10000 non-null object
CC Security Code    10000 non-null int64
CC Provider         10000 non-null object
Email               10000 non-null object
Job                 10000 non-null object
IP Address          10000 non-null object
Language            10000 non-null object
Purchase Price      10000 non-null float64
dtypes: float64(1), int64(2), object(11)
memory usage: 1.1+ MB
```

** What is the average Purchase Price? **

In [90]:
```
ecom['Purchase Price'].mean()
```

Out[90]:
```
50.34730200000025
```

** What were the highest and lowest purchase prices? **

In [92]:
```
ecom['Purchase Price'].max()
```

Out[92]:
```
99.989999999999995
```

In [93]:
```
ecom['Purchase Price'].min()
```

Out[93]:
```
0.0
```

** How many people have English 'en' as their Language of choice on the website? **

In [94]:
```
ecom[ecom['Language']=='en'].count()
```

Out[94]:
```
Address             1098
Lot                 1098
AM or PM            1098
Browser Info        1098
Company             1098
Credit Card         1098
CC Exp Date         1098
CC Security Code    1098
CC Provider         1098
Email               1098
Job                 1098
IP Address          1098
Language            1098
Purchase Price      1098
dtype: int64
```

** How many people have the job title of "Lawyer" ? **

In [95]:
```
ecom[ecom['Job'] == 'Lawyer'].info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30 entries, 470 to 9979
Data columns (total 14 columns):
Address             30 non-null object
Lot                 30 non-null object
AM or PM            30 non-null object
Browser Info        30 non-null object
Company             30 non-null object
Credit Card         30 non-null int64
CC Exp Date         30 non-null object
CC Security Code    30 non-null int64
CC Provider         30 non-null object
Email               30 non-null object
Job                 30 non-null object
IP Address          30 non-null object
Language            30 non-null object
Purchase Price      30 non-null float64
dtypes: float64(1), int64(2), object(11)
memory usage: 3.5+ KB
```

** How many people made the purchase during the AM and how many people made the purchase during PM ? **

*(Hint: Check out [value_counts() (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.value_counts.html)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.value_counts.html)) *

In [96]:
```
ecom['AM or PM'].value_counts()
```

Out[96]:
```
PM    5068
AM    4932
Name: AM or PM, dtype: int64
```

** What are the 5 most common Job Titles? **

In [97]:
```
ecom['Job'].value_counts().head(5)
```

Out[97]:
```
Interior and spatial designer    31
Lawyer                           30
Social researcher                28
Purchasing manager               27
Designer, jewellery              27
Name: Job, dtype: int64
```

** Someone made a purchase that came from Lot: "90 WT" , what was the Purchase Price for this transaction? **

In [99]:
```
ecom[ecom['Lot']=='90 WT']['Purchase Price']
```

Out[99]:
```
513    75.1
Name: Purchase Price, dtype: float64
```

** What is the email of the person with the following Credit Card Number: 4926535242672853 **

In [100]:
```
ecom[ecom["Credit Card"] == 4926535242672853]['Email']
```

Out[100]:
```
1234    bondellen@williams-garza.com
Name: Email, dtype: object
```

** How many people have American Express as their Credit Card Provider *and* made a purchase above $95 ?**

In [101]:
```
ecom[(ecom['CC Provider']=='American Express') & (ecom['Purchase Price']>95)].
```

Out[101]:
```
Address             39
Lot                 39
AM or PM            39
Browser Info        39
Company             39
Credit Card         39
CC Exp Date         39
CC Security Code    39
CC Provider         39
Email               39
Job                 39
IP Address          39
Language            39
Purchase Price      39
dtype: int64
```

** Hard: How many people have a credit card that expires in 2025? **

In [102]:
```
sum(ecom['CC Exp Date'].apply(lambda x: x[3:]) == '25')
```

Out[102]:
```
1033
```

** Hard: What are the top 5 most popular email providers/hosts (e.g. gmail.com, yahoo.com, etc...) **

In [56]:
```
ecom['Email'].apply(lambda x: x.split('@')[1]).value_counts().head(5)
```

Out[56]:
```
hotmail.com     1638
yahoo.com       1616
gmail.com       1605
smith.com         42
williams.com      37
Name: Email, dtype: int64
```

Great Job!


(http://www.pieriandata.com)

Matplotlib Overview Lecture

Introduction

Matplotlib is the "grandfather" library of data visualization with Python. It was created by John Hunter. He created it to try to replicate MatLab's (another programming language) plotting capabilities in Python. So if you happen to be familiar with matlab, matplotlib will feel natural to you.

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
- Very customizable in general

Matplotlib allows you to create reproducible figures programmatically. Let's learn how to use it! Before continuing this lecture, I encourage you just to 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 under the name `plt` (the tidy way):

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

You'll also need to use this line to see plots in the notebook:

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

That line is only for jupyter notebooks, if you are using another editor, you'll use: **plt.show()** at the end of all your plotting commands to have the figure pop up in another window.

Basic Example

Let's walk through a very simple example using two numpy arrays:

Example

Let's walk through a very simple example using two numpy arrays. You can also use lists, but most likely you'll be passing numpy arrays or pandas columns (which essentially also behave like arrays).

** The data we want to plot:**

In [7]:
```
import numpy as np
x = np.linspace(0, 5, 11)
y = x ** 2
```

In [8]:
```
x
```

Out[8]:
```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ])
```

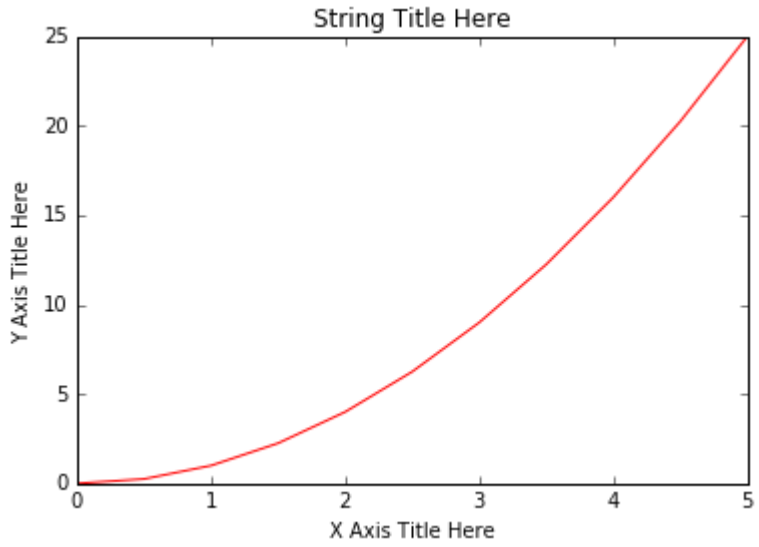
In [10]:
```
y
```

Out[10]:
```
array([  0.  ,   0.25,   1.  ,   2.25,   4.  ,   6.25,   9.  ,  12.25,
        16.  ,  20.25,  25.  ])
```

Basic Matplotlib Commands

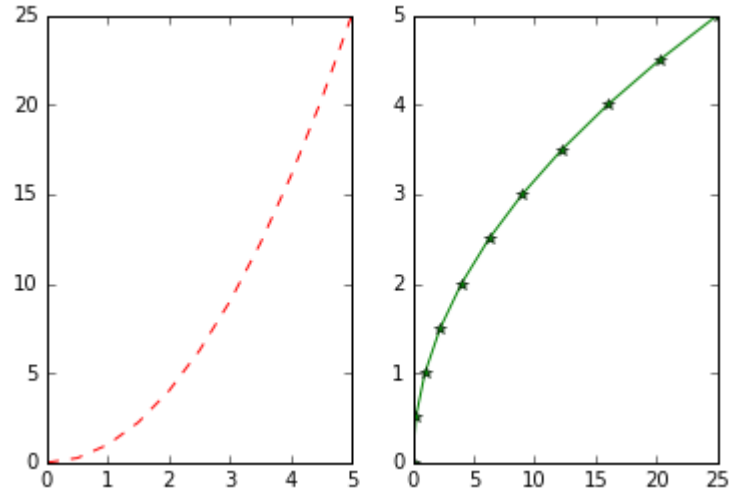
We can create a very simple line plot using the following (I encourage you to pause and use Shift+Tab along the way to check out the document strings for the functions we are using).

In [13]:
```
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 [14]:
```
# 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

Now that we've seen the basics, let's break it all down with a more formal introduction of Matplotlib's Object Oriented API. This means we will instantiate figure objects and then call

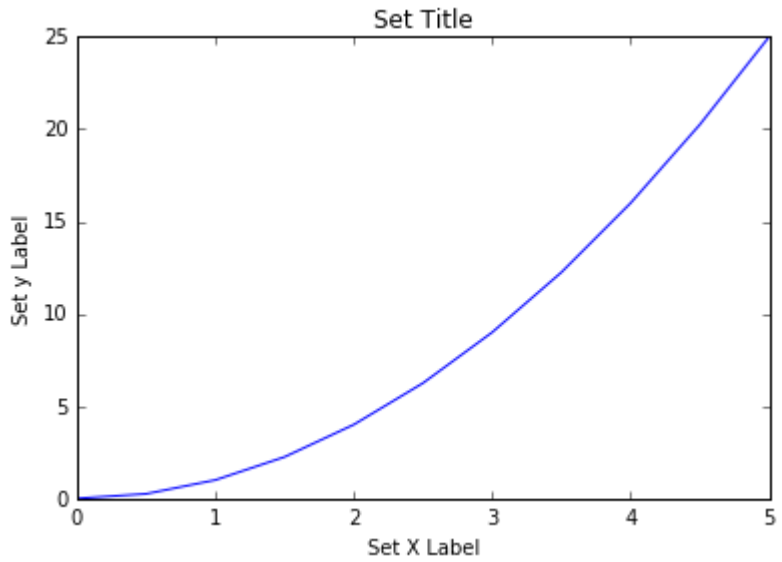
Introduction to the Object Oriented Method

The main idea in using the more formal Object Oriented method is to create figure objects and then just call methods or attributes off of that object. This approach is nicer when dealing with a canvas that has multiple plots on it.

To begin we create a figure instance. Then we can add axes to that figure:

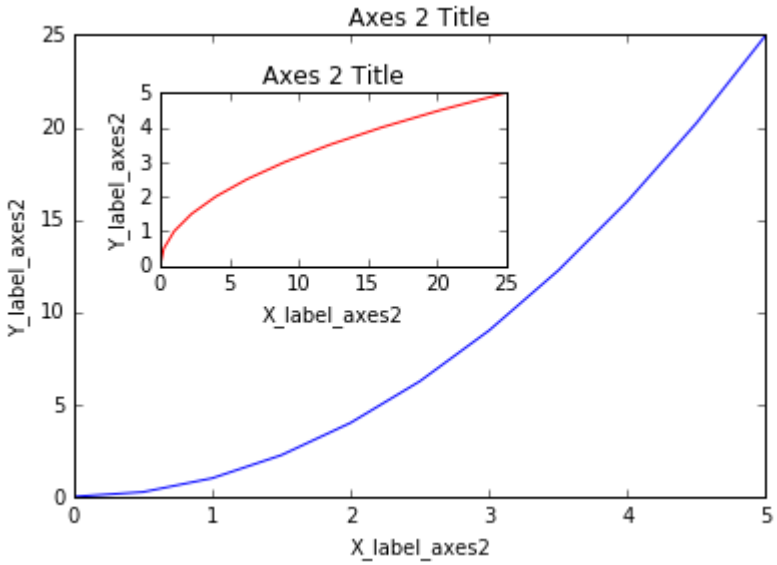
In [15]:
```
# Create Figure (empty canvas)
fig = plt.figure()

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

# 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[15]:
```
<matplotlib.text.Text at 0x111c85198>
```



Code is a little more complicated, but the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

In [16]:
```
# 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');
```

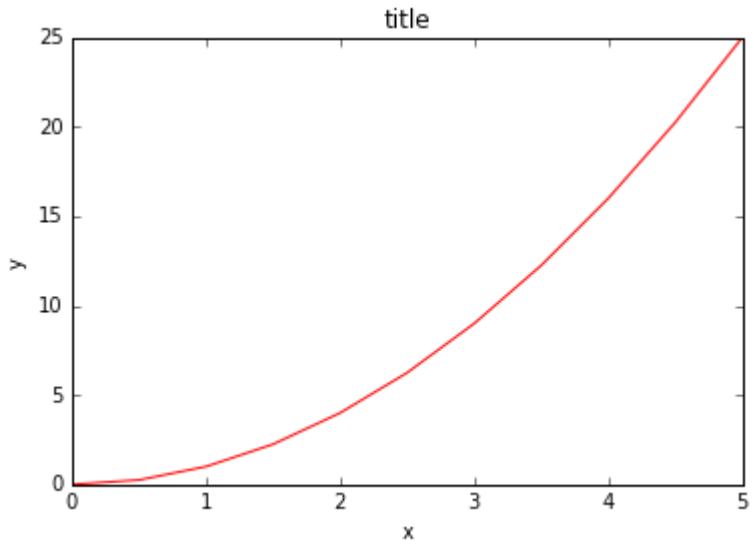


subplots()

The plt.subplots() object will act as a more automatic axis manager.

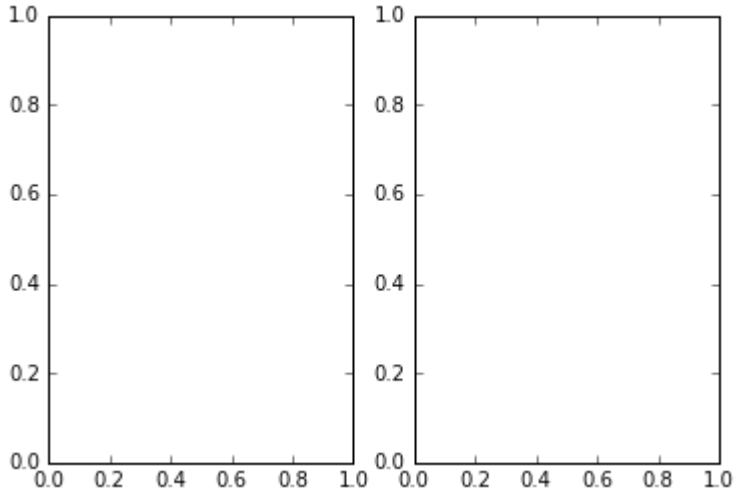
Basic use cases:

In [18]:
```
# 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');
```



Then you can specify the number of rows and columns when creating the subplots() object:

In [24]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



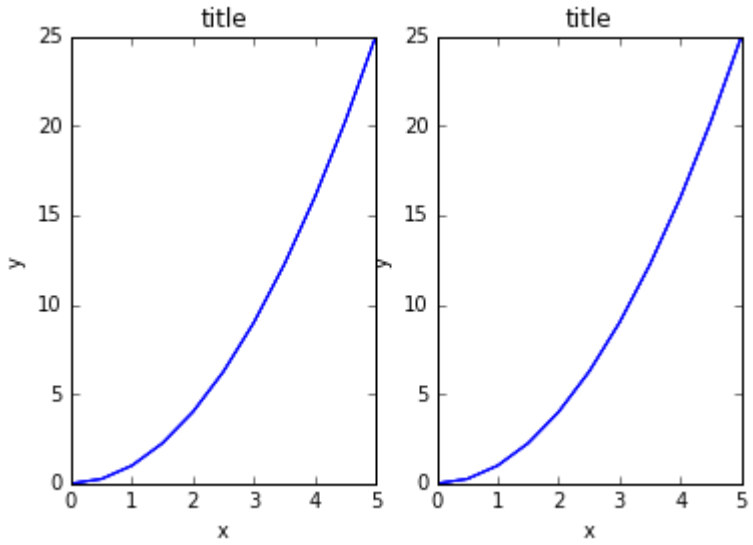
In [25]:
```
# Axes is an array of axes to plot on
axes
```

Out[25]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x111f0f8d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x1121f5588>], dtype=
object)
```

We can iterate through this array:

In [28]:
```
for ax in axes:
    ax.plot(x, y, 'b')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

# Display the figure object
fig
```

Out[28]:



A common issue with matplolib is overlapping subplots or figures. We ca use **fig.tight_layout()** or **plt.tight_layout()** method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

In [32]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'g')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig
plt.tight_layout()
```

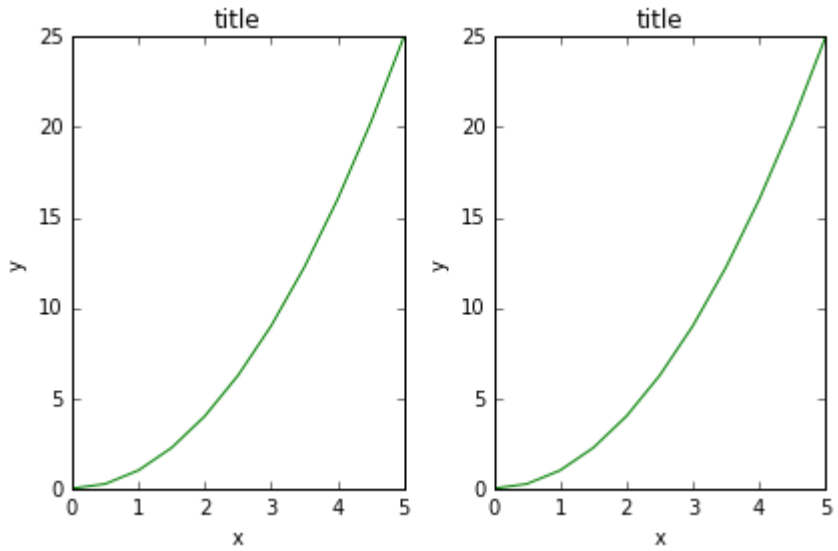


Figure size, aspect ratio and DPI

Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created. You can use the `figsize` and `dpi` keyword arguments.

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

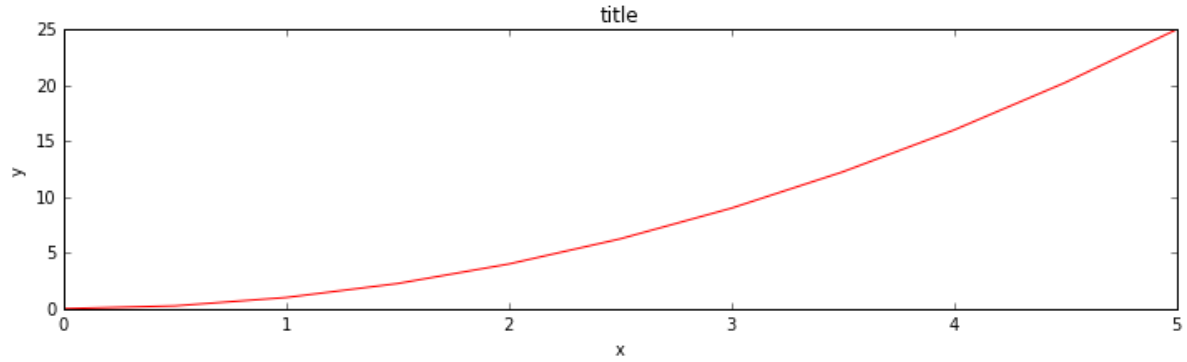
For example:

In [33]:
```
fig = plt.figure(figsize=(8,4), dpi=100)
```

```
<matplotlib.figure.Figure at 0x11228ea58>
```

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

In [34]:
```
fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('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 [68]:
```
fig.savefig("filename.png")
```

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

In [69]:
```
fig.savefig("filename.png", dpi=200)
```

Legends, labels and titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how decorate a figure with titles, axis labels, and legends.

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 [41]:
```
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:

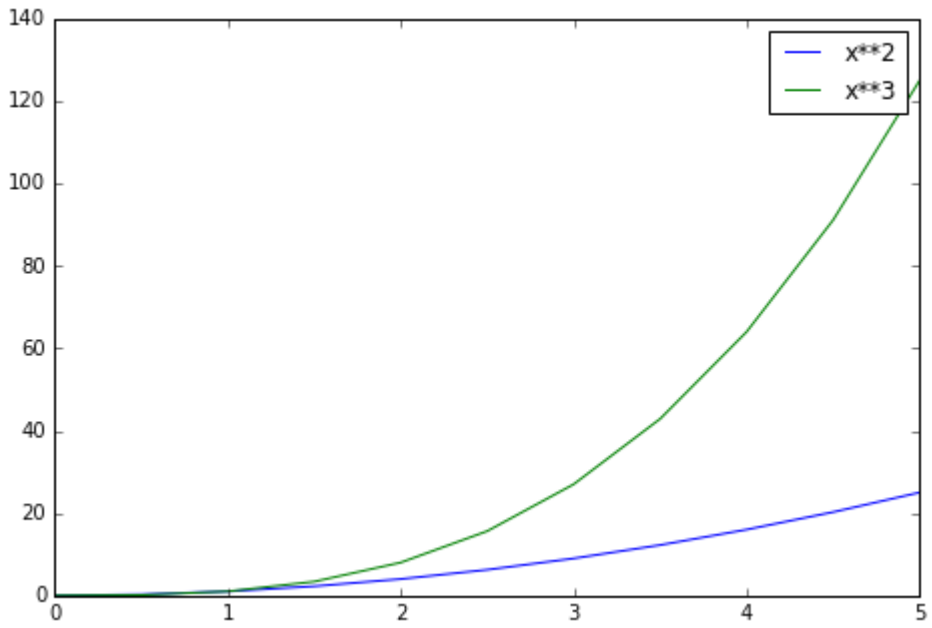
In [42]:
```
ax.set_xlabel("x")
ax.set_ylabel("y");
```

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 [48]:
```
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[48]:
```
<matplotlib.legend.Legend at 0x113a3d8d0>
```

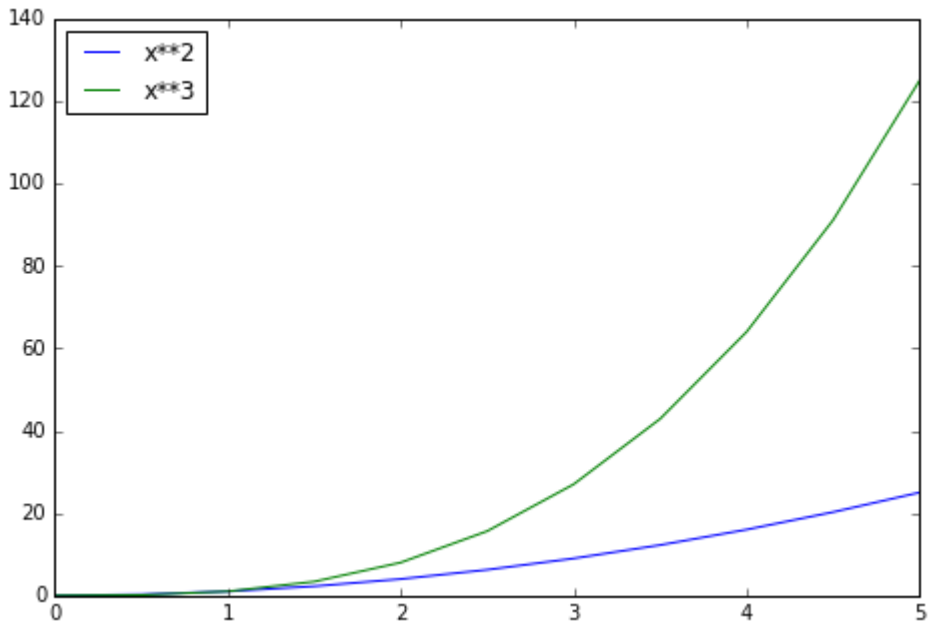


Notice how are legend overlaps some of the actual plot!

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 [52]:
```
# Lots of options....

ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner

# .. many more options are available

# Most common to choose
ax.legend(loc=0) # let matplotlib decide the optimal location
fig
```

Out[52]:



Setting colors, linewidths, linetypes

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

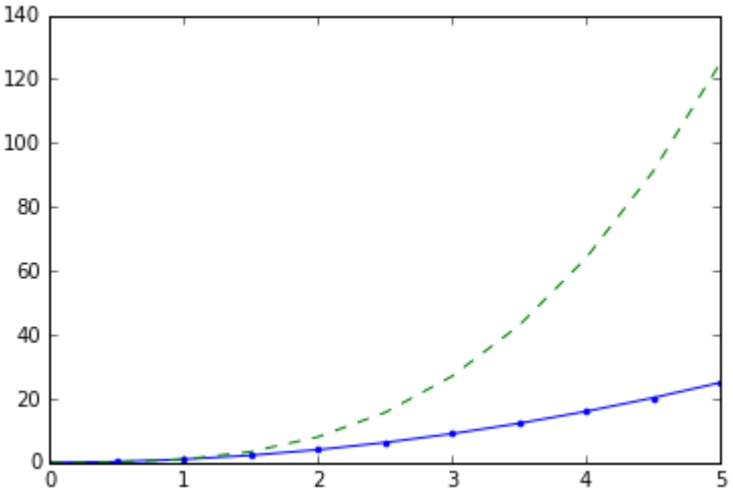
There is the basic MATLAB like syntax (which I would suggest you avoid using for more clairty sake:

Colors with MatLab like syntax

With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. 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 [54]:
```
# 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[54]:
```
[<matplotlib.lines.Line2D at 0x111fae048>]
```

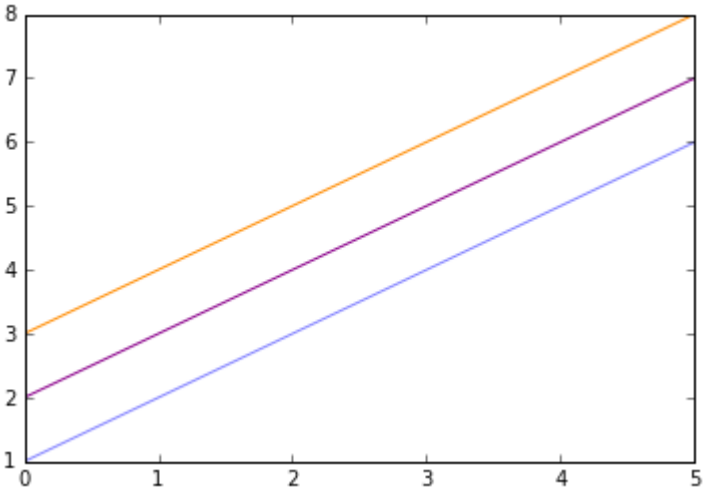


Colors with the color= parameter

We can also define colors by their names or RGB hex codes and optionally provide an alpha value using the `color` and `alpha` keyword arguments. Alpha indicates opacity.

In [56]:
```
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
```

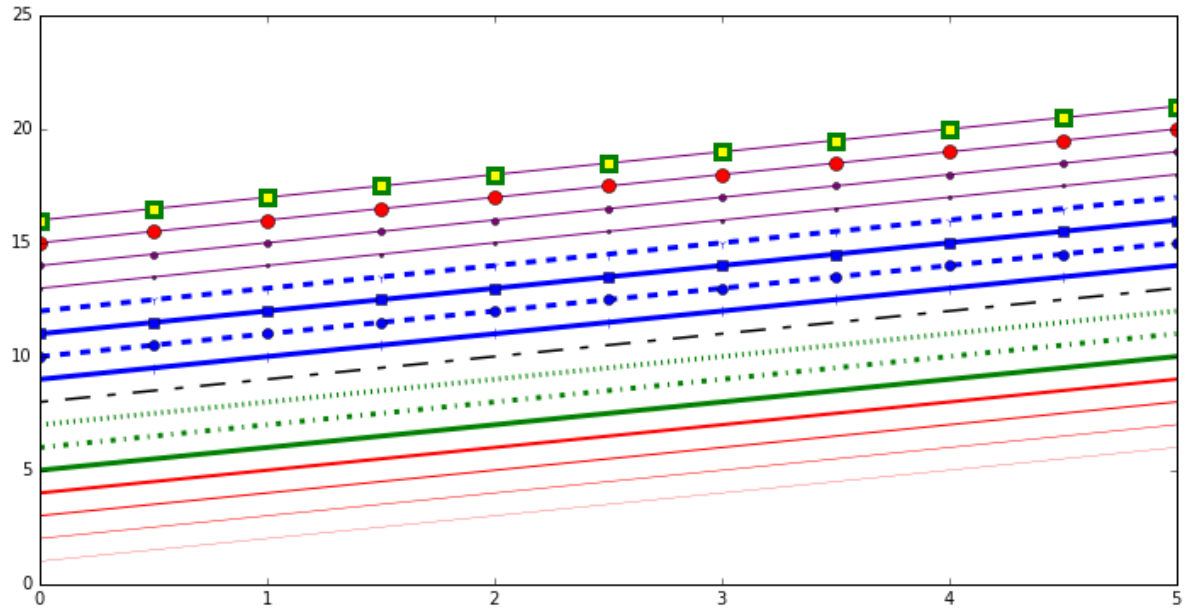
Out[56]:
```
[<matplotlib.lines.Line2D at 0x112179390>]
```



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 [57]:
```
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=0.50)
ax.plot(x, x+3, color="red", linewidth=1.00)
ax.plot(x, x+4, color="red", linewidth=2.00)

# 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=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3
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')
ax.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')

# 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, marke
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```



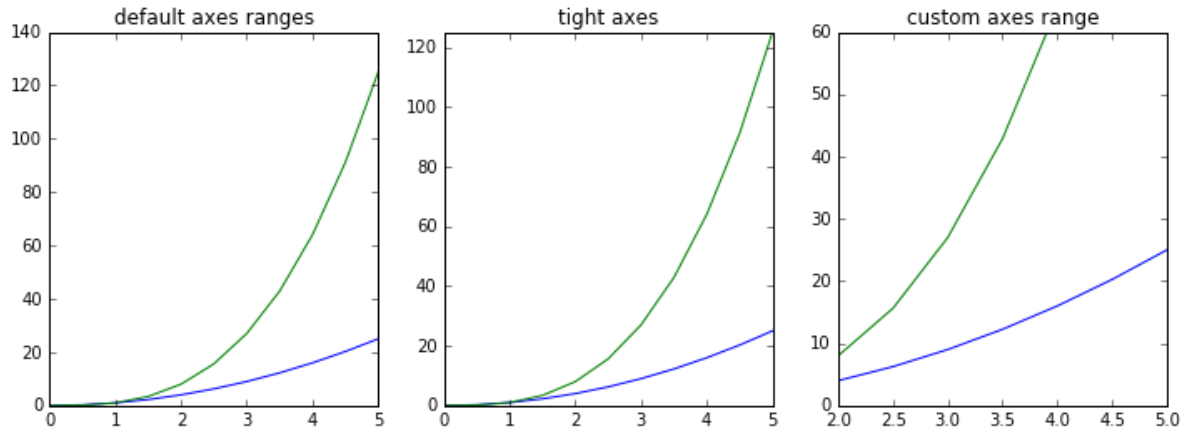
Control over axis appearance

In this section we will look at controlling axis sizing properties in a matplotlib figure.

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 [58]:
```
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");
```

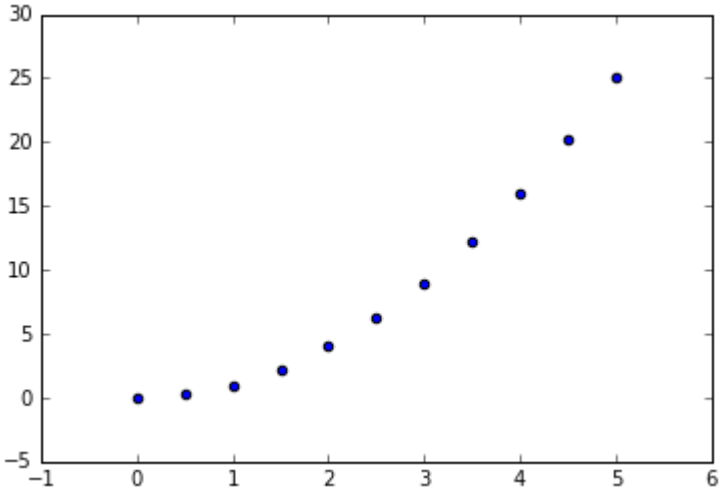


Special Plot Types

There are many specialized plots we can create, such as barplots, histograms, scatter plots, and much more. Most of these type of plots we will actually create using seaborn, a statistical plotting library for Python. But here are a few examples of these type of plots:

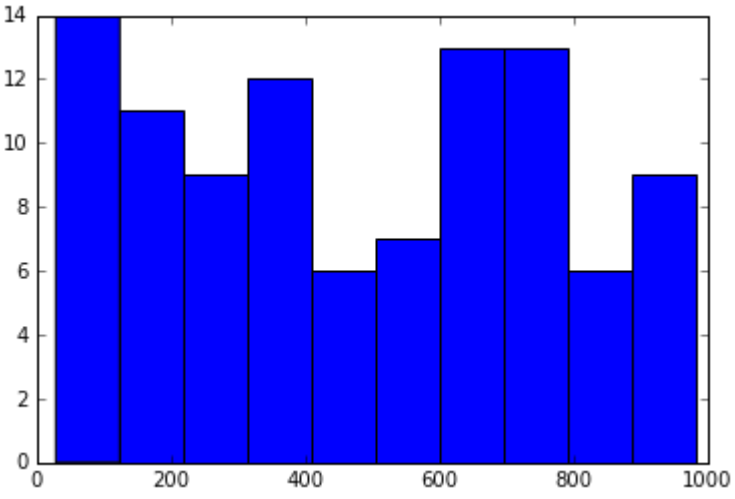
In [60]:
```
plt.scatter(x,y)
```

Out[60]:
```
<matplotlib.collections.PathCollection at 0x1122be438>
```

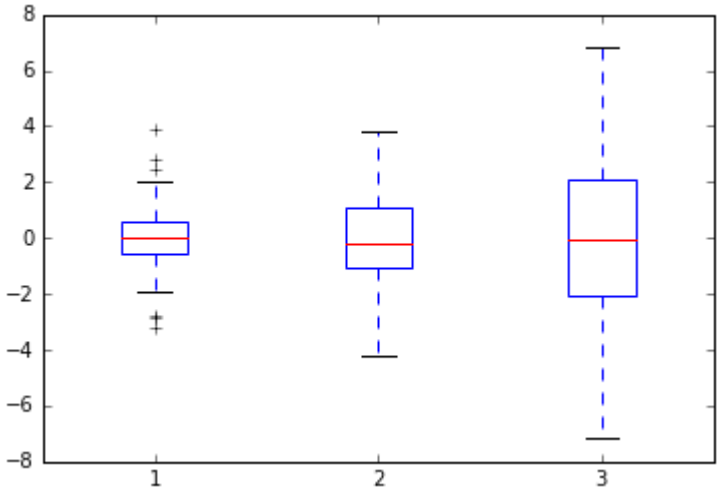


In [65]:
```
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

Out[65]:
```
(array([ 14.,  11.,   9.,  12.,   6.,   7.,  13.,  13.,   6.,   9.]),
 array([  28. ,  123.5,  219. ,  314.5,  410. ,  505.5,  601. ,  696.5,
         792. ,  887.5,  983. ]),
 <a list of 10 Patch objects>)
```



In [69]:
```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data,vert=True,patch_artist=True);
```



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.


(http://www.pieriandata.com)

Matplotlib Exercises - Solutions

Welcome to the exercises for reviewing matplotlib! Take your time with these, Matplotlib can be tricky to understand at first. These are relatively simple plots, but they can be hard if this is your first time with matplotlib, feel free to reference the solutions as you go along.

Also don't worry if you find the matplotlib syntax frustrating, we actually won't be using it that often throughout the course, we will switch to using seaborn and pandas built-in visualization capabilities. But, those are built-off of matplotlib, which is why it is still important to get exposure to it!

** * 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 [45]:
```
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. What command do you use if you aren't using the jupyter notebook?**

In [46]:
```
import matplotlib.pyplot as plt
%matplotlib inline
# plt.show() for non-notebook users
```

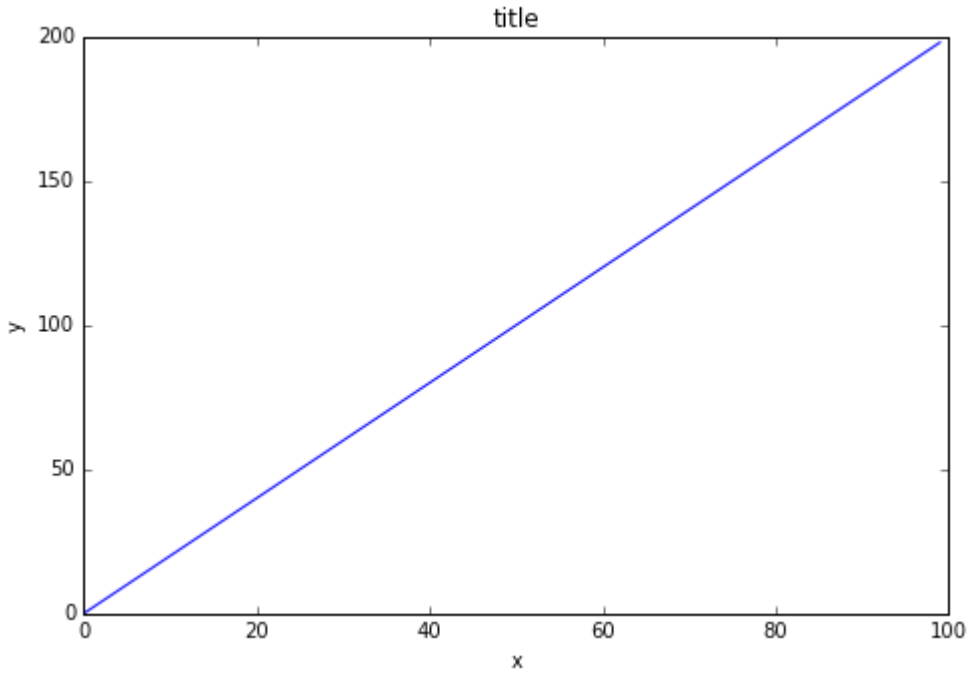
Exercise 1

** 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. **

In [47]:
```
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')
```

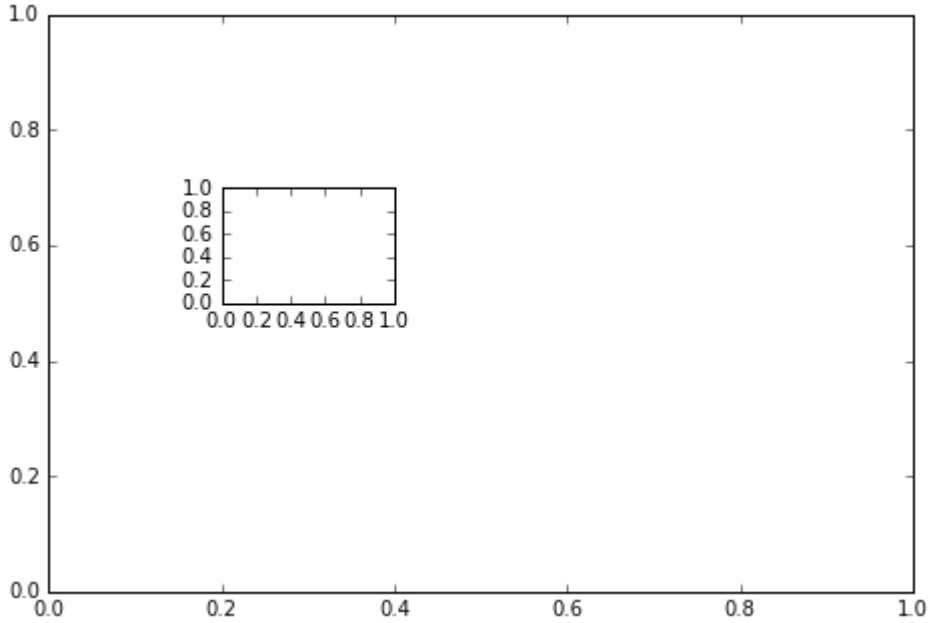
Out[47]:
```
<matplotlib.text.Text at 0x114ce6630>
```



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.**

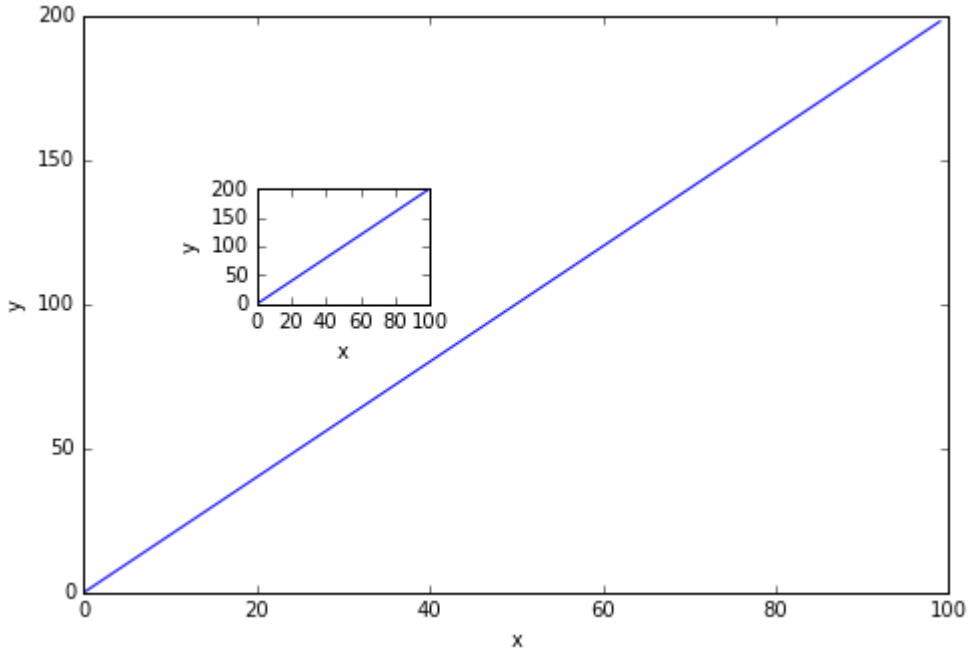
In [48]:
```
fig = plt.figure()

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 [49]:
```
ax1.plot(x,y)
ax1.set_xlabel('x')
ax1.set_ylabel('y')


ax2.plot(x,y)
ax2.set_xlabel('x')
ax2.set_ylabel('y')

fig # Show figure object
```

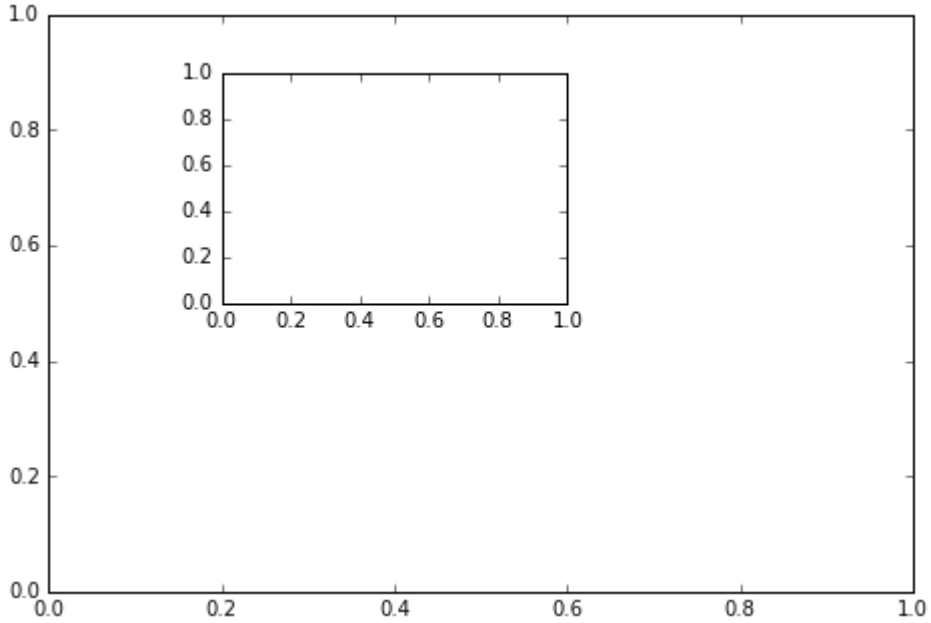
Out[49]:



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]**

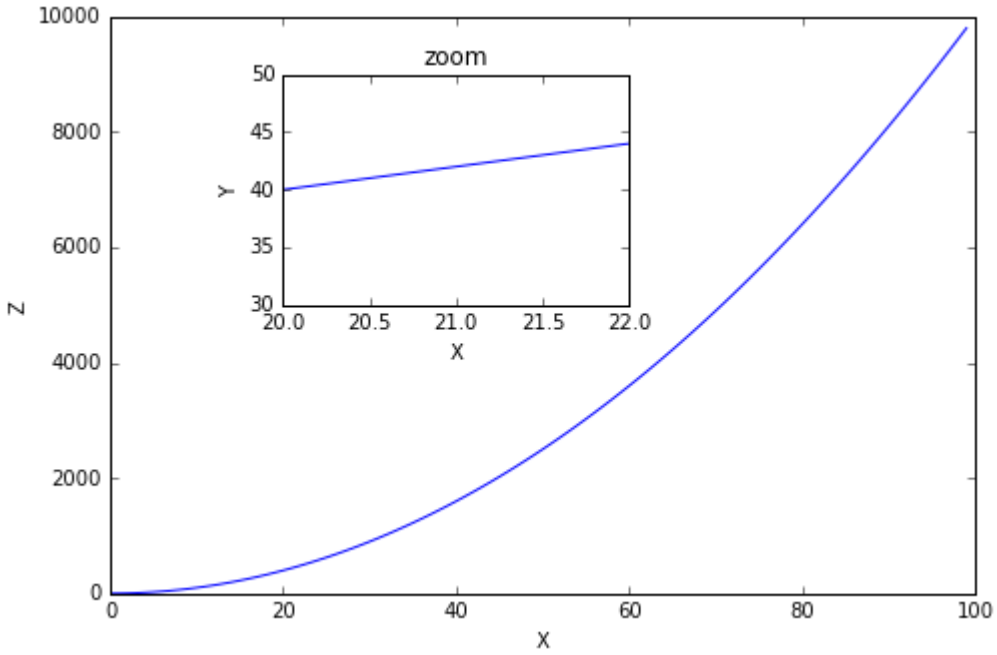
In [50]:
```
fig = plt.figure()

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 xlimits and y limits on the inserted plot:**

In [51]:
```
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
```

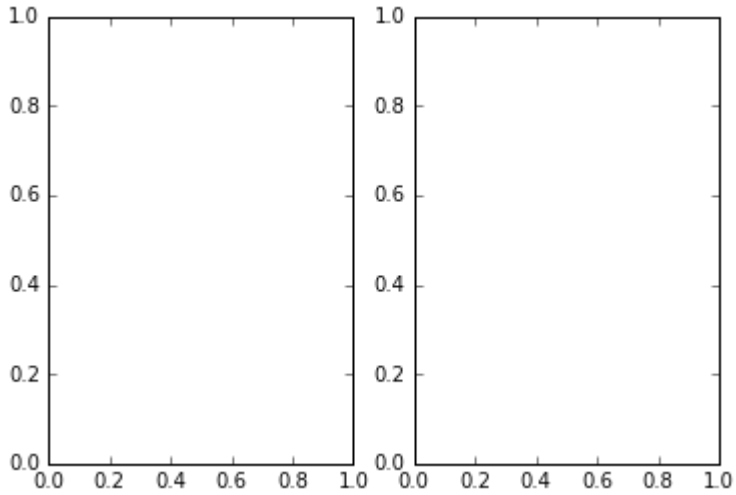
Out[51]:



Exercise 4

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

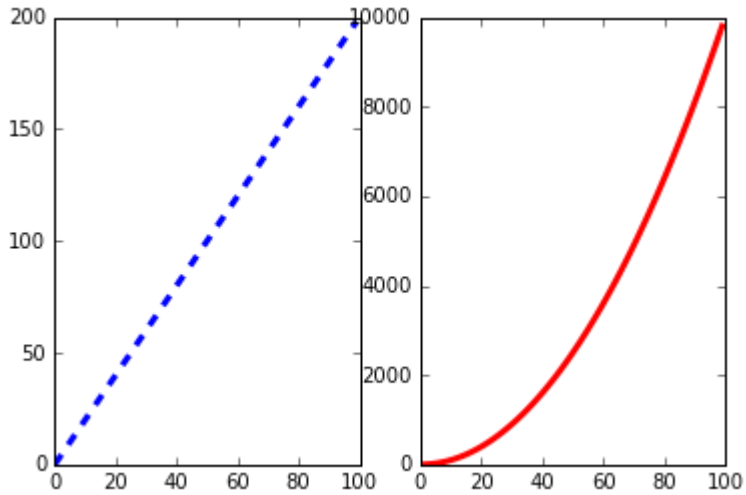
In [52]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



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

In [53]:
```
axes[0].plot(x,y,color="blue", lw=3, ls='--')
axes[1].plot(x,z,color="red", lw=3, ls='-')
fig
```

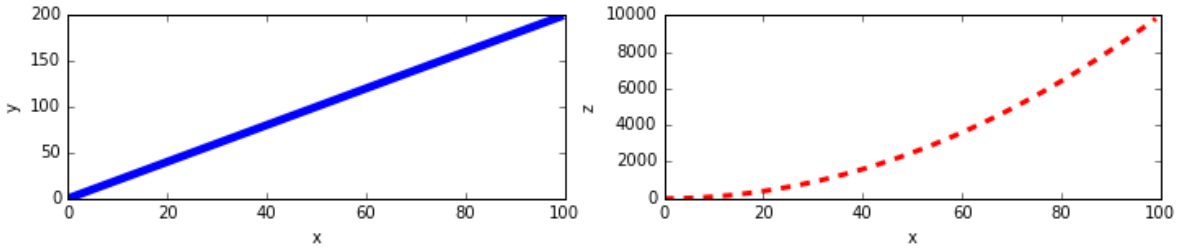
Out[53]:



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

In [54]:
```
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[54]: <matplotlib.text.Text at 0x115847da0>



Great Job!

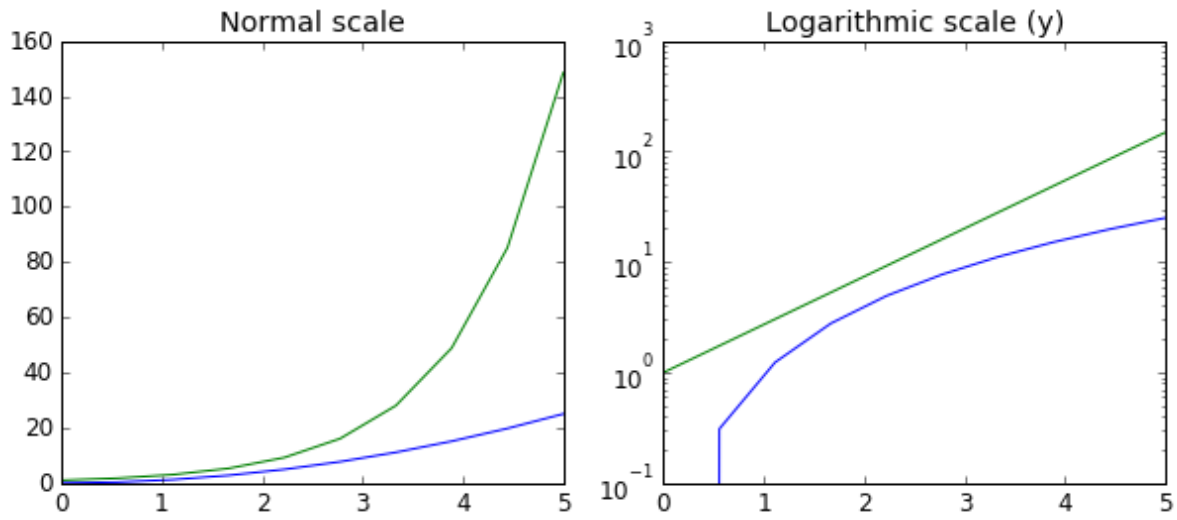
Advanced Matplotlib Concepts Lecture

In this lecture we cover some more advanced topics which you won't usually use as often. You can always reference the documentation for more resources!

Logarithmic scale

It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

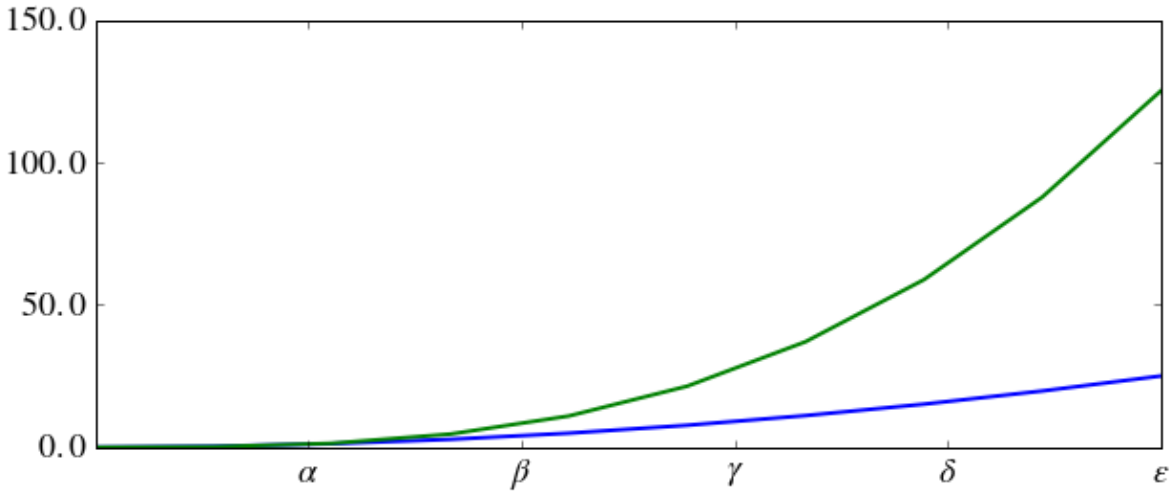
In [94]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

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 explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

In [95]:
```
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'$\eps

yticks = [0, 50, 100, 150]
ax.set_yticks(yticks)
ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use LaTeX f
```

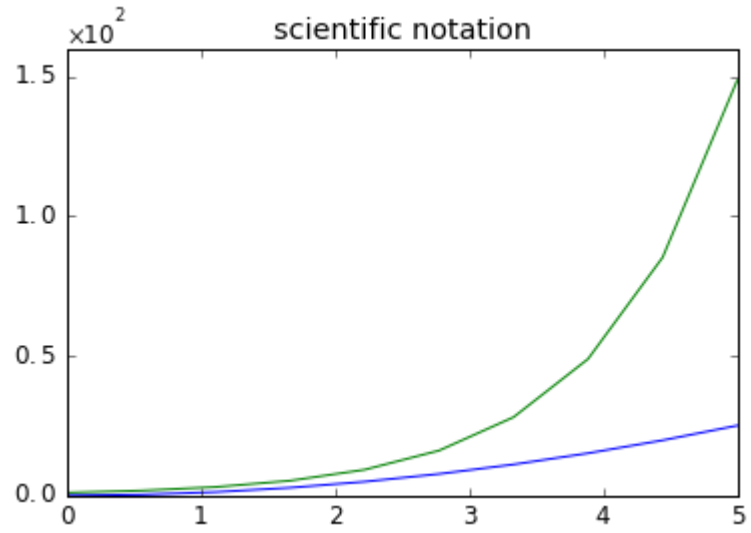


There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html (http://matplotlib.org/api/ticker_api.html) for details.

Scientific notation

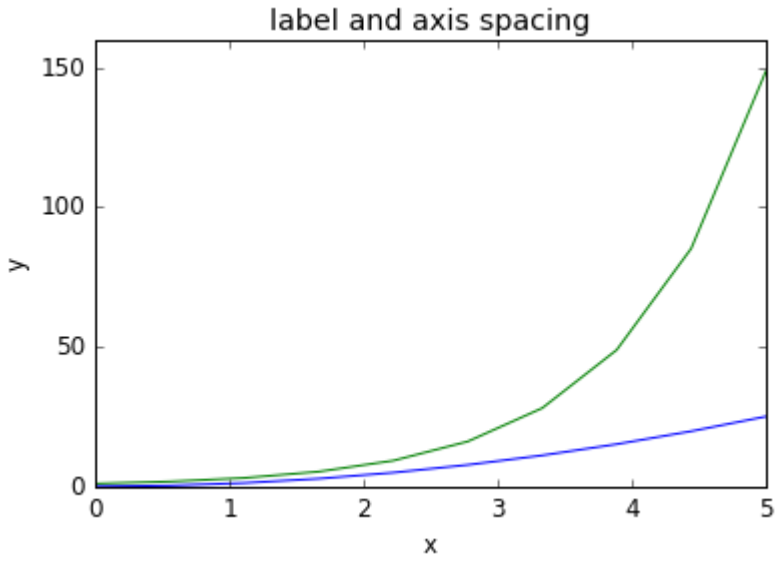
With large numbers on axes, it is often better use scientific notation:

In [96]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_title("scientific notation")

ax.set_yticks([0, 50, 100, 150])

from matplotlib import ticker
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.yaxis.set_major_formatter(formatter)
```



Axis number and axis label spacing

In [97]:
```
# distance between x and y axis and the numbers on the axes
matplotlib.rcParams['xtick.major.pad'] = 5
matplotlib.rcParams['ytick.major.pad'] = 5

fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

ax.set_xlabel("x")
ax.set_ylabel("y");
```

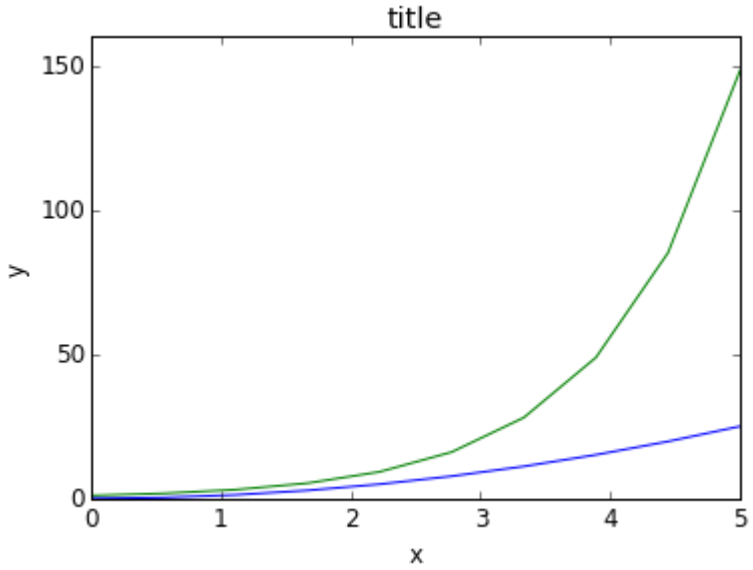


In [98]:
```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

Axis position adjustments

Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using `subplots_adjust` :

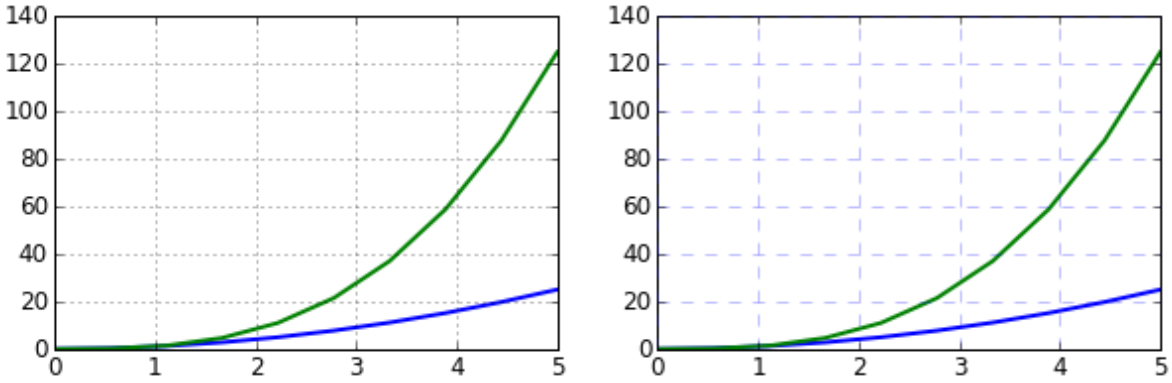
In [99]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("title")
ax.set_xlabel("x")
ax.set_ylabel("y")

fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```



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:

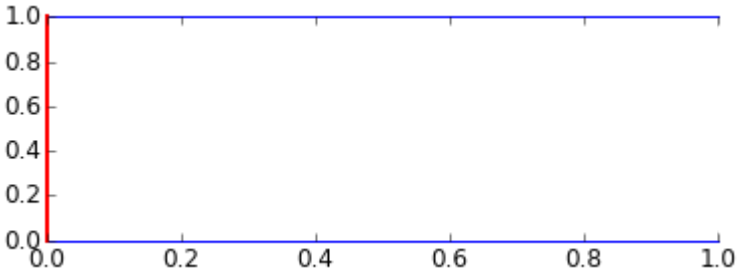
In [100]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# 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=0.5, linestyle='dashed', linewidth=0.5)
```



Axis spines

We can also change the properties of axis spines:

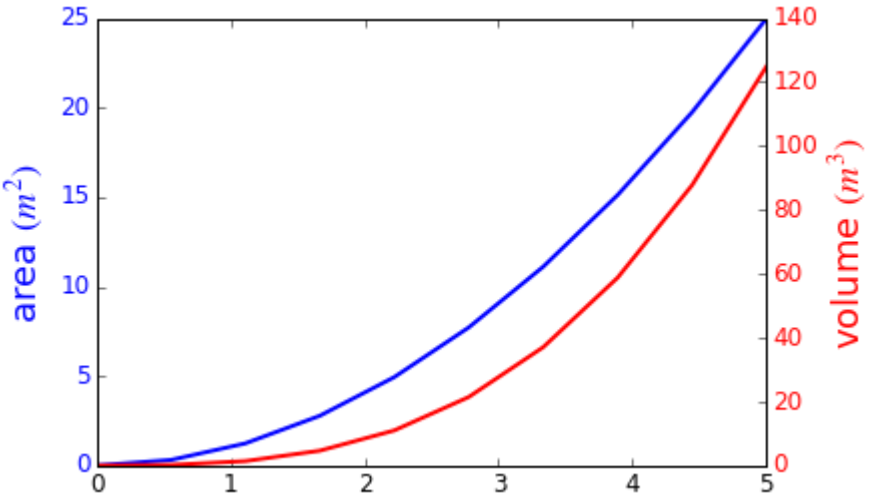
In [101]:
```
fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```



Twin axes

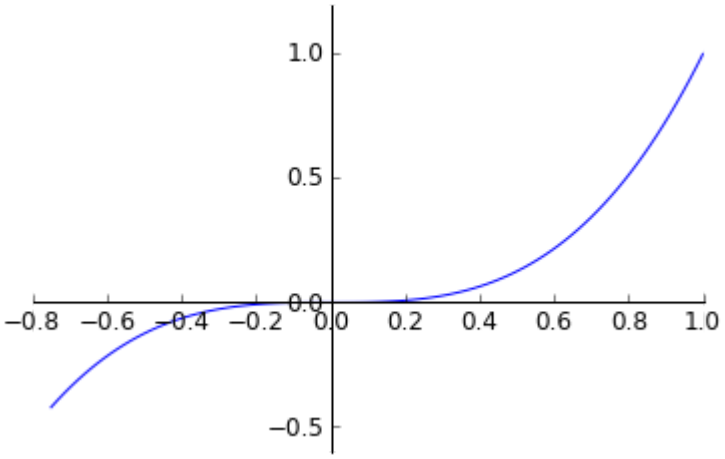
Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves

In [102]:
```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```



Axes where x and y is zero

In [103]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```

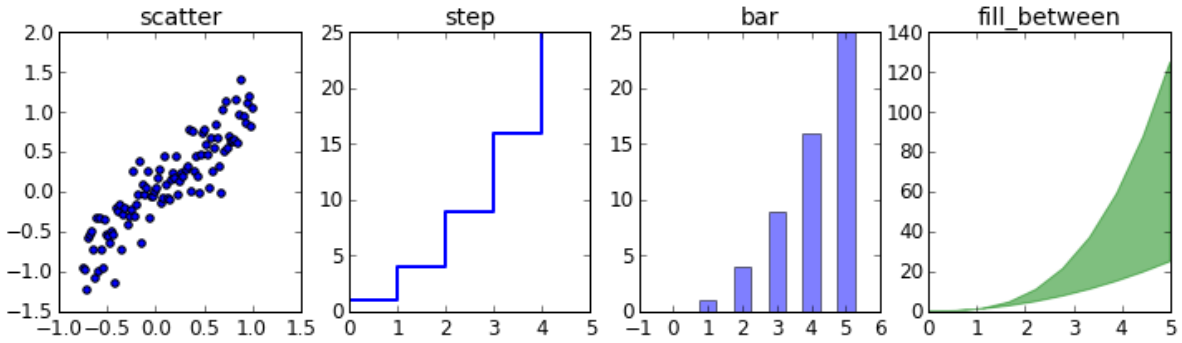


Other 2D plot styles

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 [104]:
```
n = np.array([0,1,2,3,4,5])
```

In [105]:
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*np.random.randn(len(xx)))
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");
```



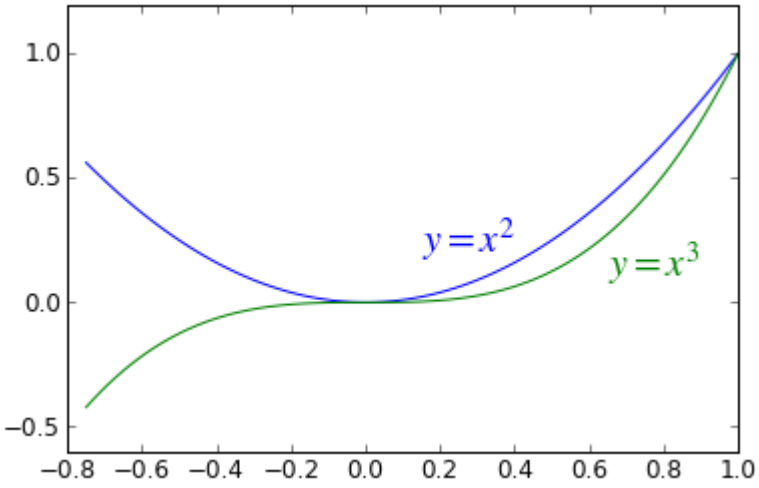
In [ ]:

In [ ]:

Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

In [108]:
```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

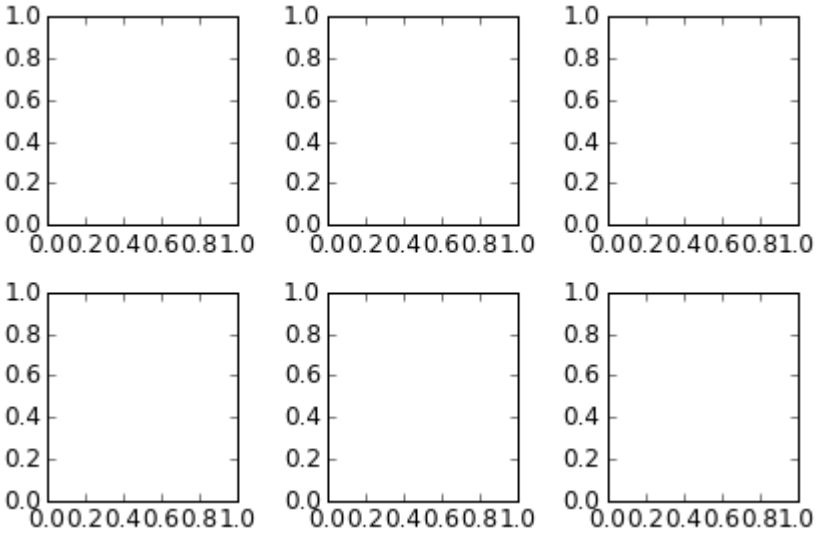


Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using `fig.add_axes` or using a sub-figure layout manager such as `subplots`, `subplot2grid`, or `gridspec`:

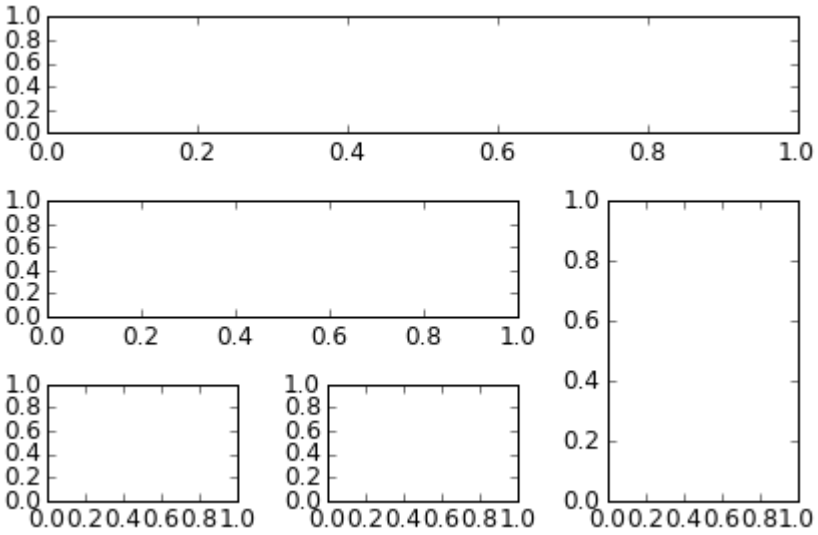
subplots

In [109]:
```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



subplot2grid

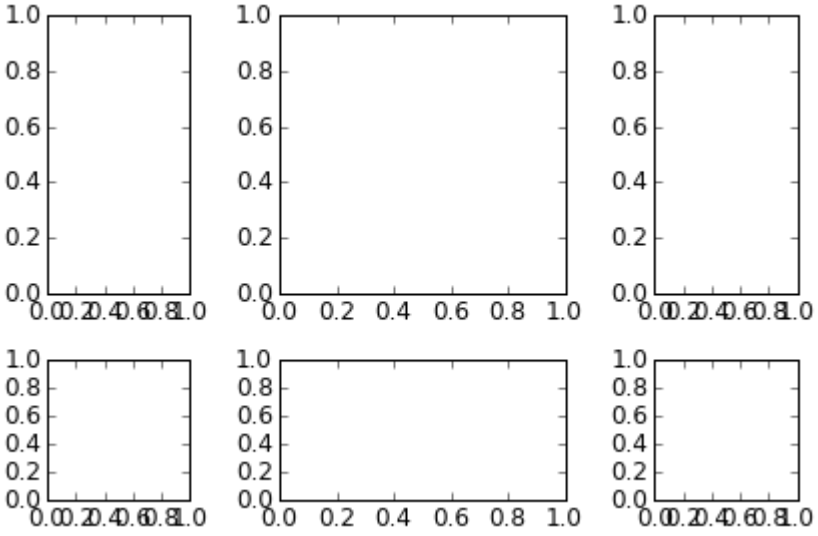
In [110]:
```
fig = plt.figure()
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
fig.tight_layout()
```



gridspec

In [111]:
```
import matplotlib.gridspec as gridspec
```

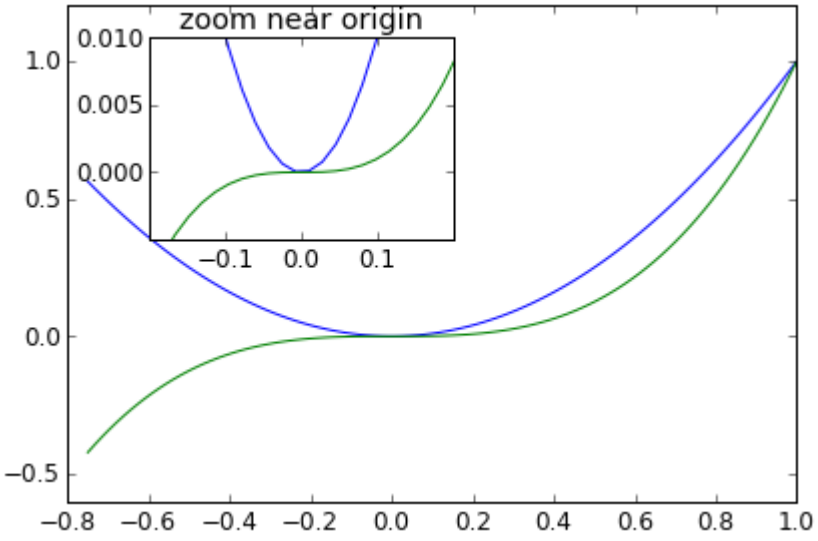
In [112]:
```
fig = plt.figure()

gs = gridspec.GridSpec(2, 3, height_ratios=[2,1], width_ratios=[1,2,1])
for g in gs:
    ax = fig.add_subplot(g)

fig.tight_layout()
```



add_axes

Manually adding axes with `add_axes` is useful for adding insets to figures:

In [113]:
```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)
fig.tight_layout()

# inset
inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

inset_ax.plot(xx, xx**2, xx, xx**3)
inset_ax.set_title('zoom near origin')

# set axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set axis tick locations
inset_ax.set_yticks([0, 0.005, 0.01])
inset_ax.set_xticks([-0.1,0,.1]);
```



Colormap and contour figures

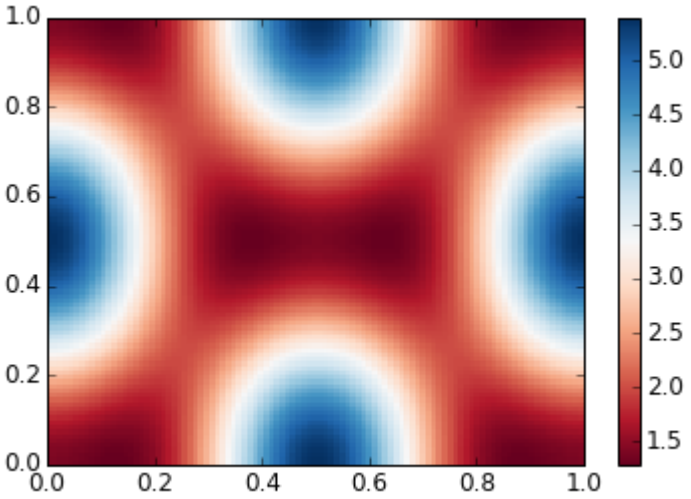
Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps,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

In [114]:
```
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_
```

In [115]:
```
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
```

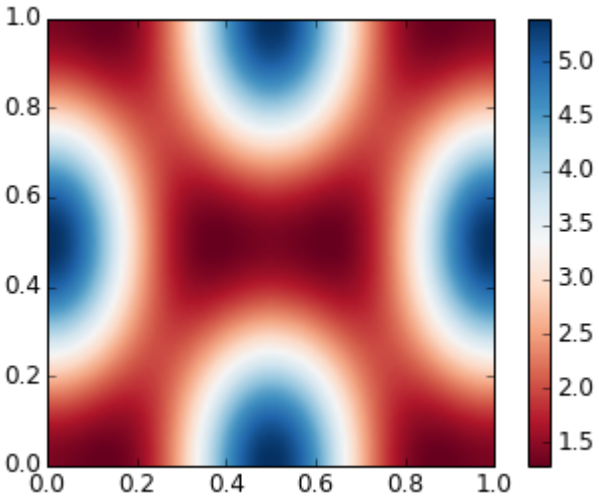
pcolor

In [116]:
```
fig, ax = plt.subplots()

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



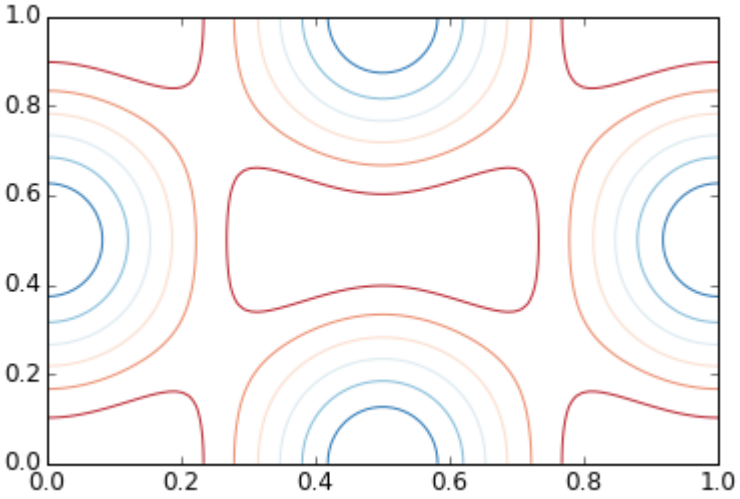
imshow

In [117]:
```
fig, ax = plt.subplots()

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

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



contour

In [118]:
```
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).ma
```



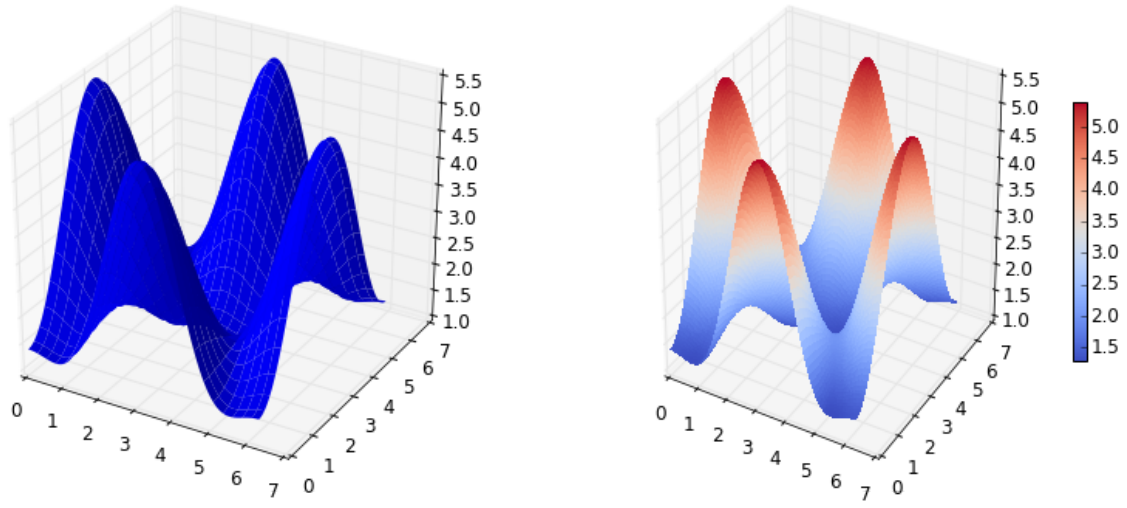
3D figures

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 [119]:
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

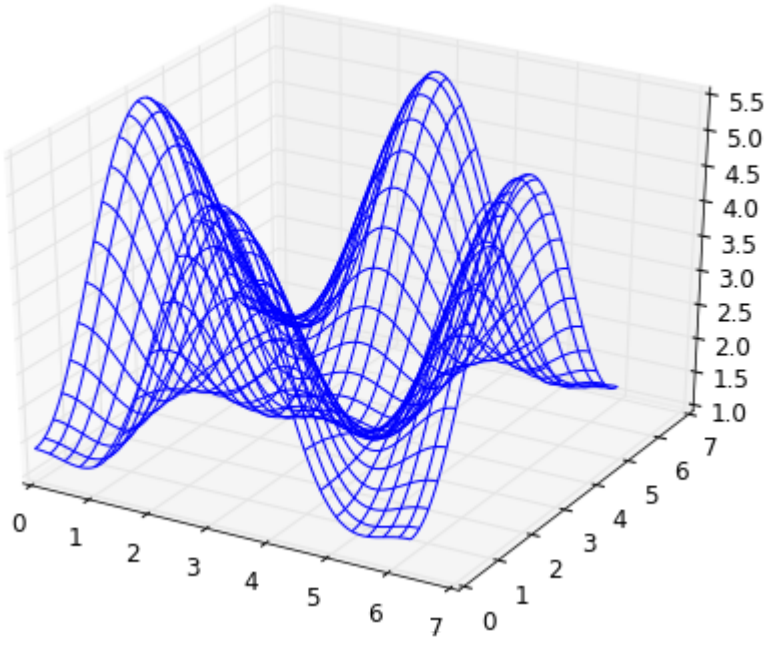
Surface plots

In [121]:
```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argu
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
cb = fig.colorbar(p, shrink=0.5)
```



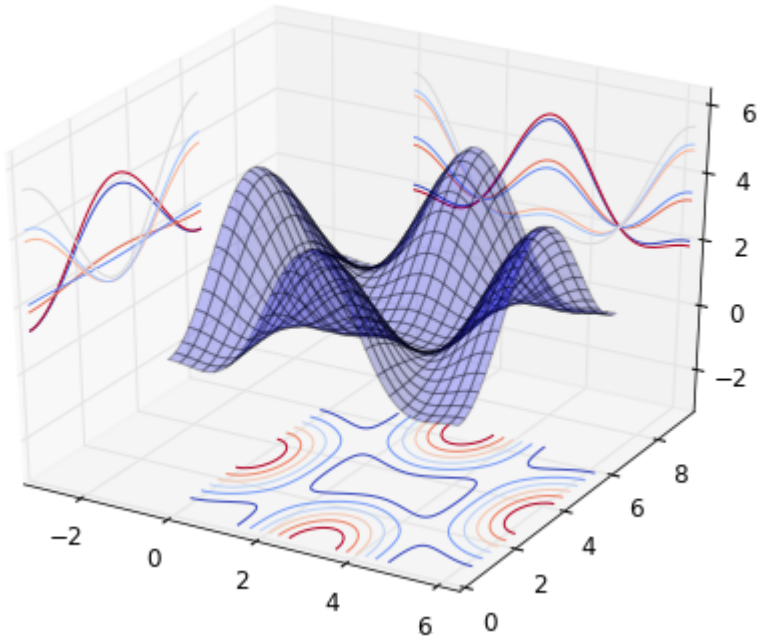
Wire-frame plot

In [122]:
```
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 [123]:
```
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.coolwar
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwar
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwa

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



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.


(http://www.pieriandata.com)

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 [4]:
```
tips.head()
```

Out[4]:

	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

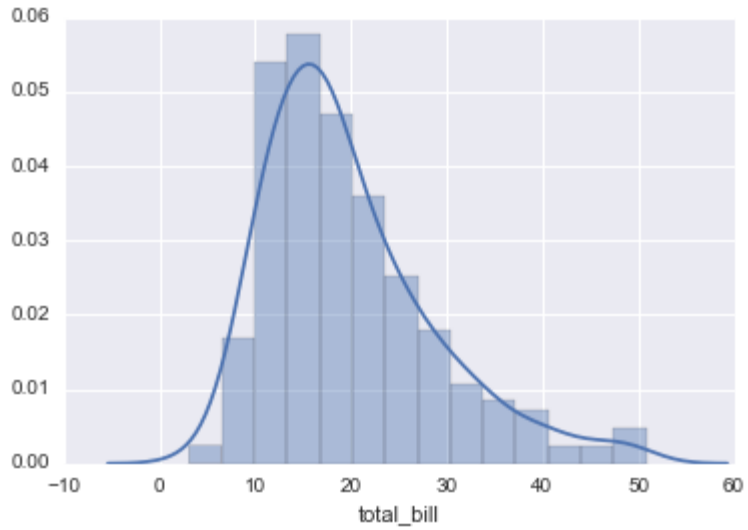
distplot

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

In [16]:
```
sns.distplot(tips['total_bill'])
# Safe to ignore warnings
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

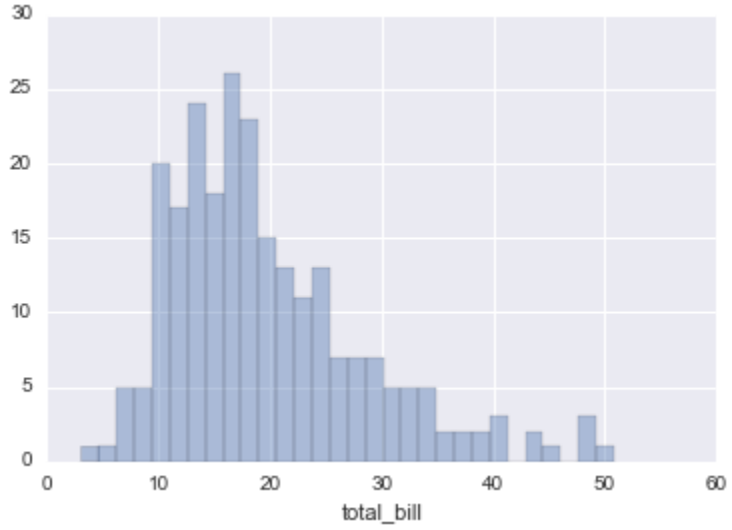
Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11dd8e5f8>
```



To remove the kde layer and just have the histogram use:

In [9]:
```
sns.distplot(tips['total_bill'],kde=False,bins=30)
```

Out[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c7b8668>
```



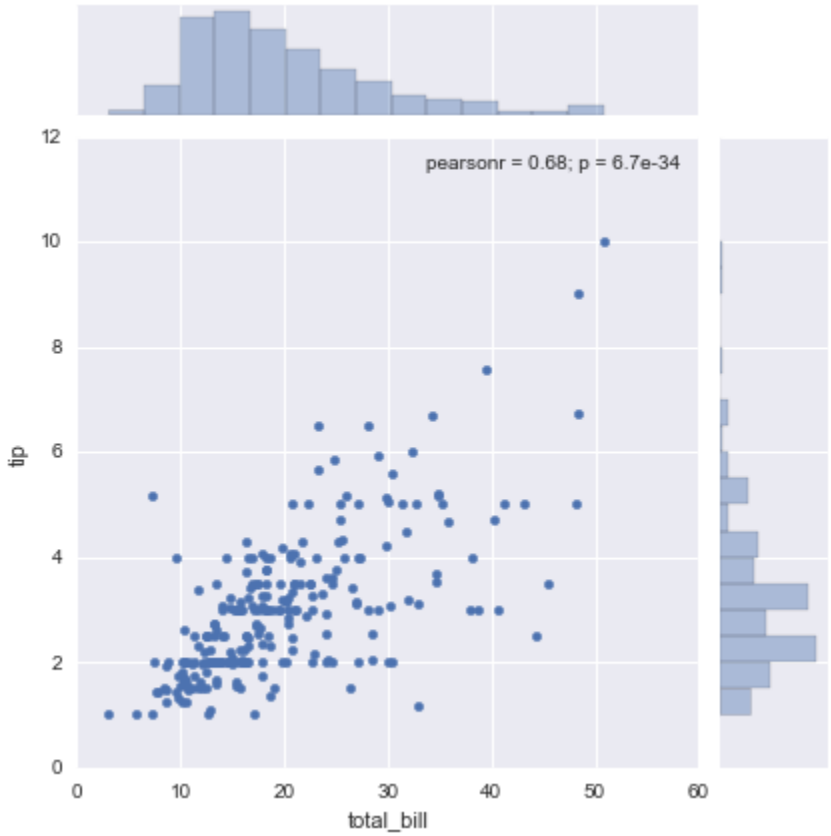
jointplot

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”

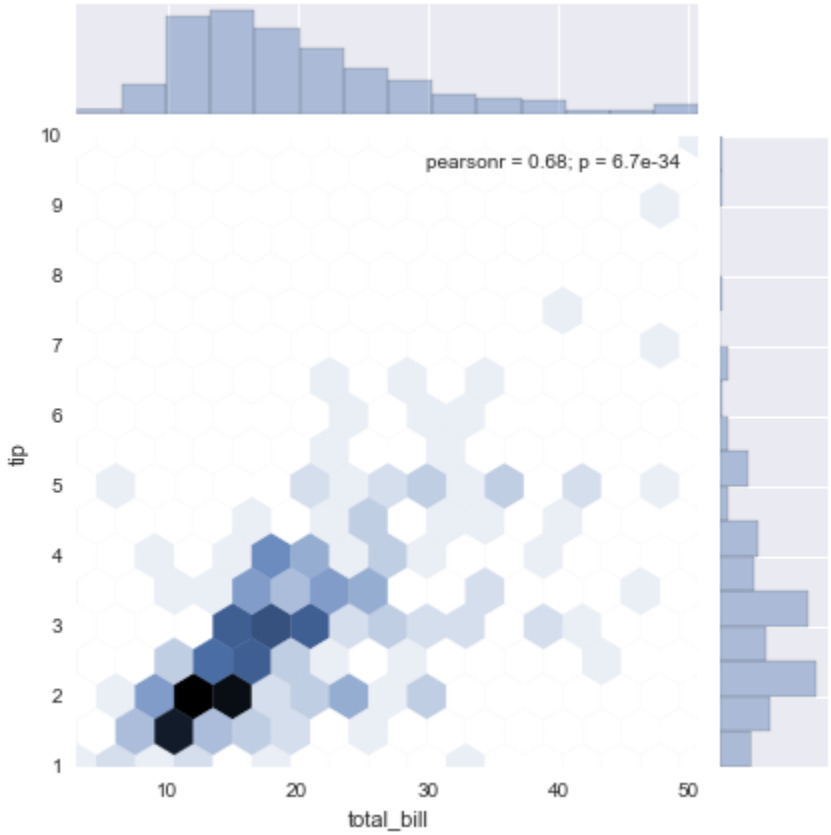
In [12]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='scatter')
```

Out[12]:
```
<seaborn.axisgrid.JointGrid at 0x11cfb28d0>
```



In [15]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='hex')
```

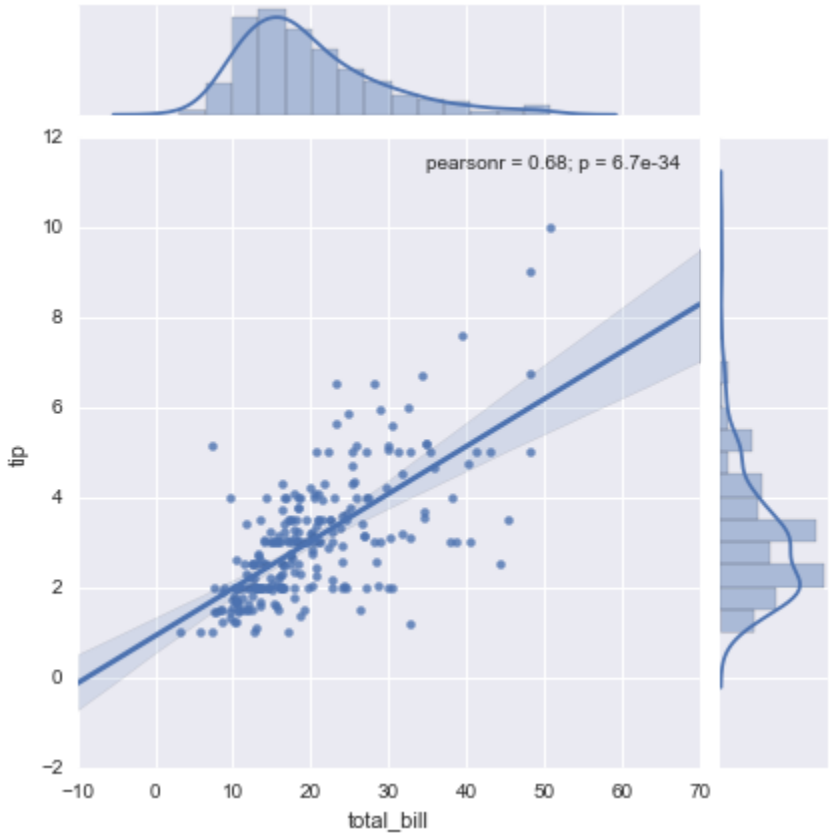
Out[15]:
```
<seaborn.axisgrid.JointGrid at 0x11d96f160>
```



In [17]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='reg')
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[17]: <seaborn.axisgrid.JointGrid at 0x11e0cfba8>

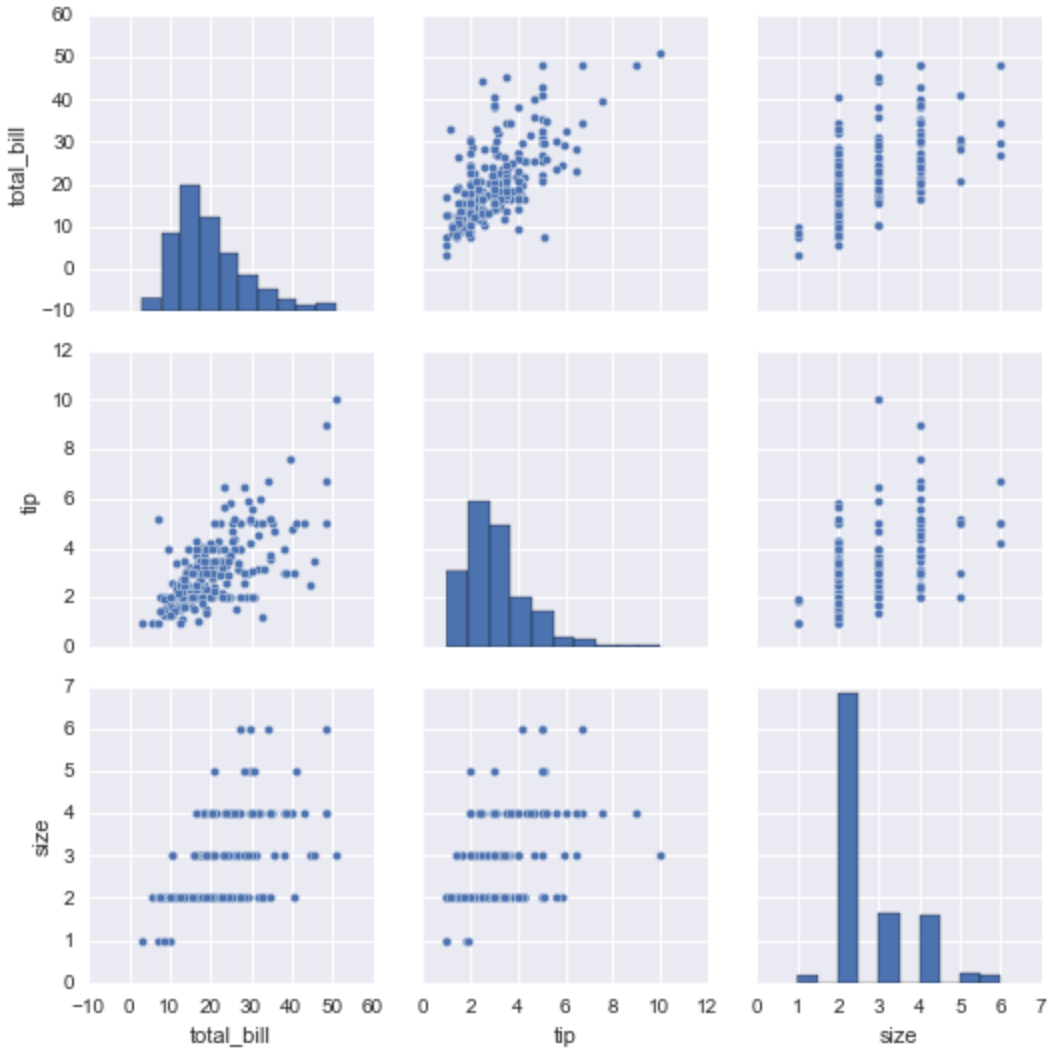


pairplot

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

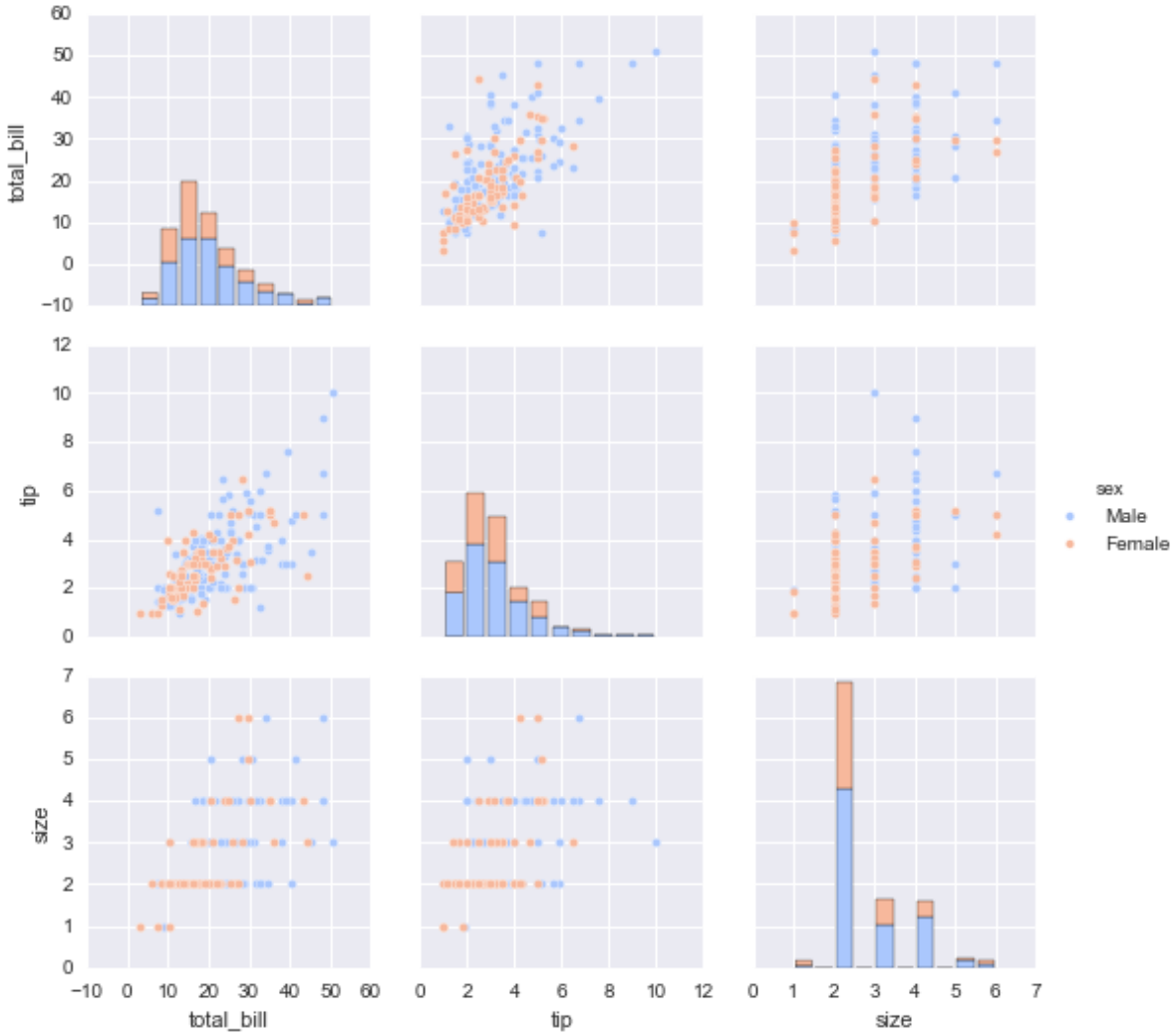
In [18]:
```
sns.pairplot(tips)
```

Out[18]:
```
<seaborn.axisgrid.PairGrid at 0x11e844208>
```



In [21]:
```
sns.pairplot(tips,hue='sex',palette='coolwarm')
```

Out[21]:
```
<seaborn.axisgrid.PairGrid at 0x11ff7a828>
```

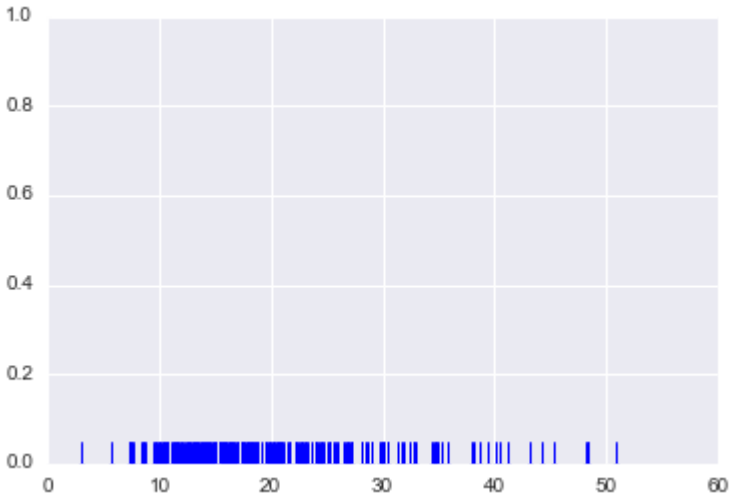


rugplot

rugplots are actually a very simple concept, they just draw a dash mark for every point on a univariate distribution. They are the building block of a KDE plot:

In [22]:
```
sns.rugplot(tips['total_bill'])
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1207c8b70>
```

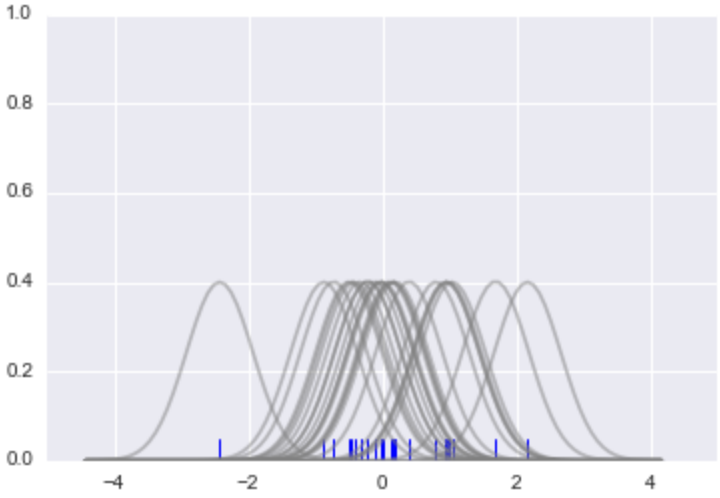


kdeplot

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. For example:

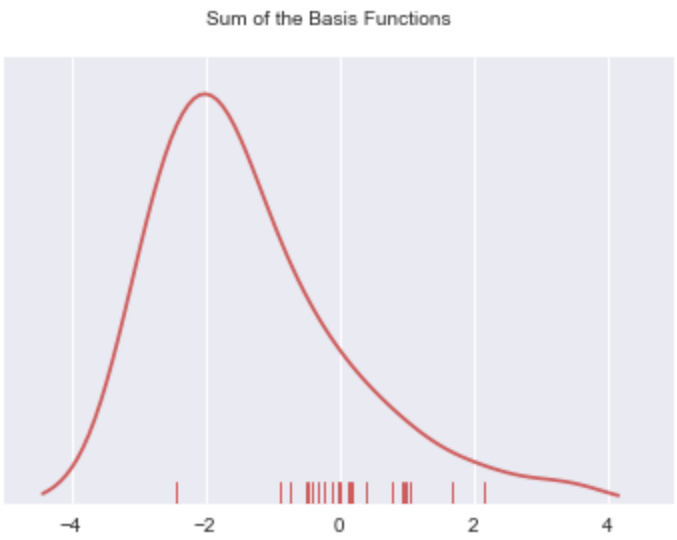
In [35]:
```
# Don't worry about understanding this code!
# It's just for the diagram below
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

#Create dataset
dataset = np.random.randn(25)

# Create another rugplot
sns.rugplot(dataset);

# Set up the x-axis for the plot
x_min = dataset.min() - 2
x_max = dataset.max() + 2

# 100 equally spaced points from x_min to x_max
x_axis = np.linspace(x_min,x_max,100)

# Set up the bandwidth, for info on this:
url = 'http://en.wikipedia.org/wiki/Kernel_density_estimation#Practical_estimat

bandwidth = ((4*dataset.std()**5)/(3*len(dataset)))**.2


# Create an empty kernel list
kernel_list = []

# Plot each basis function
for data_point in dataset:

    # Create a kernel for each point and append to list
    kernel = stats.norm(data_point,bandwidth).pdf(x_axis)
    kernel_list.append(kernel)

    #Scale for plotting
    kernel = kernel / kernel.max()
    kernel = kernel * .4
    plt.plot(x_axis,kernel,color = 'grey',alpha=0.5)

plt.ylim(0,1)
```

Out[35]:
```
(0, 1)
```



In [37]:
```
# To get the kde plot we can sum these basis functions.

# Plot the sum of the basis function
sum_of_kde = np.sum(kernel_list,axis=0)

# Plot figure
fig = plt.plot(x_axis,sum_of_kde,color='indianred')

# Add the initial rugplot
sns.rugplot(dataset,c = 'indianred')

# Get rid of y-tick marks
plt.yticks([])

# Set title
plt.suptitle("Sum of the Basis Functions")
```

Out[37]: <matplotlib.text.Text at 0x121c41da0>

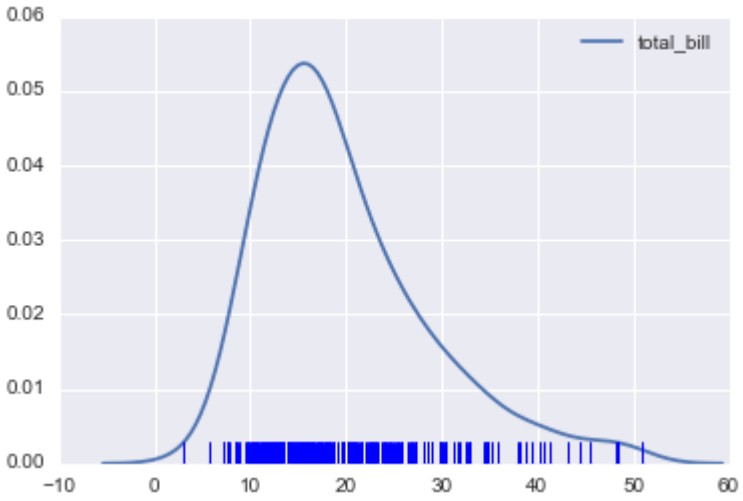


So with our tips dataset:

In [41]:
```
sns.kdeplot(tips['total_bill'])
sns.rugplot(tips['total_bill'])
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/k
detools.py:20: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

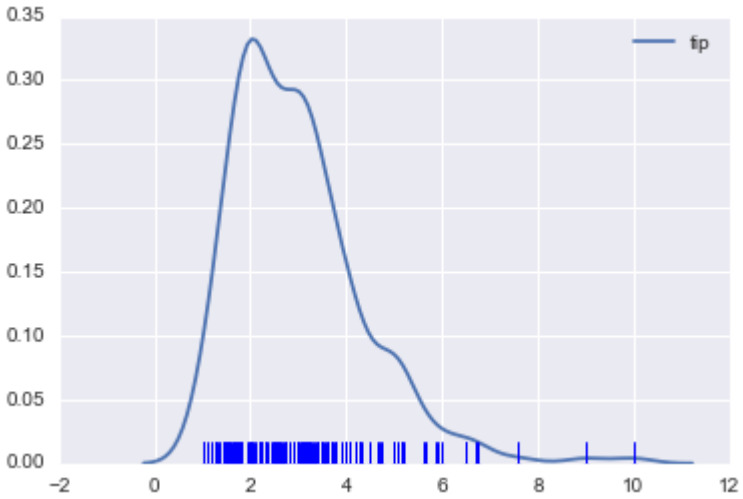
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x121b82c50>



In [42]:
```
sns.kdeplot(tips['tip'])
sns.rugplot(tips['tip'])
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/k
detools.py:20: VisibleDeprecationWarning: using a non-integer number instead
of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x12252cfd0>



Great Job!


(http://www.pieriandata.com)

Categorical Data Plots

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

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

Let's go through examples of each!

In [5]:
```
import seaborn as sns
%matplotlib inline
```

In [6]:
```
tips = sns.load_dataset('tips')
tips.head()
```

Out[6]:

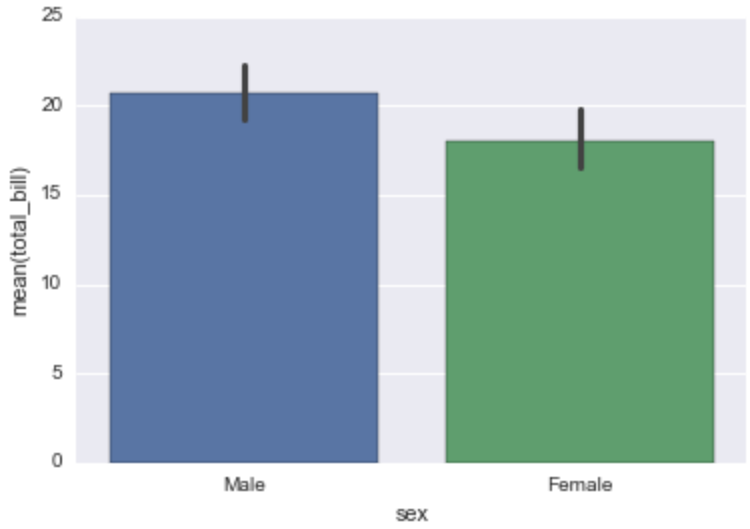
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

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 [8]:
```
sns.barplot(x='sex',y='total_bill',data=tips)
```

Out[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c99b8d0>
```

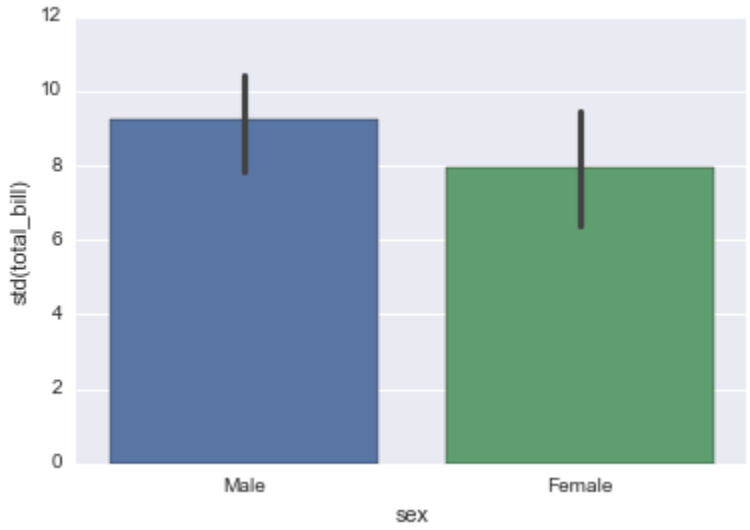


In [10]:
```
import numpy as np
```

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

In [11]:
```
sns.barplot(x='sex',y='total_bill',data=tips,estimator=np.std)
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c9b00b8>
```

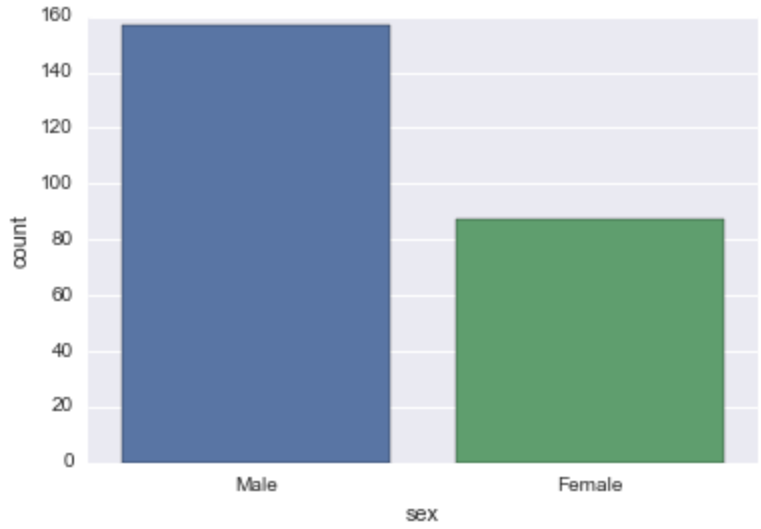


countplot

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 [13]:
```
sns.countplot(x='sex',data=tips)
```

Out[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1153276d8>
```

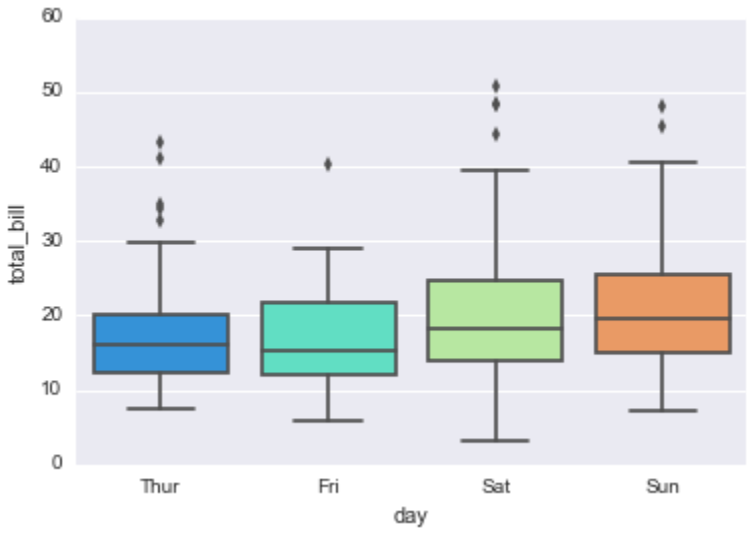


boxplot and violinplot

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.

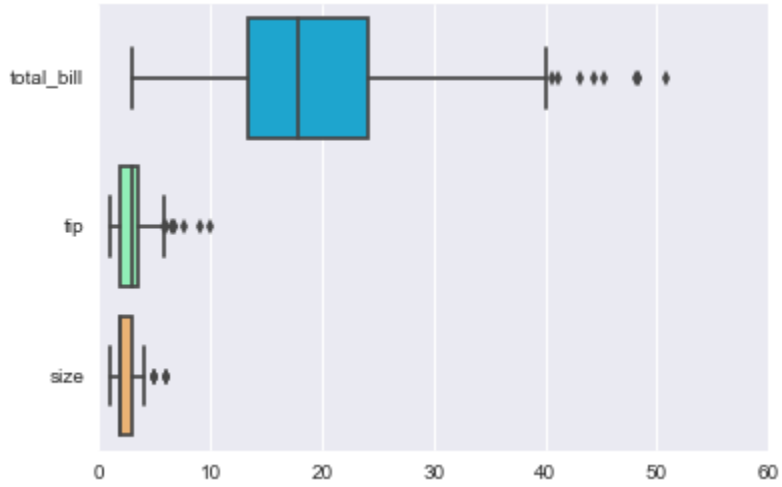
In [22]:
```
sns.boxplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11db81630>
```



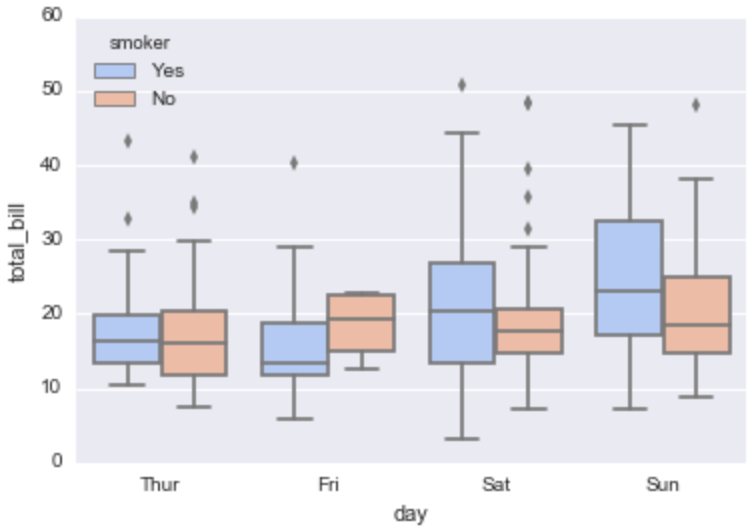
In [25]:
```
# Can do entire dataframe with orient='h'
sns.boxplot(data=tips,palette='rainbow',orient='h')
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x11e2c0b00>



In [26]:
```
sns.boxplot(x="day", y="total_bill", hue="smoker",data=tips, palette="coolwarm"
```

Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x11e2c77f0>

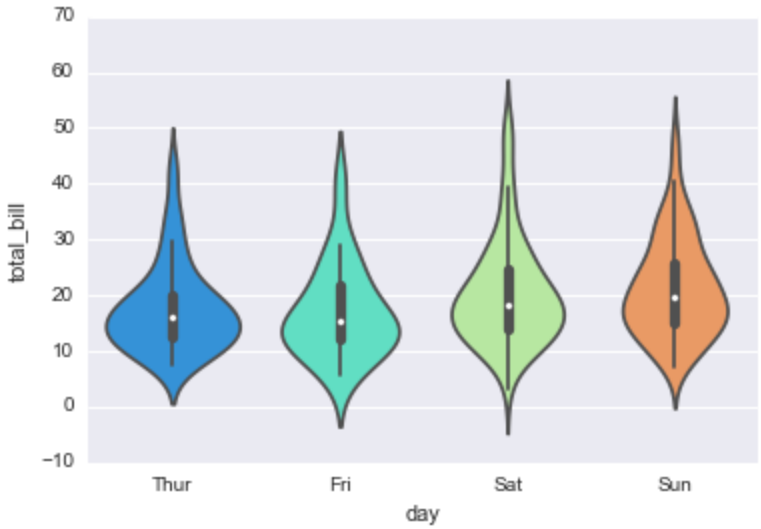


violinplot

A violin plot plays a similar role as a box and whisker plot. It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared. Unlike a box plot, in which all of the plot components correspond to actual datapoints, the violin plot features a kernel density estimation of the underlying distribution.

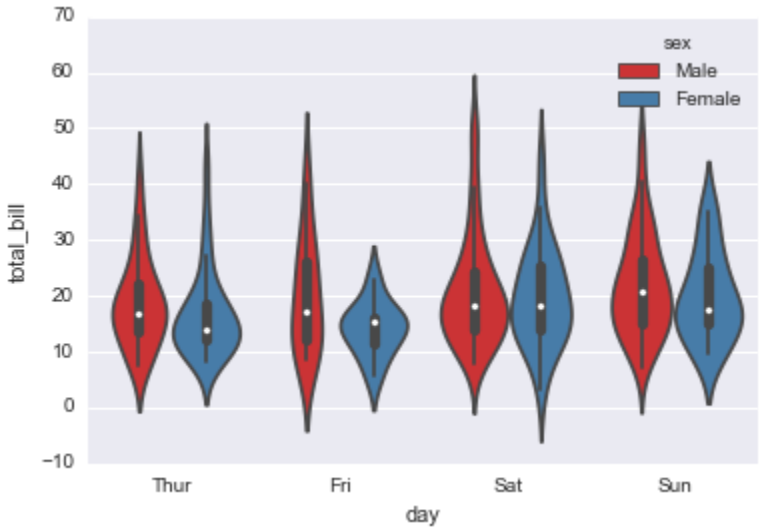
In [27]:
```
sns.violinplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e682ba8>
```



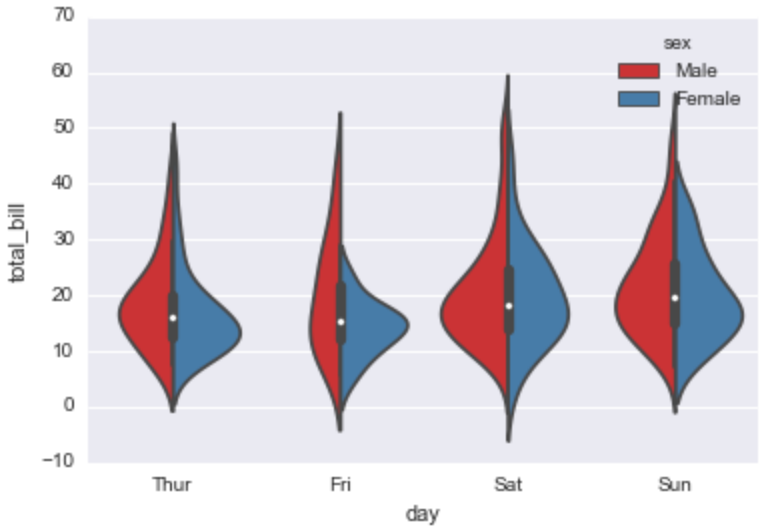
In [37]:
```
sns.violinplot(x="day", y="total_bill", data=tips,hue='sex',palette='Set1')
```

Out[37]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f739dd8>
```



In [36]:
```
sns.violinplot(x="day", y="total_bill", data=tips,hue='sex',split=True,palette=
```

Out[36]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f4d0710>
```



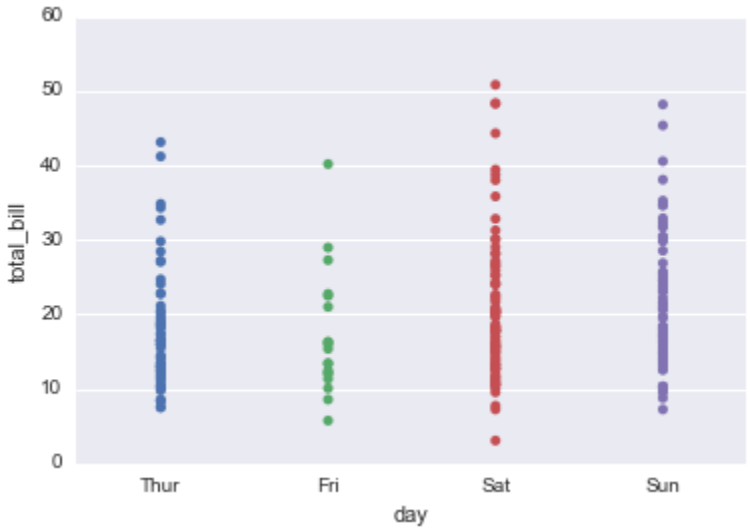
stripplot and swarmplot

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).

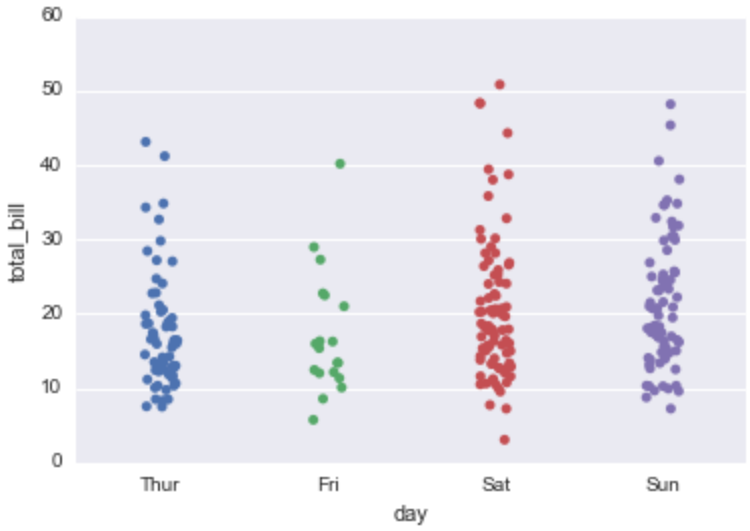
In [38]:
```
sns.stripplot(x="day", y="total_bill", data=tips)
```

Out[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x120272278>
```



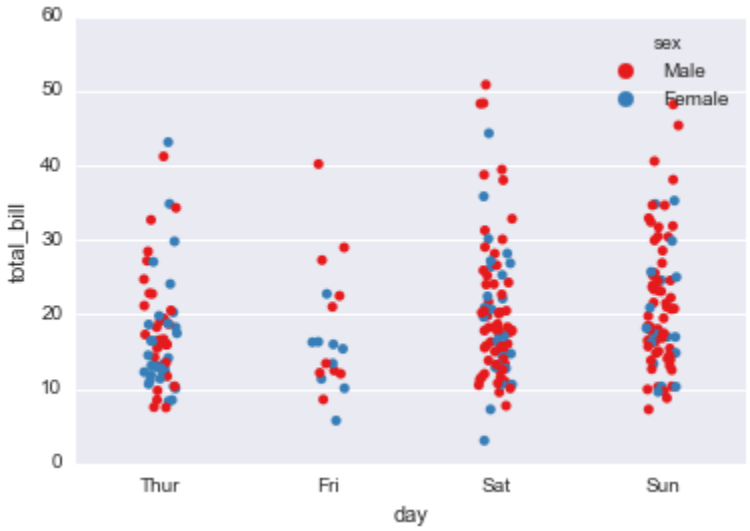
In [39]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True)
```

Out[39]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1203a8470>
```



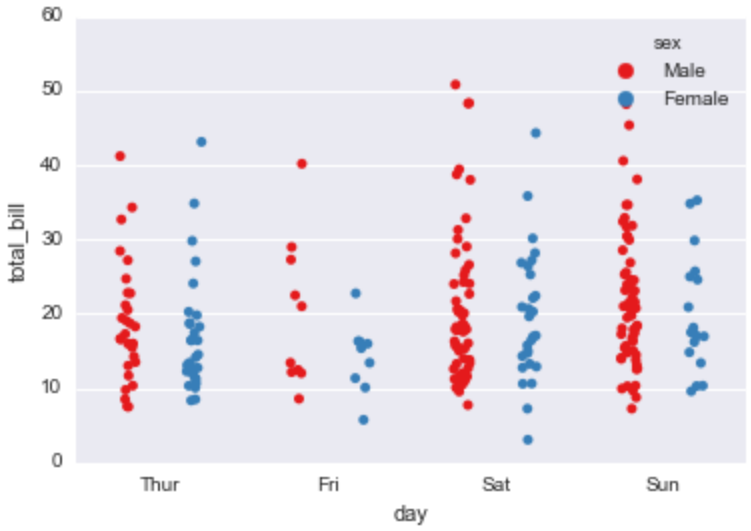
In [42]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True,hue='sex',palette=
```

Out[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12092e518>
```



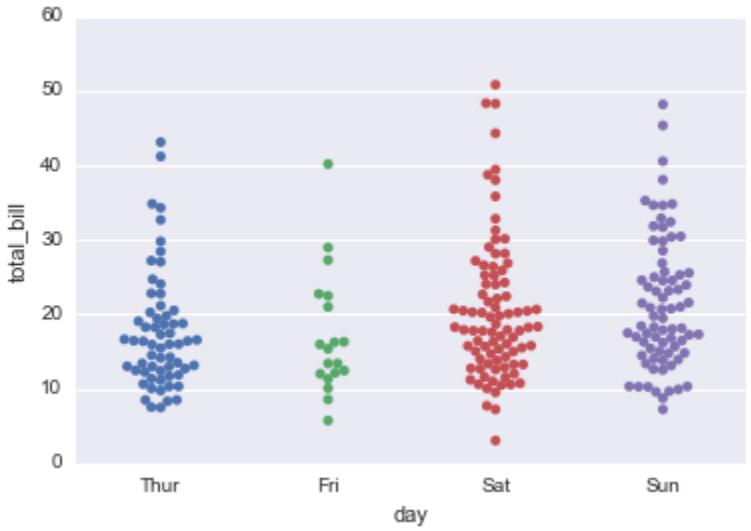
In [43]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True,hue='sex',palette=
```

Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x12099db70>



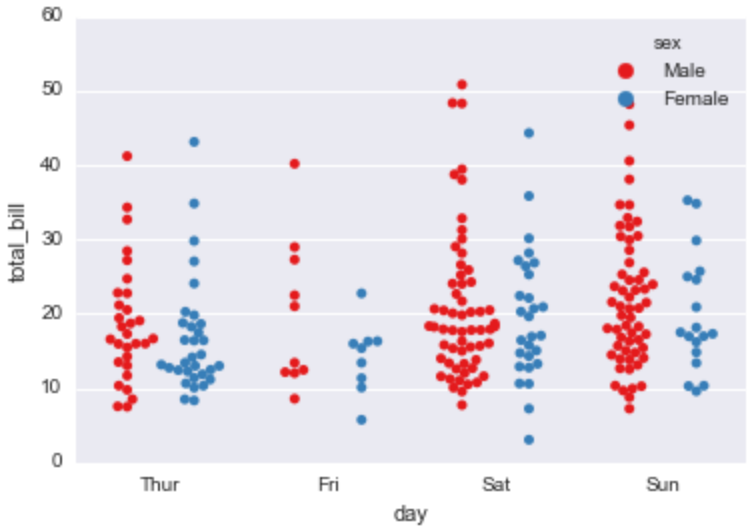
In [44]:
```
sns.swarmplot(x="day", y="total_bill", data=tips)
```

Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x120c463c8>



In [47]:
```
sns.swarmplot(x="day", y="total_bill",hue='sex',data=tips, palette="Set1", spli
```

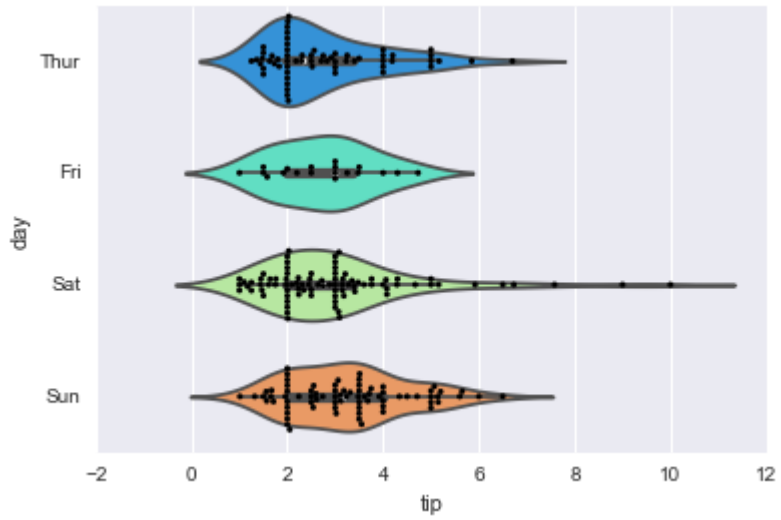
Out[47]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1211b6da0>
```



Combining Categorical Plots

In [61]:
```
sns.violinplot(x="tip", y="day", data=tips,palette='rainbow')
sns.swarmplot(x="tip", y="day", data=tips,color='black',size=3)
```

Out[61]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1228af668>
```

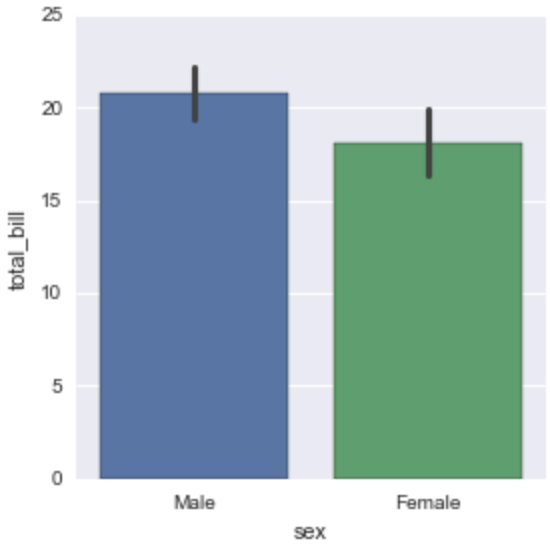


factorplot

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

In [15]:
```
sns.factorplot(x='sex',y='total_bill',data=tips,kind='bar')
```

Out[15]:
```
<seaborn.axisgrid.FacetGrid at 0x11d03a278>
```



Great Job!


(http://www.pieriandata.com)

Matrix Plots

Matrix plots allow you to plot data as color-encoded matrices and can also be used to indicate clusters within the data (later in the machine learning section we will learn how to formally cluster data).

Let's begin by exploring seaborn's heatmap and clutermap:

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

In [7]:
```
flights = sns.load_dataset('flights')
```

In [10]:
```
tips = sns.load_dataset('tips')
```

In [11]:
```
tips.head()
```

Out[11]:

	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

In [4]:
```
flights.head()
```

Out[4]:

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Heatmap

In order for a heatmap to work properly, your data should already be in a matrix form, the sns.heatmap function basically just colors it in for you. For example:

In [12]:
```
tips.head()
```

Out[12]:

	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

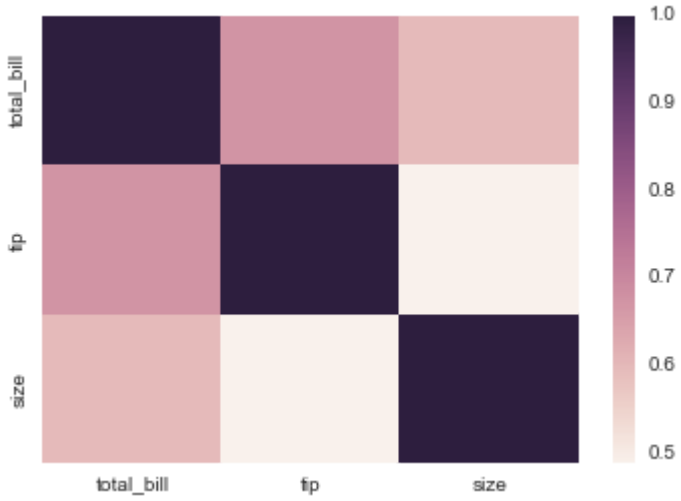
In [14]:
```
# Matrix form for correlation data
tips.corr()
```

Out[14]:

	total_bill	tip	size
total_bill	1.000000	0.675734	0.598315
tip	0.675734	1.000000	0.489299
size	0.598315	0.489299	1.000000

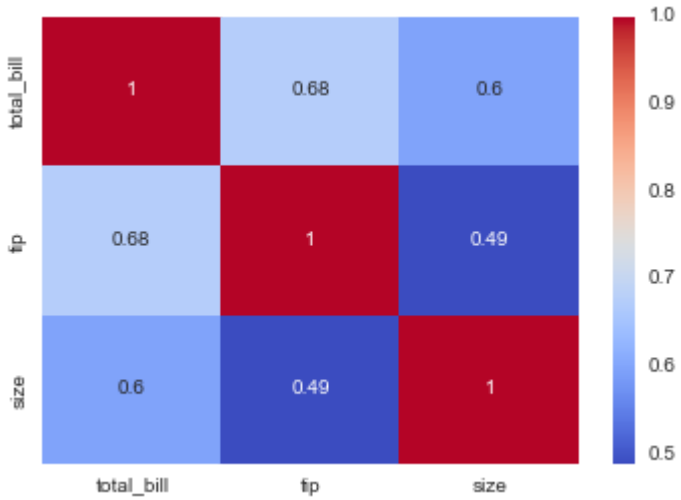
In [16]:
```
sns.heatmap(tips.corr())
```

Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c31d470>
```



In [19]:
```
sns.heatmap(tips.corr(),cmap='coolwarm',annot=True)
```

Out[19]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c97a978>
```



Or for the flights data:

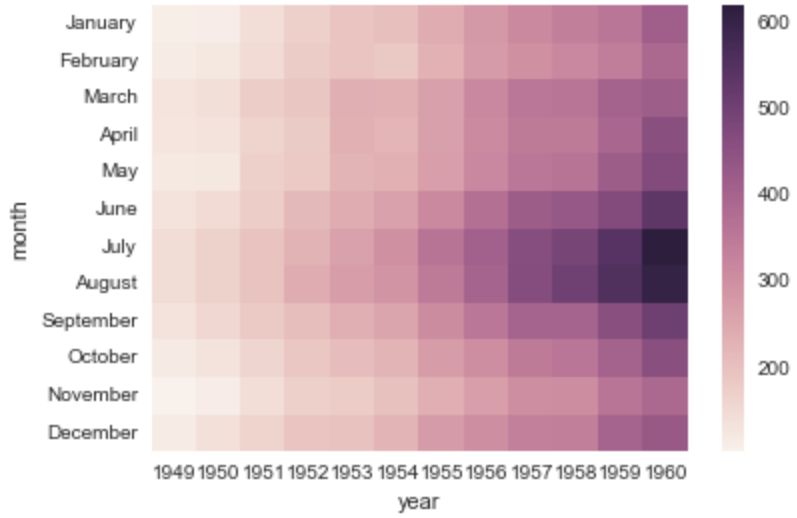
In [23]:
```
flights.pivot_table(values='passengers',index='month',columns='year')
```

Out[23]:

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
January	112	115	145	171	196	204	242	284	315	340	360	417
February	118	126	150	180	196	188	233	277	301	318	342	391
March	132	141	178	193	236	235	267	317	356	362	406	419
April	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
June	135	149	178	218	243	264	315	374	422	435	472	535
July	148	170	199	230	264	302	364	413	465	491	548	622
August	148	170	199	242	272	293	347	405	467	505	559	606
September	136	158	184	209	237	259	312	355	404	404	463	508
October	119	133	162	191	211	229	274	306	347	359	407	461
November	104	114	146	172	180	203	237	271	305	310	362	390
December	118	140	166	194	201	229	278	306	336	337	405	432

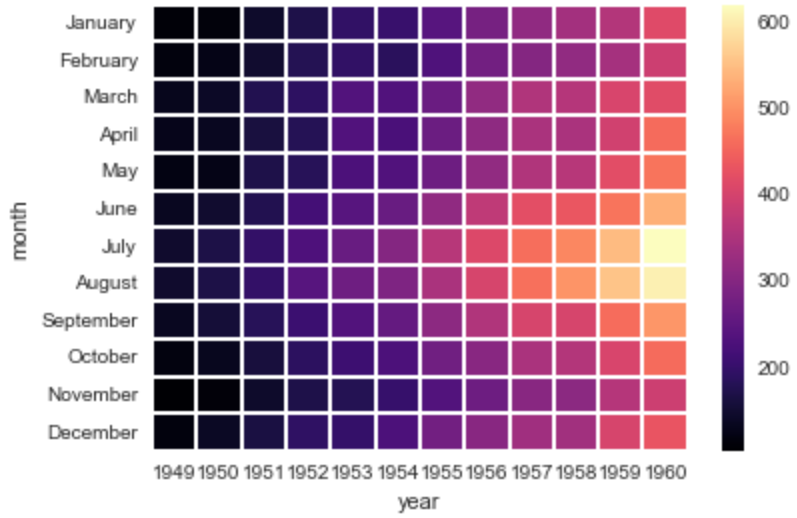
In [24]:
```
pvflights = flights.pivot_table(values='passengers',index='month',columns='year')
sns.heatmap(pvflights)
```

Out[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11cd09320>
```



In [30]:
```
sns.heatmap(pvflights,cmap='magma',linecolor='white',linewidths=1)
```

Out[30]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11d852780>
```

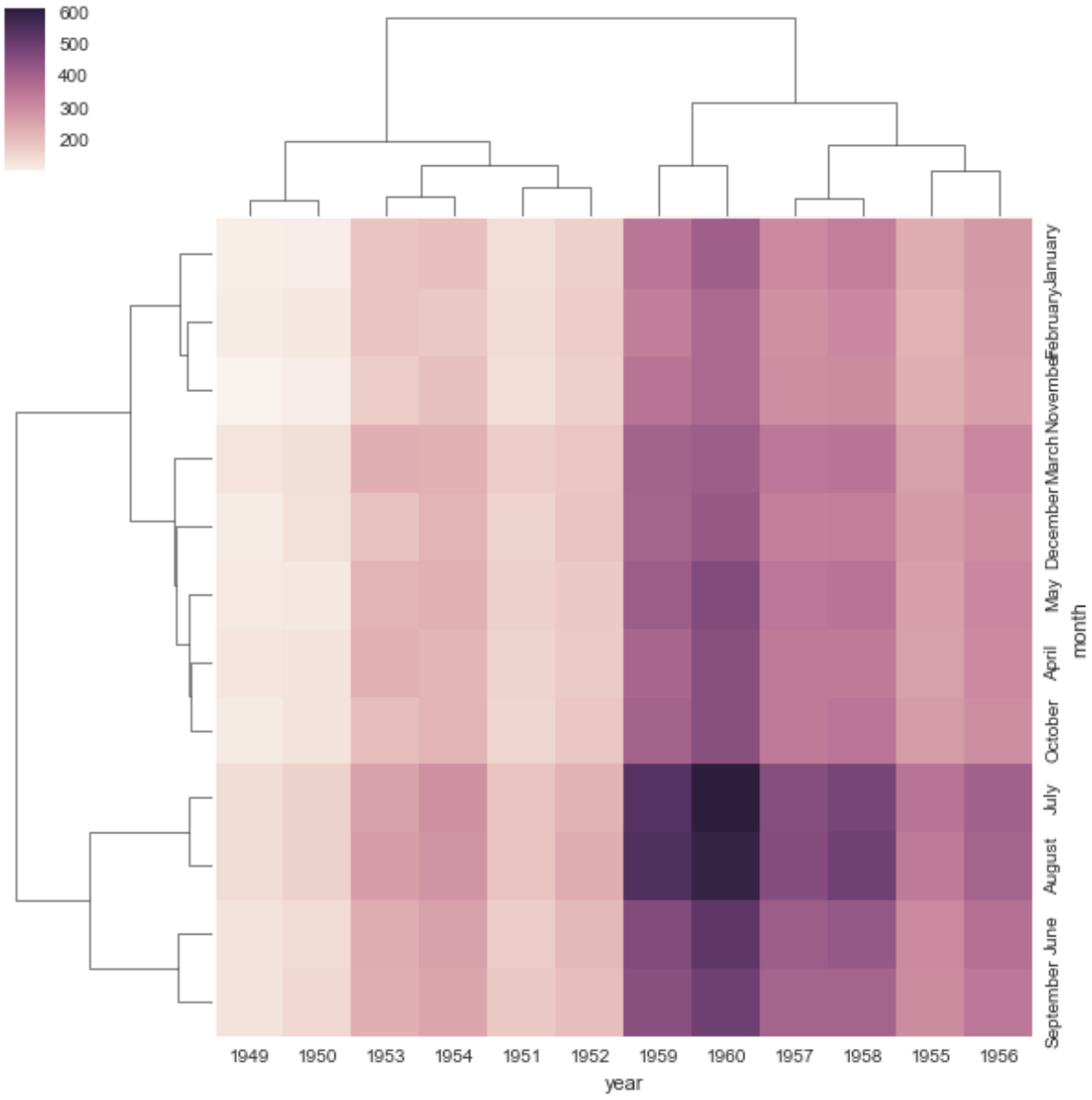


clustermap

The clustermap uses hierarchal clustering to produce a clustered version of the heatmap. For example:

In [31]:
```
sns.clustermap(pvflights)
```

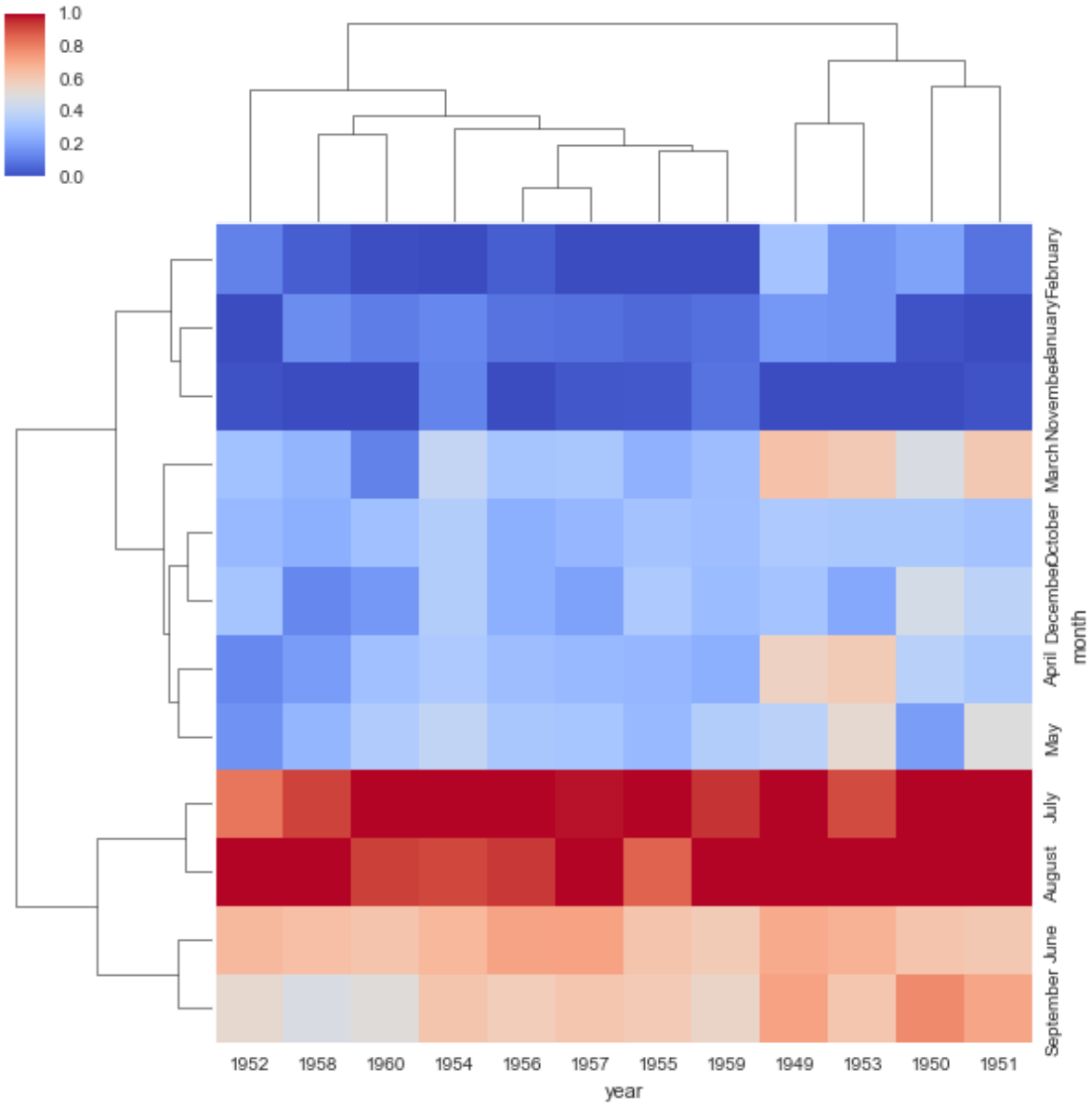
Out[31]:
```
<seaborn.matrix.ClusterGrid at 0x11dbdf4a8>
```



Notice now how the years and months are no longer in order, instead they are grouped by similarity in value (passenger count). That means we can begin to infer things from this plot, such as August and July being similar (makes sense, since they are both summer travel months)

In [34]:
```
# More options to get the information a little clearer like normalization
sns.clustermap(pvflights,cmap='coolwarm',standard_scale=1)
```

Out[34]: <seaborn.matrix.ClusterGrid at 0x11ef9d390>



Great Job!


(http://www.pieriandata.com)

Grids

Grids are general types of plots that allow you to map plot types to rows and columns of a grid, this helps you create similar plots separated by features.

In [22]:
```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [27]:
```
iris = sns.load_dataset('iris')
```

In [28]:
```
iris.head()
```

Out[28]:

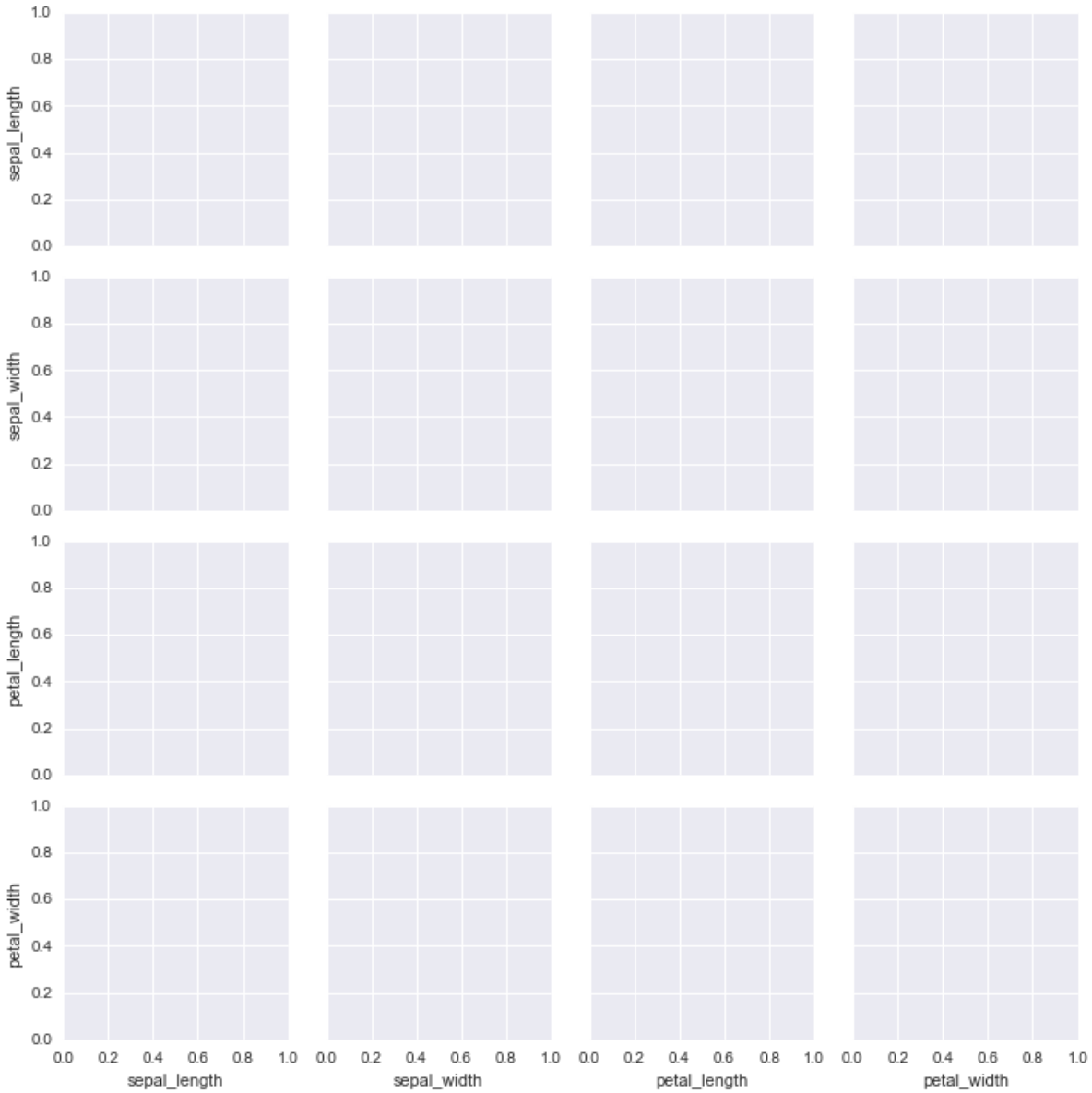
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

PairGrid

Pairgrid is a subplot grid for plotting pairwise relationships in a dataset.

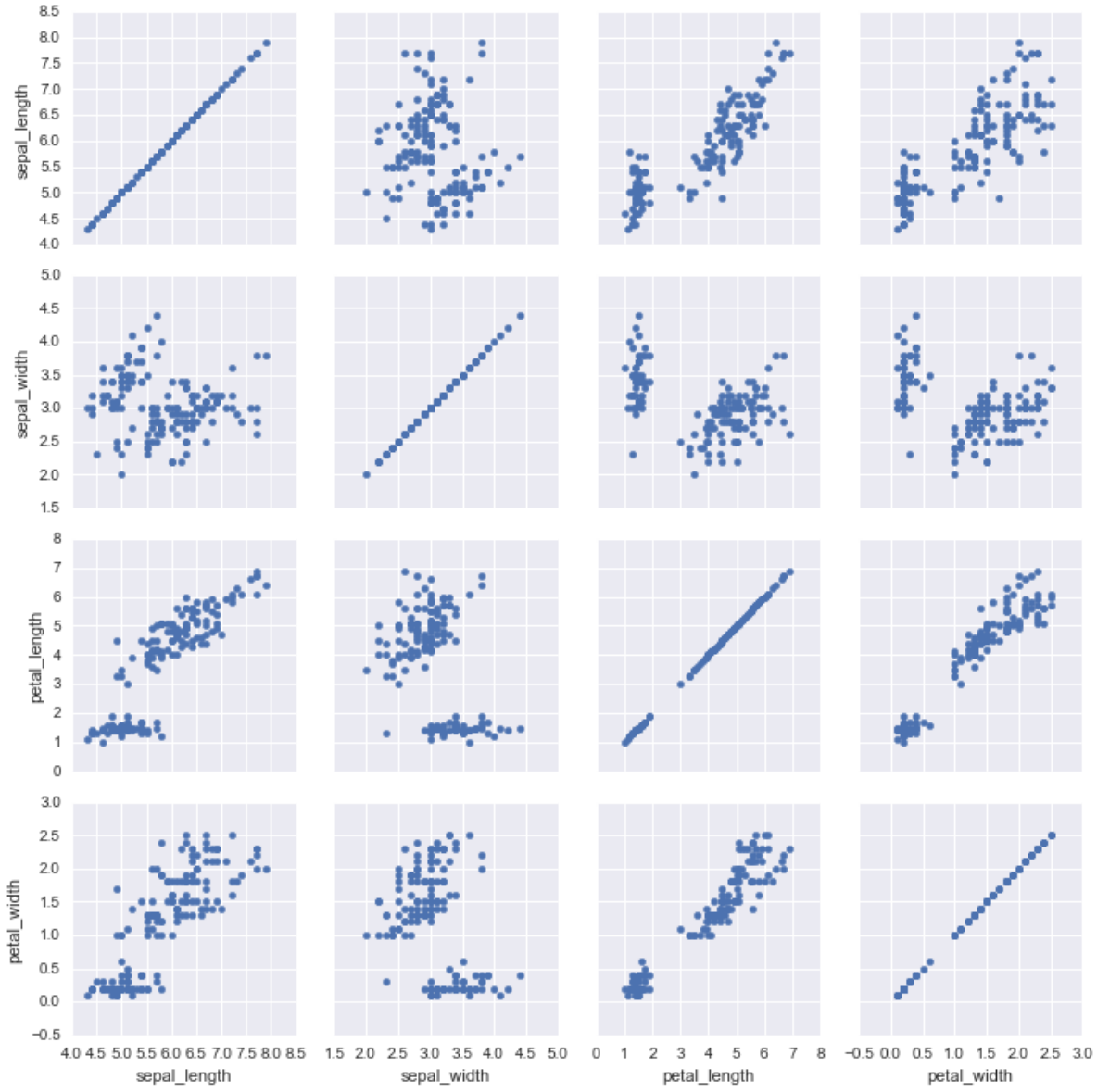
In [25]:
```
# Just the Grid
sns.PairGrid(iris)
```

Out[25]:
```
<seaborn.axisgrid.PairGrid at 0x11e39bb38>
```



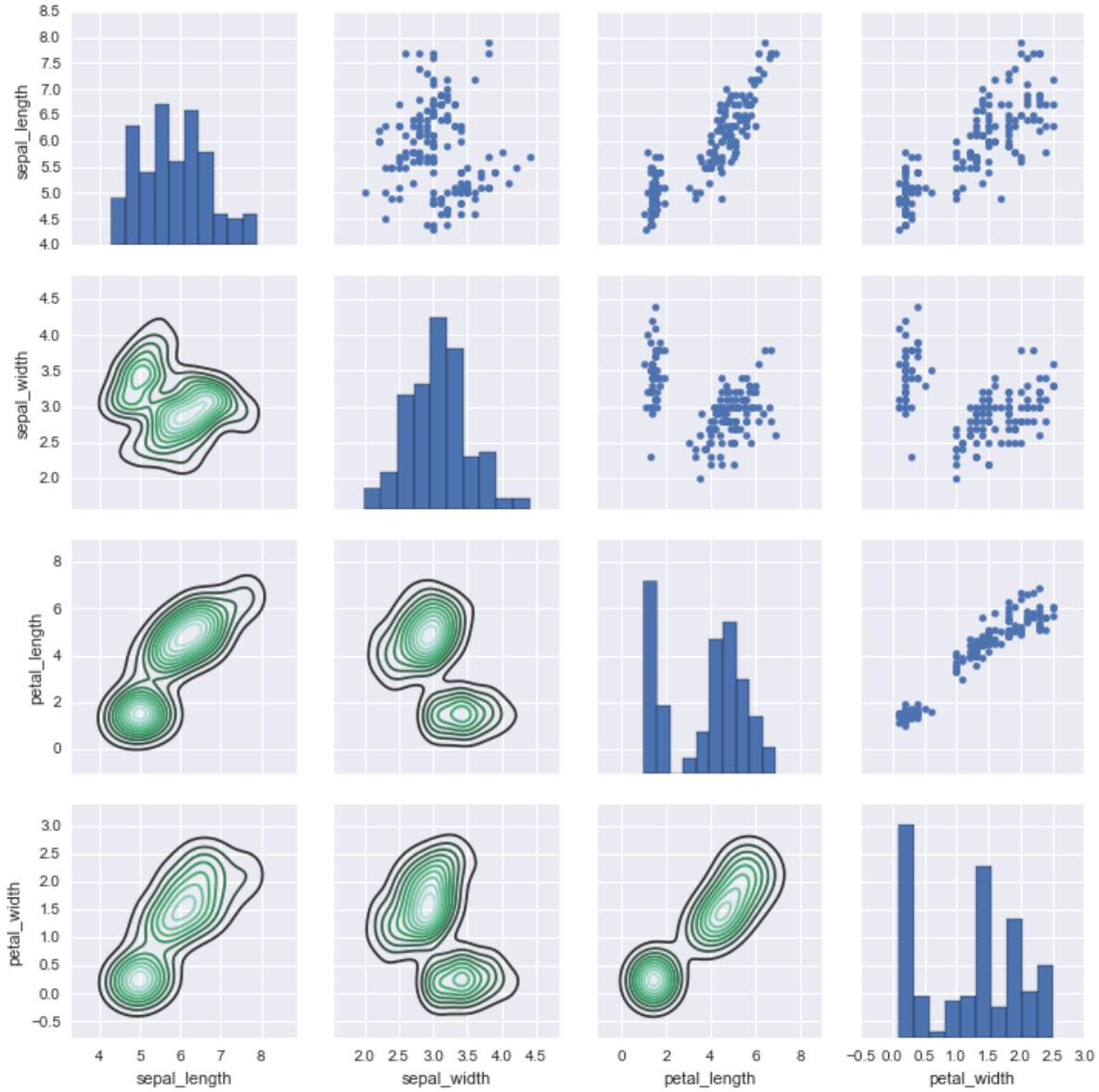
In [26]:
```
# Then you map to the grid
g = sns.PairGrid(iris)
g.map(plt.scatter)
```

Out[26]: <seaborn.axisgrid.PairGrid at 0x11f431208>



In [30]:
```
# Map to upper,lower, and diagonal
g = sns.PairGrid(iris)
g.map_diag(plt.hist)
g.map_upper(plt.scatter)
g.map_lower(sns.kdeplot)
```

Out[30]:
```
<seaborn.axisgrid.PairGrid at 0x121944128>
```

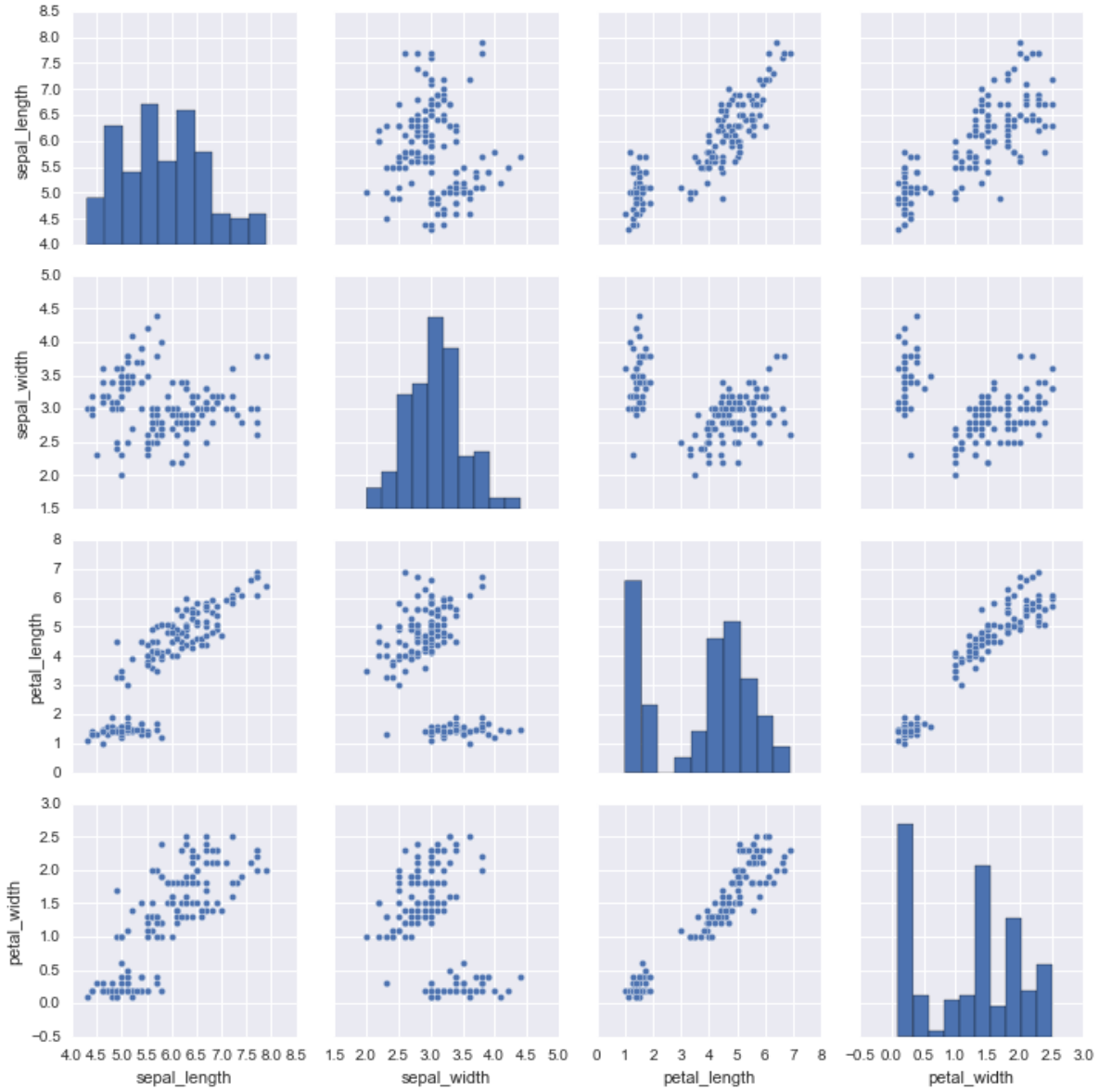


pairplot

pairplot is a simpler version of PairGrid (you'll use quite often)

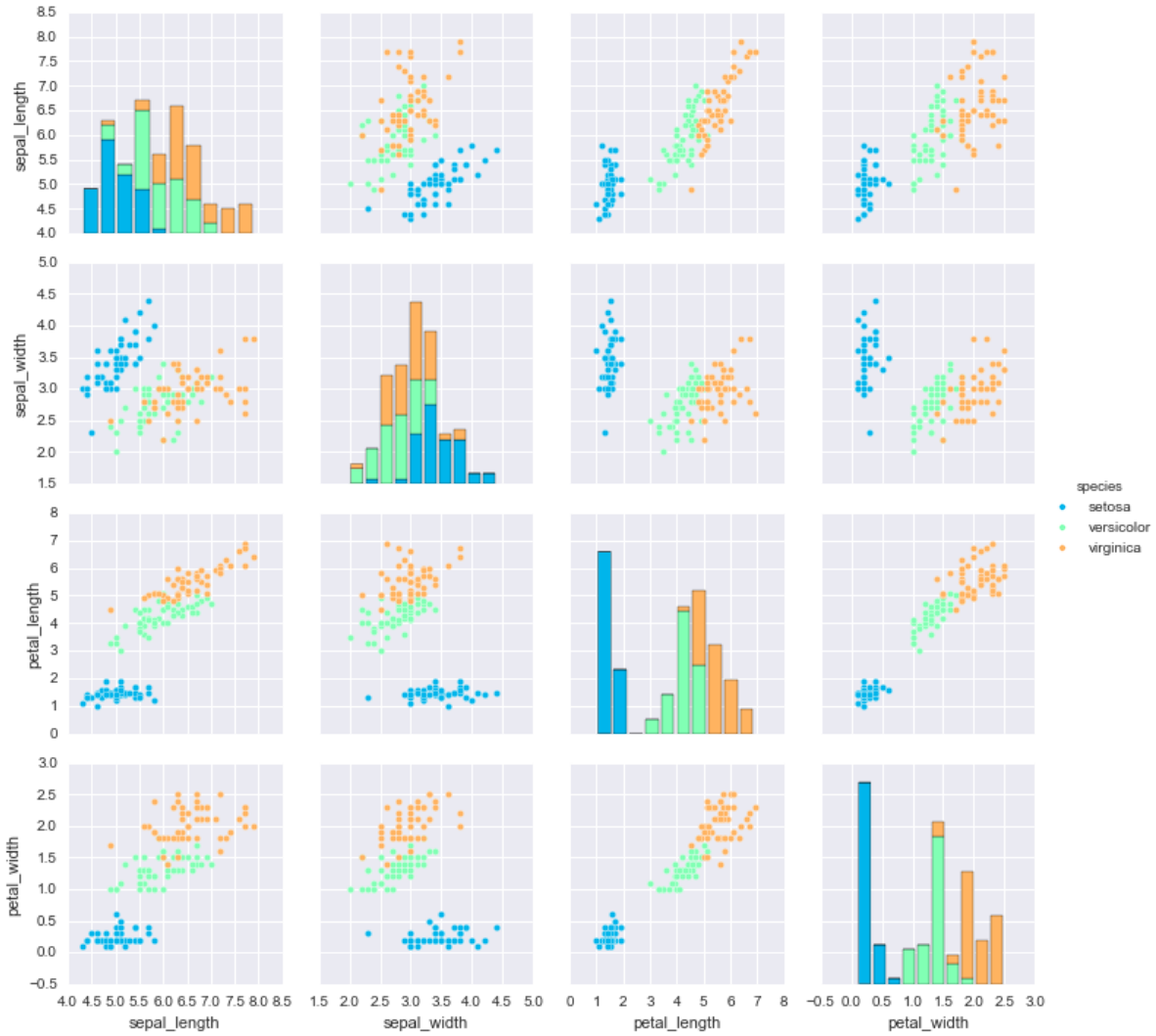
In [31]:
```
sns.pairplot(iris)
```

Out[31]:
```
<seaborn.axisgrid.PairGrid at 0x12024b5c0>
```



In [33]:
```
sns.pairplot(iris,hue='species',palette='rainbow')
```

Out[33]:
```
<seaborn.axisgrid.PairGrid at 0x12633f0f0>
```



Facet Grid

FacetGrid is the general way to create grids of plots based off of a feature:

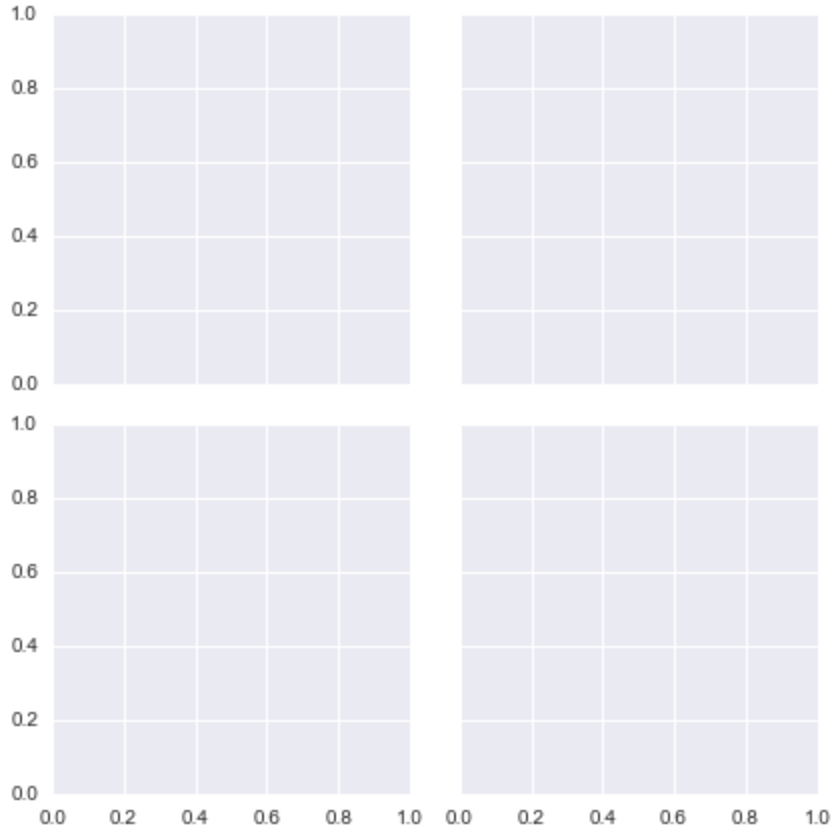
In [34]:
```
tips = sns.load_dataset('tips')
```

In [35]:
```
tips.head()
```

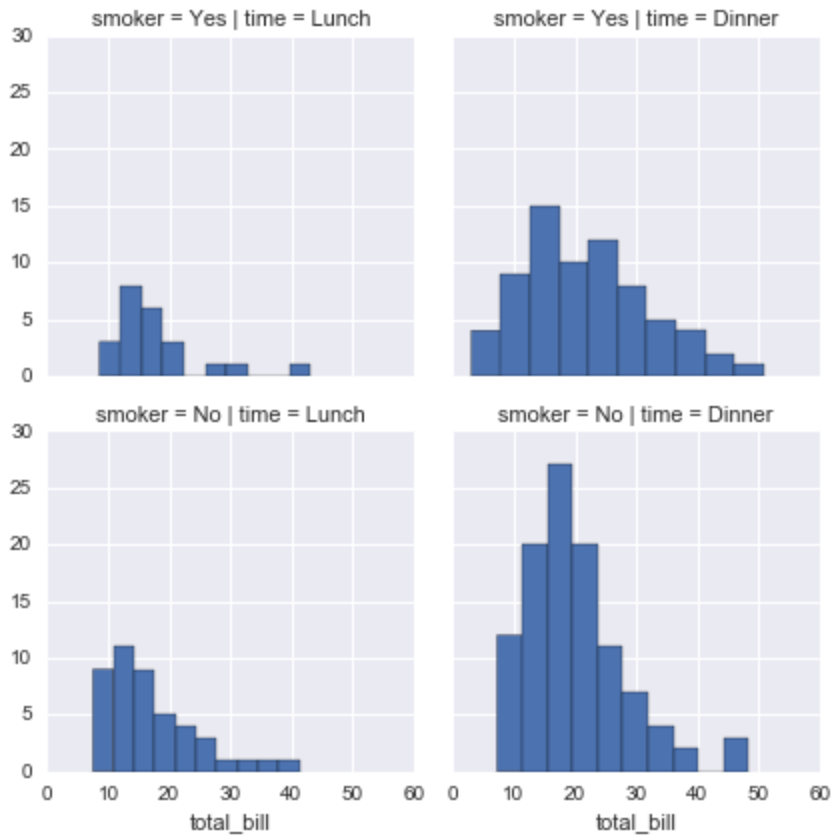
Out[35]:

	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

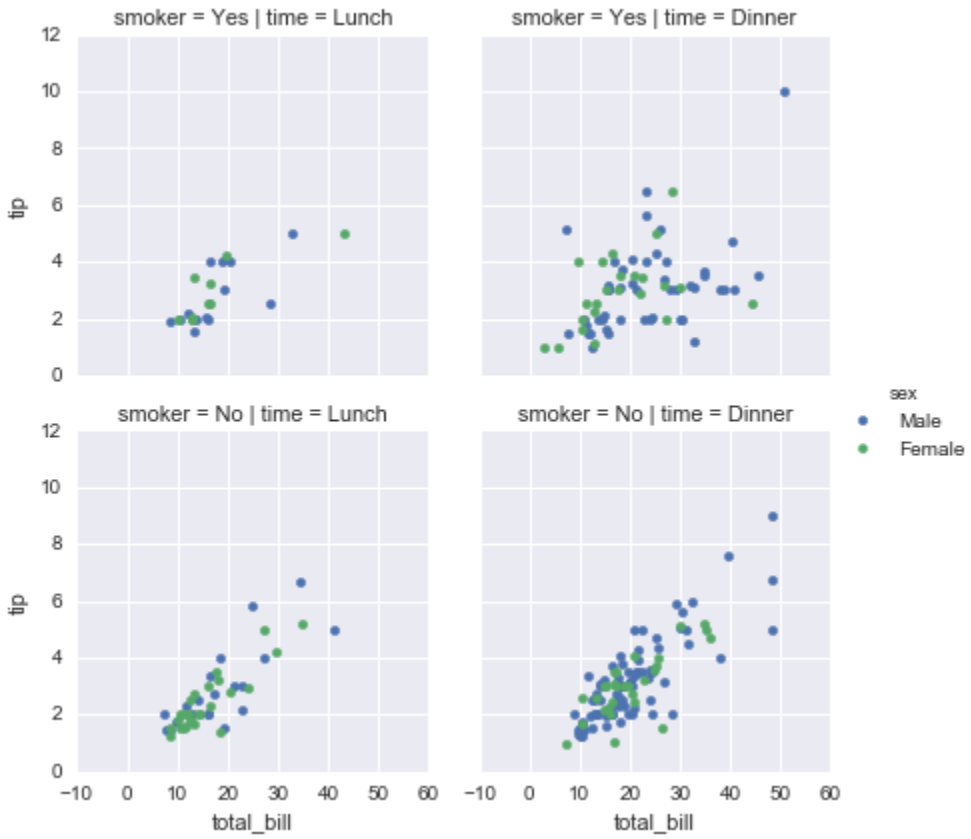
In [36]:
```
# Just the Grid
g = sns.FacetGrid(tips, col="time", row="smoker")
```



In [37]:
```
g = sns.FacetGrid(tips, col="time",  row="smoker")
g = g.map(plt.hist, "total_bill")
```



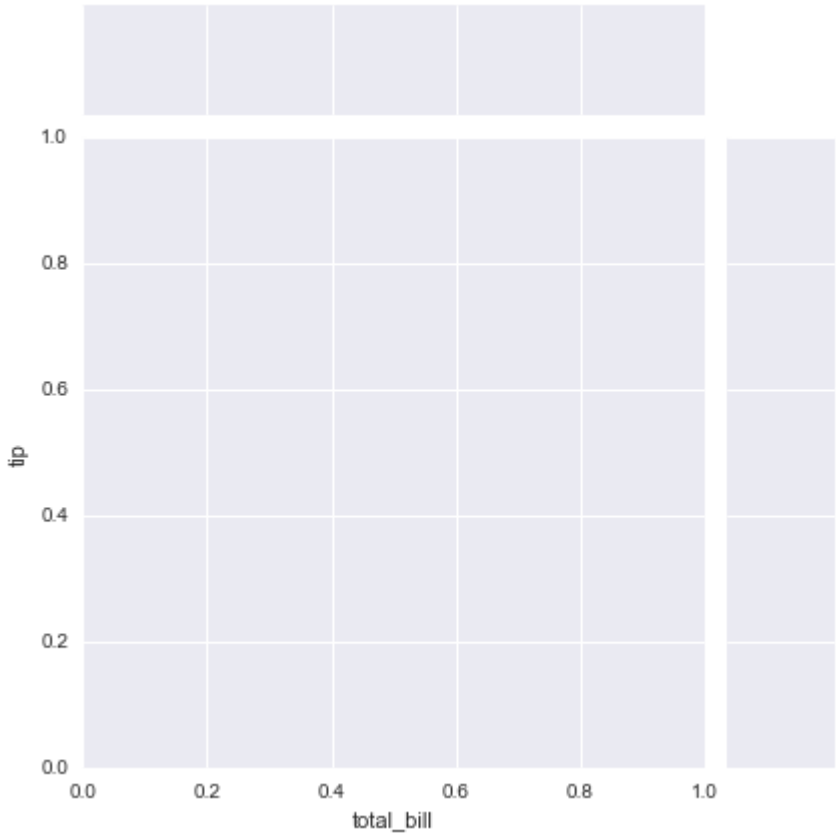
In [42]:
```
g = sns.FacetGrid(tips, col="time",  row="smoker",hue='sex')
# Notice hwo the arguments come after plt.scatter call
g = g.map(plt.scatter, "total_bill", "tip").add_legend()
```



JointGrid

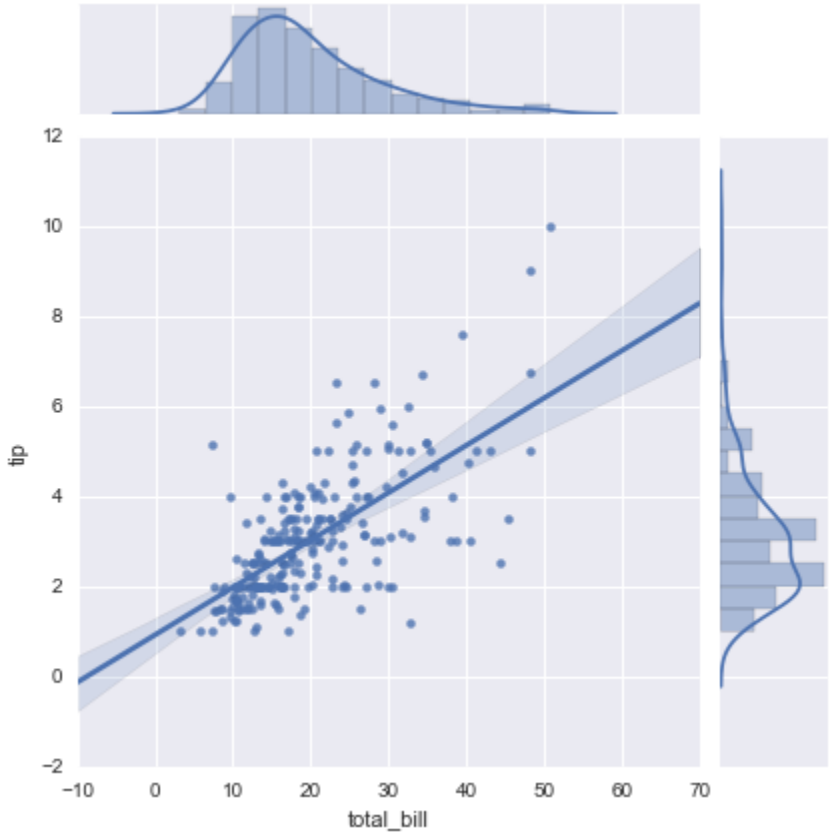
JointGrid is the general version for jointplot() type grids, for a quick example:

In [43]:
```
g = sns.JointGrid(x="total_bill", y="tip", data=tips)
```



In [45]:
```
g = sns.JointGrid(x="total_bill", y="tip", data=tips)
g = g.plot(sns.regplot, sns.distplot)
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```



Reference the documentation as necessary for grid types, but most of the time you'll just use the easier plots discussed earlier.

Great Job!


(http://www.pieriandata.com)

Regression Plots

Seaborn has many built-in capabilities for regression plots, however we won't really discuss regression until the machine learning section of the course, so we will only cover the **lmplot()** function for now.

lmplot allows you to display linear models, but it also conveniently allows you to split up those plots based off of features, as well as coloring the hue based off of features.

Let's explore how this works:

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

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

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

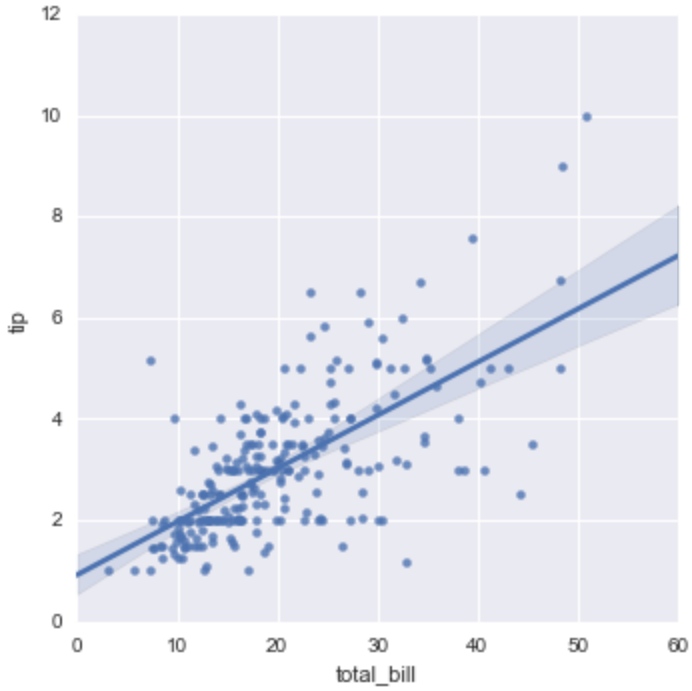
Out[3]:

	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

lmplot()

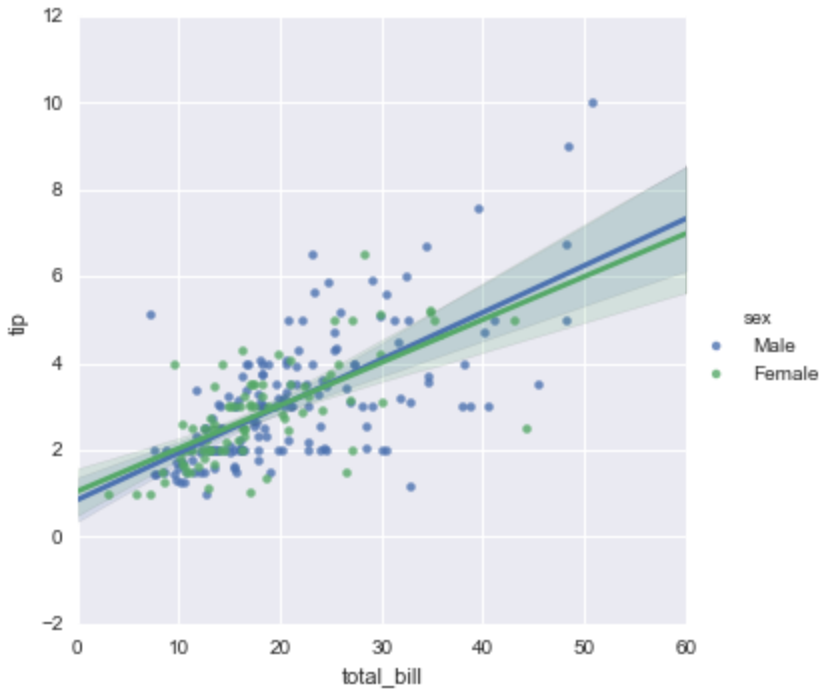
In [5]:
```
sns.lmplot(x='total_bill',y='tip',data=tips)
```

Out[5]:
```
<seaborn.axisgrid.FacetGrid at 0x117c81048>
```



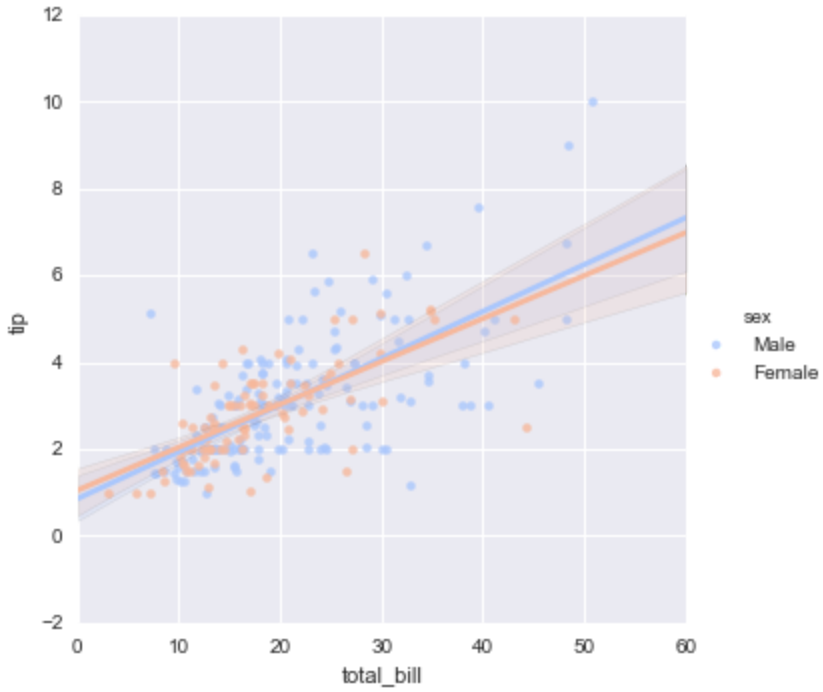
In [6]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,hue='sex')
```

Out[6]:
```
<seaborn.axisgrid.FacetGrid at 0x11bf59f98>
```



In [13]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,hue='sex',palette='coolwarm')
```

Out[13]:
```
<seaborn.axisgrid.FacetGrid at 0x11d0ca080>
```

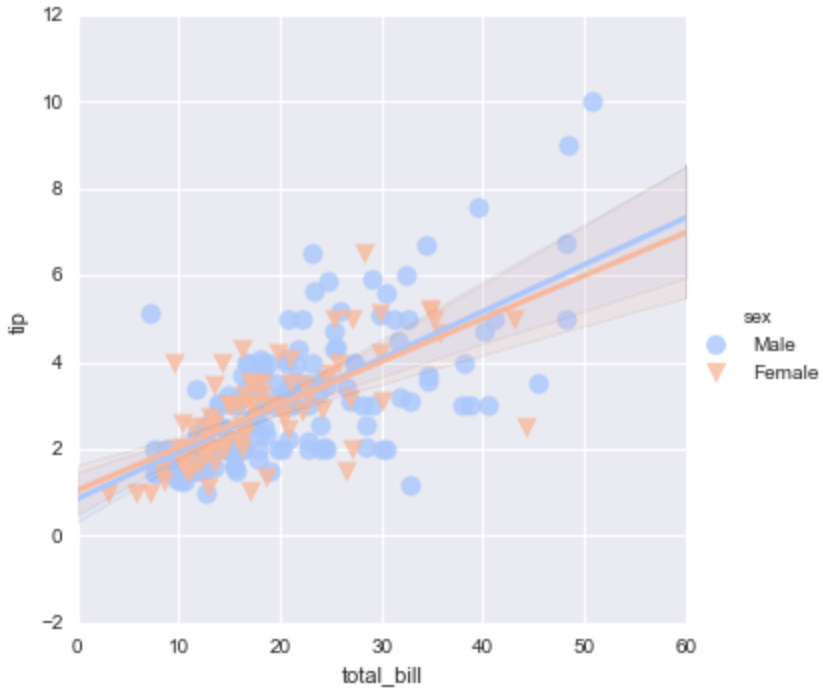


Working with Markers

lmplot kwargs get passed through to **regplot** which is a more general form of lmplot(). regplot has a scatter_kws parameter that gets passed to plt.scatter. So you want to set the s parameter in that dictionary, which corresponds (a bit confusingly) to the squared markersize. In other words you end up passing a dictionary with the base matplotlib arguments, in this case, s for size of a scatter plot. In general, you probably won't remember this off the top of your head, but instead reference the documentation.

In [16]:
```
# http://matplotlib.org/api/markers_api.html
sns.lmplot(x='total_bill',y='tip',data=tips,hue='sex',palette='coolwarm',
           markers=['o','v'],scatter_kws={'s':100})
```

Out[16]:
```
<seaborn.axisgrid.FacetGrid at 0x11d8ab748>
```

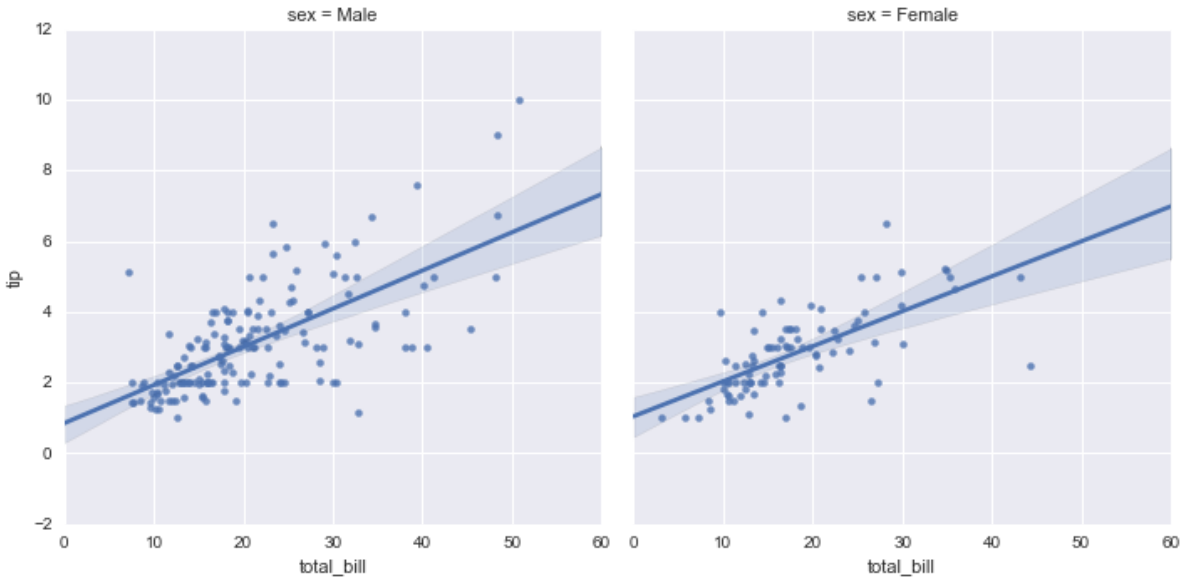


Using a Grid

We can add more variable separation through columns and rows with the use of a grid. Just indicate this with the col or row arguments:

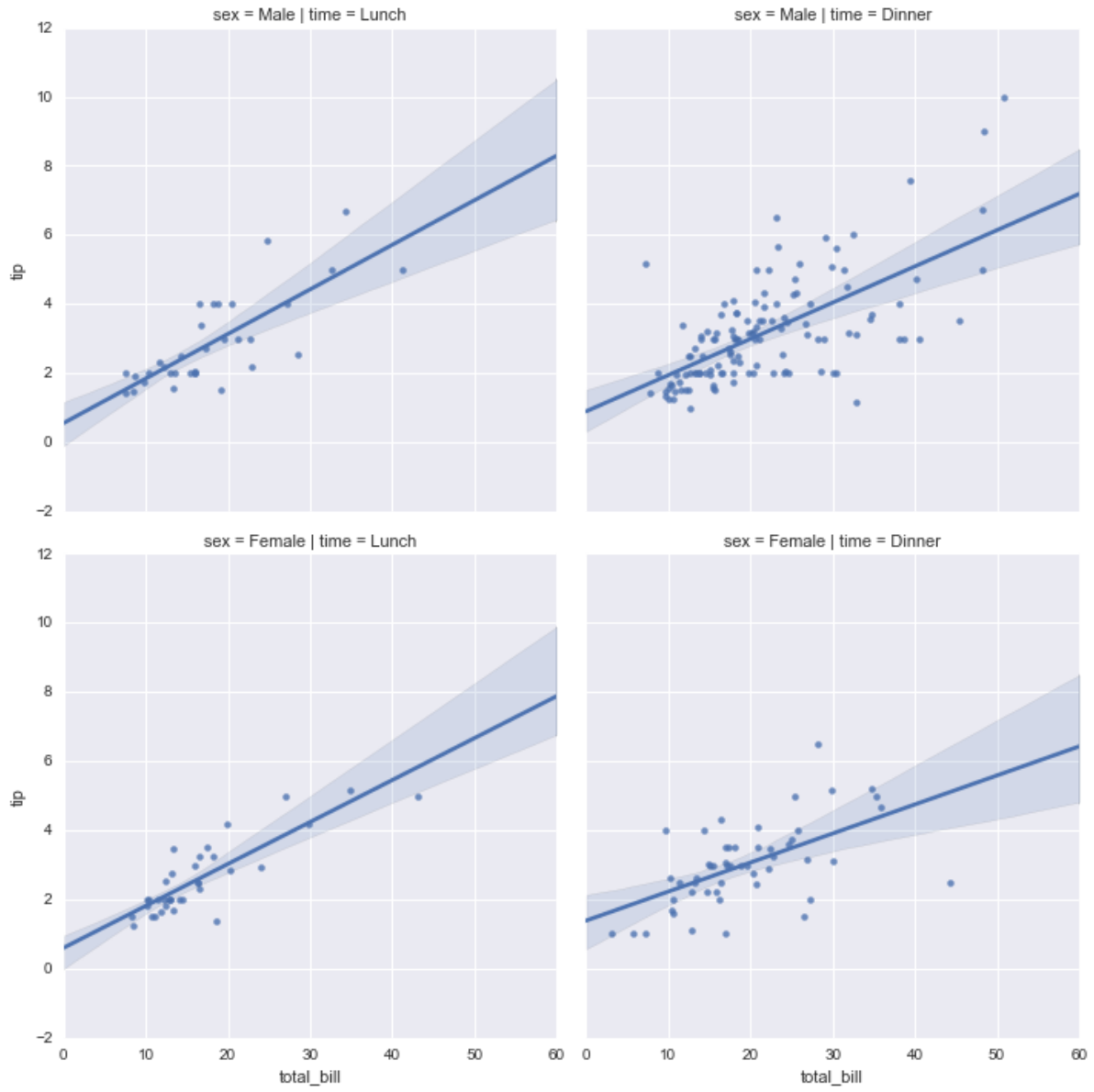
In [28]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,col='sex')
```

Out[28]:
```
<seaborn.axisgrid.FacetGrid at 0x1215cd048>
```



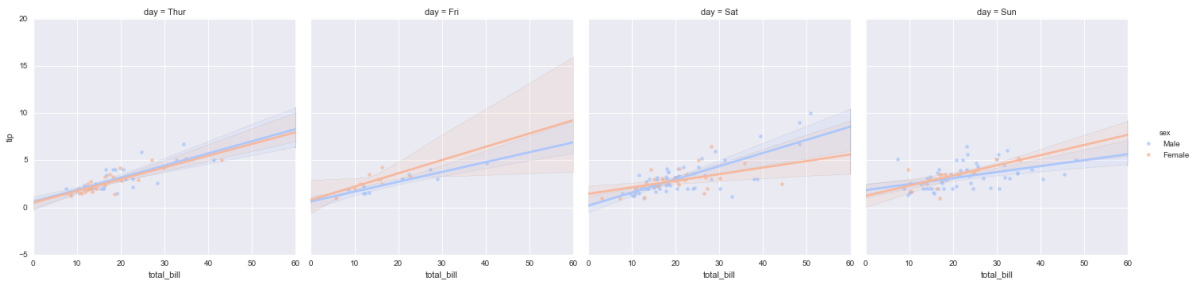
In [30]:
```
sns.lmplot(x="total_bill", y="tip", row="sex", col="time",data=tips)
```

Out[30]:
```
<seaborn.axisgrid.FacetGrid at 0x1226144e0>
```



In [24]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,col='day',hue='sex',palette='coolwa
```

Out[24]:
```
<seaborn.axisgrid.FacetGrid at 0x11cbb02e8>
```

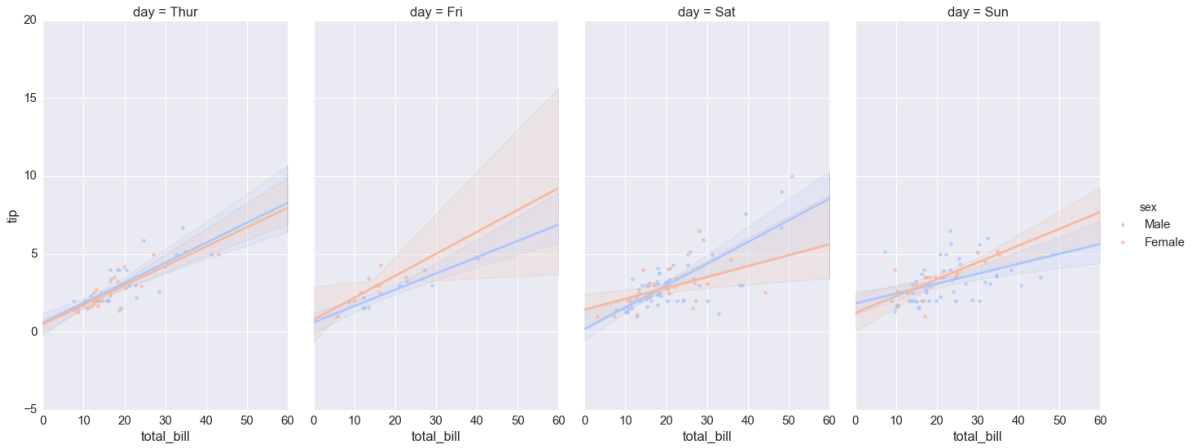


Aspect and Size

Seaborn figures can have their size and aspect ratio adjusted with the **size** and **aspect** parameters:

In [36]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,col='day',hue='sex',palette='coolwa
           aspect=0.6,size=8)
```

Out[36]:
```
<seaborn.axisgrid.FacetGrid at 0x12420e5c0>
```



You're probably wondering how to change the font size or control the aesthetics even more, check out the Style and Color Lecture and Notebook for more info on that!

Great Job!


(http://www.pieriandata.com)

Style and Color

We've shown a few times how to control figure aesthetics in seaborn, but let's now go over it formally:

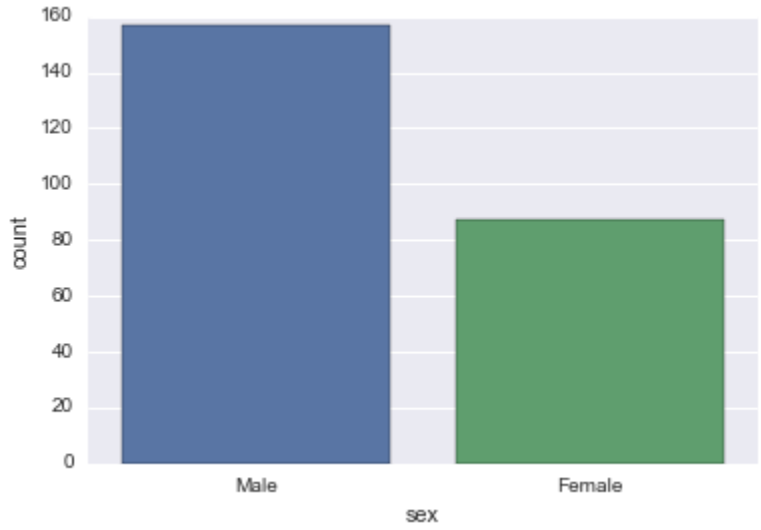
In [8]:
```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
tips = sns.load_dataset('tips')
```

Styles

You can set particular styles:

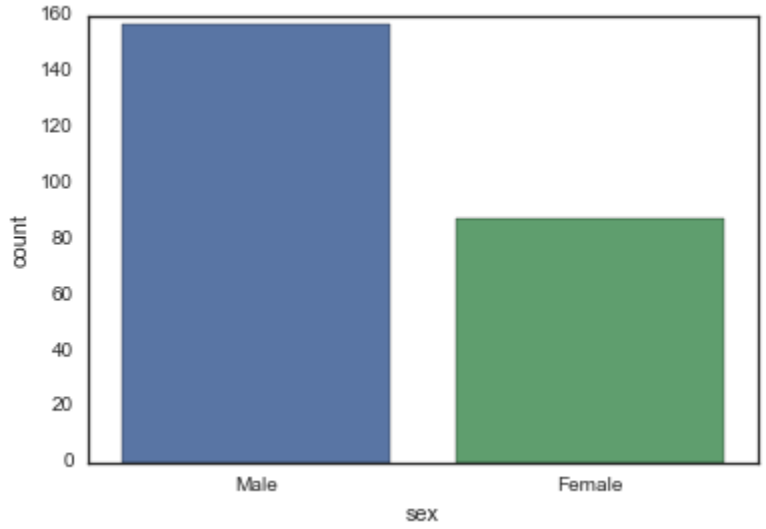
In [2]:
```
sns.countplot(x='sex',data=tips)
```

Out[2]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11990cc88>
```



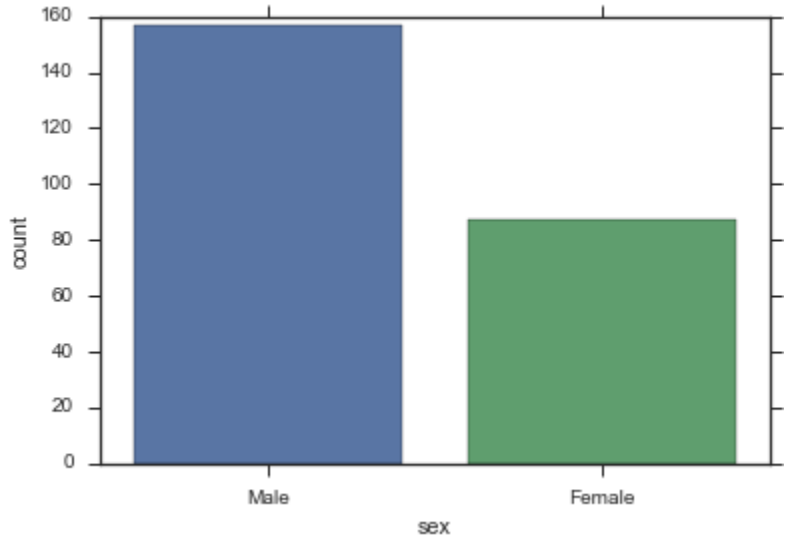
In [3]:
```
sns.set_style('white')
sns.countplot(x='sex',data=tips)
```

Out[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c2ba9b0>
```



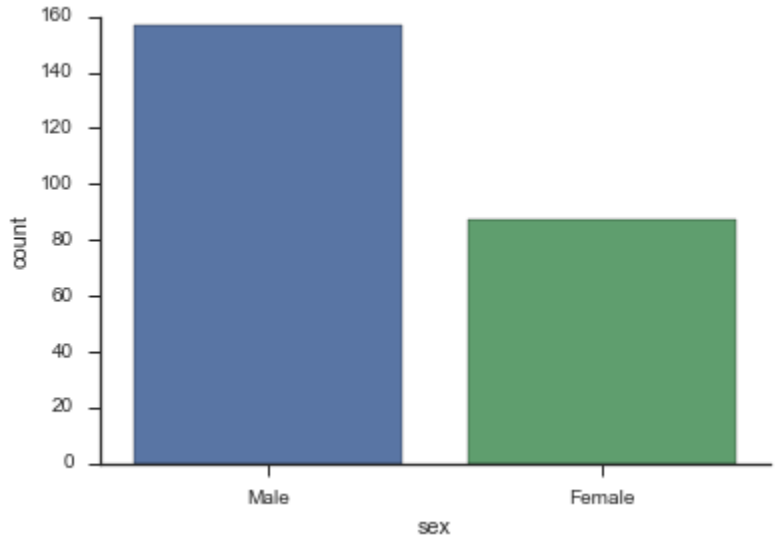
In [4]:
```
sns.set_style('ticks')
sns.countplot(x='sex',data=tips,palette='deep')
```

Out[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x119986978>
```

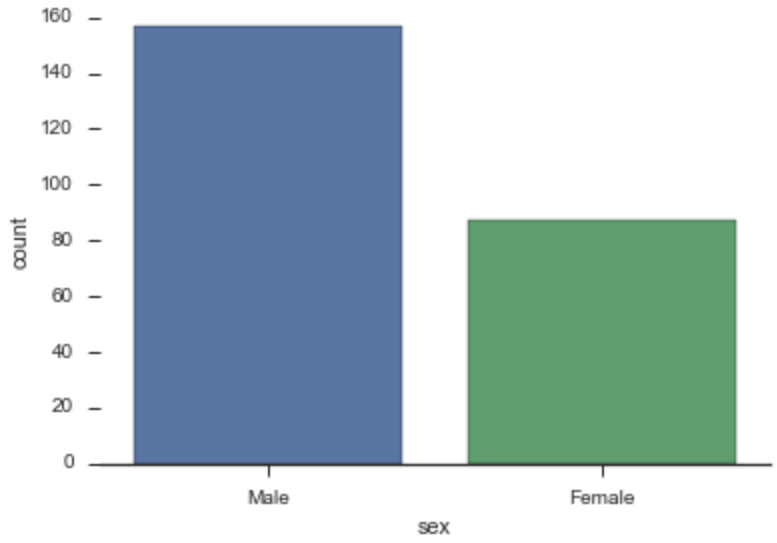


Spine Removal

In [5]:
```
sns.countplot(x='sex',data=tips)
sns.despine()
```



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



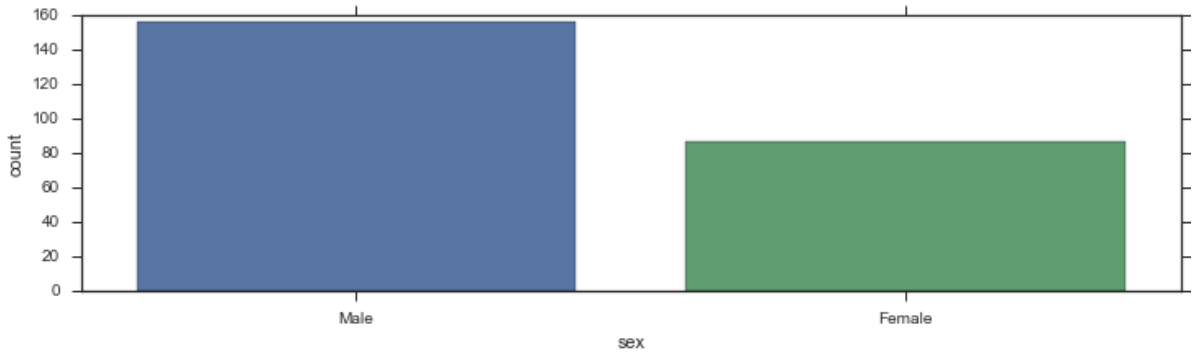
Size and Aspect

You can use matplotlib's **plt.figure(figsize=(width,height)* * to change the size of most seaborn plots.

You can control the size and aspect ratio of most seaborn grid plots by passing in parameters: size, and aspect. For example:

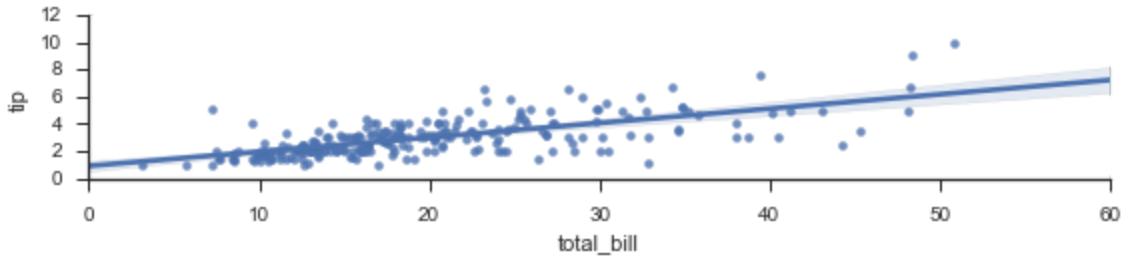
In [11]:
```
# Non Grid Plot
plt.figure(figsize=(12,3))
sns.countplot(x='sex',data=tips)
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11cabbf28>
```



In [13]:
```
# Grid Type Plot
sns.lmplot(x='total_bill',y='tip',size=2,aspect=4,data=tips)
```

Out[13]:
```
<seaborn.axisgrid.FacetGrid at 0x11cd69048>
```

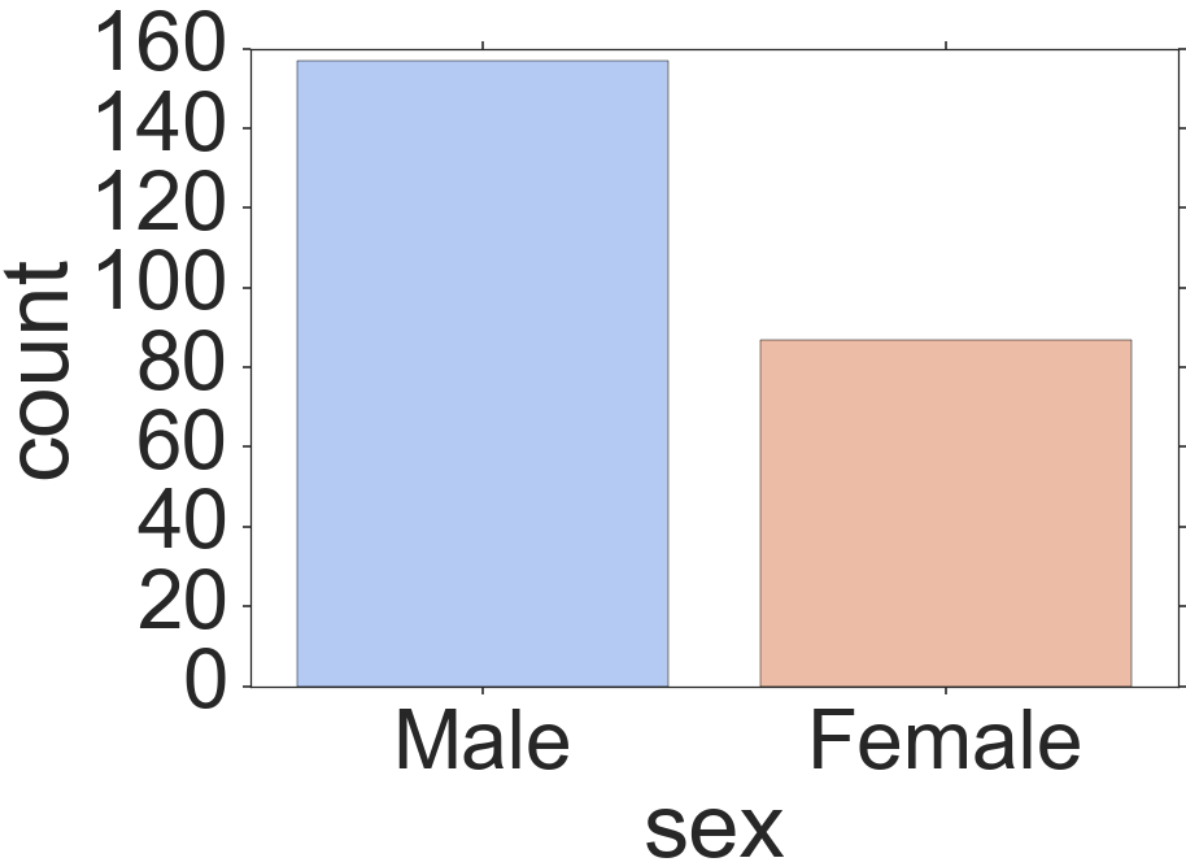


Scale and Context

The set_context() allows you to override default parameters:

In [17]:
```
sns.set_context('poster',font_scale=4)
sns.countplot(x='sex',data=tips,palette='coolwarm')
```

Out[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e2a2128>
```



Check out the documentation page for more info on these topics:
https://stanford.edu/~mwaskom/software/seaborn/tutorial/aesthetics.html
(https://stanford.edu/~mwaskom/software/seaborn/tutorial/aesthetics.html)

In [15]:
```
sns.puppyplot()
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/bs4/__init__.py:166: UserWa
rning: No parser was explicitly specified, so I'm using the best available HT
ML parser for this system ("lxml"). This usually isn't a problem, but if you
run this code on another system, or in a different virtual environment, it ma
y use a different parser and behave differently.

To get rid of this warning, change this:

 BeautifulSoup([your markup])

to this:

 BeautifulSoup([your markup], "lxml")

  markup_type=markup_type))
```

Out[15]: Titan the Pit Bull Terrier Pictures 1058495

Great Job!


(http://www.pieriandata.com)

Seaborn Exercises

Time to practice your new seaborn skills! Try to recreate the plots below (don't worry about color schemes, just the plot itself.

The Data

We will be working with a famous titanic data set for these exercises. Later on in the Machine Learning section of the course, we will revisit this data, and use it to predict survival rates of passengers. For now, we'll just focus on the visualization of the data with seaborn:

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

In [2]:
```
sns.set_style('whitegrid')
```

In [3]:
```
titanic = sns.load_dataset('titanic')
```

In [4]:
```
titanic.head()
```

Out[4]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	d
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	

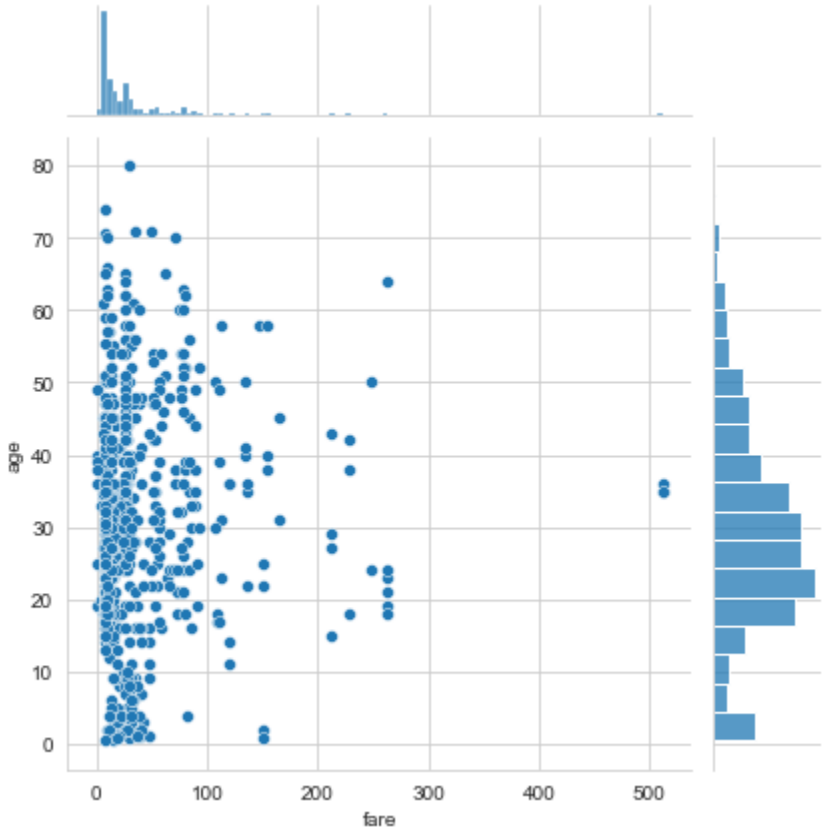
Exercises

** Recreate the plots below using the titanic dataframe. There are very few hints since most of the plots can be done with just one or two lines of code and a hint would basically give away the solution. Keep careful attention to the x and y labels for hints.**

*** *Note! In order to not lose the plot image, make sure you don't code in the cell that is directly above the plot, there is an extra cell above that one which won't overwrite that plot! ***

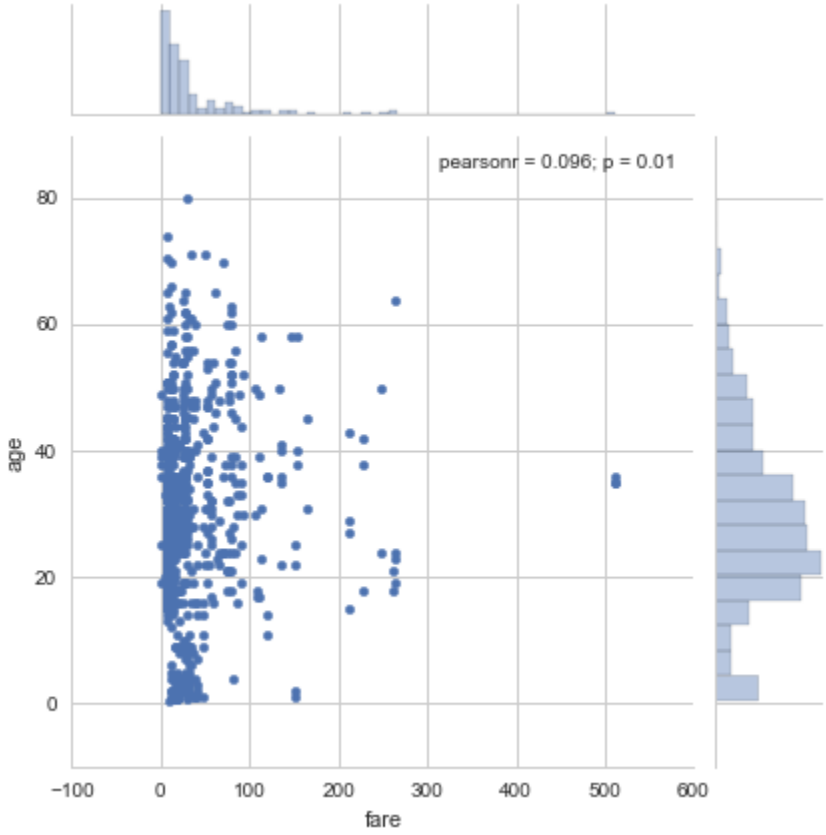
In [6]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
sns.jointplot(x='fare',y='age',data=titanic,kind='scatter')
```

Out[6]:
```
<seaborn.axisgrid.JointGrid at 0x23d6f068310>
```



In [41]:

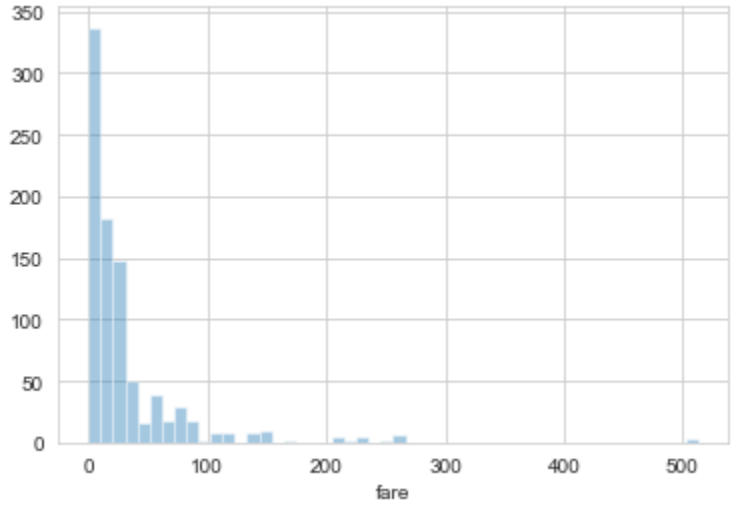
Out[41]: <seaborn.axisgrid.JointGrid at 0x11d0389e8>



In [7]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
sns.distplot(titanic['fare'],kde=False)
```

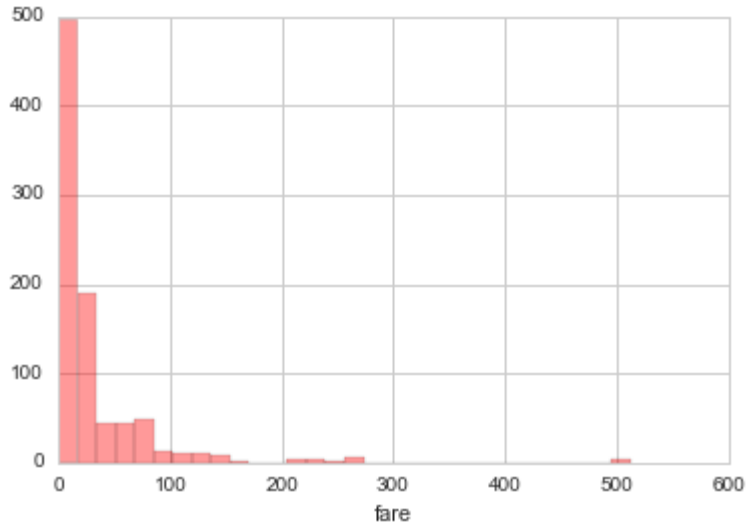
```
C:\Users\mahdi\anaconda3\lib\site-packages\seaborn\distributions.py:2619: Fut
ureWarning: `distplot` is a deprecated function and will be removed in a futu
re version. Please adapt your code to use either `displot` (a figure-level fu
nction with similar flexibility) or `histplot` (an axes-level function for hi
stograms).
  warnings.warn(msg, FutureWarning)
```

Out[7]: <AxesSubplot:xlabel='fare'>



In [44]:

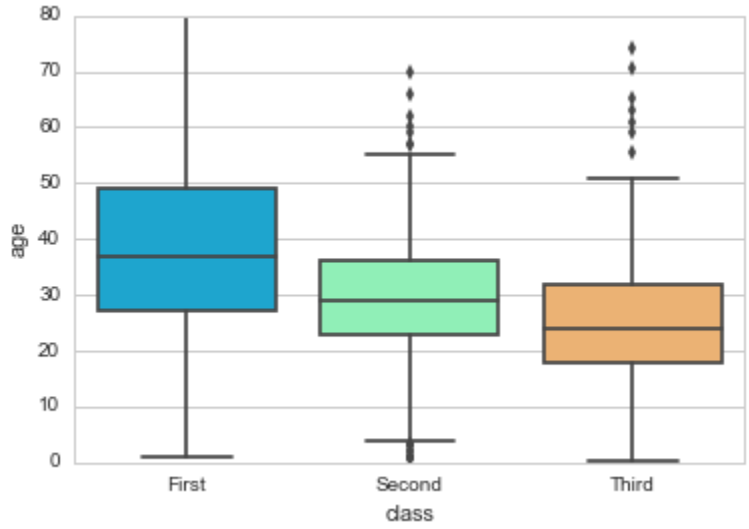
Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x11fc5ca90>



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [45]:
```

```

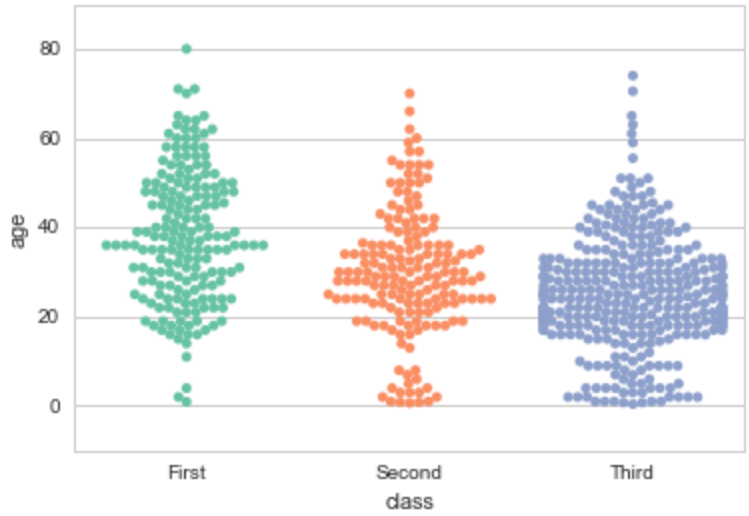
Out[45]: <matplotlib.axes._subplots.AxesSubplot at 0x11f23da90>



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [46]:
```

```

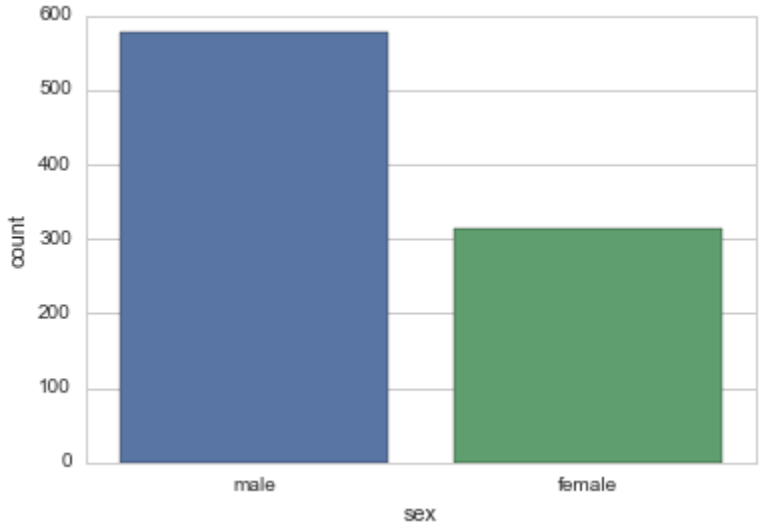
Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x11f215320>



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [47]:
```

```

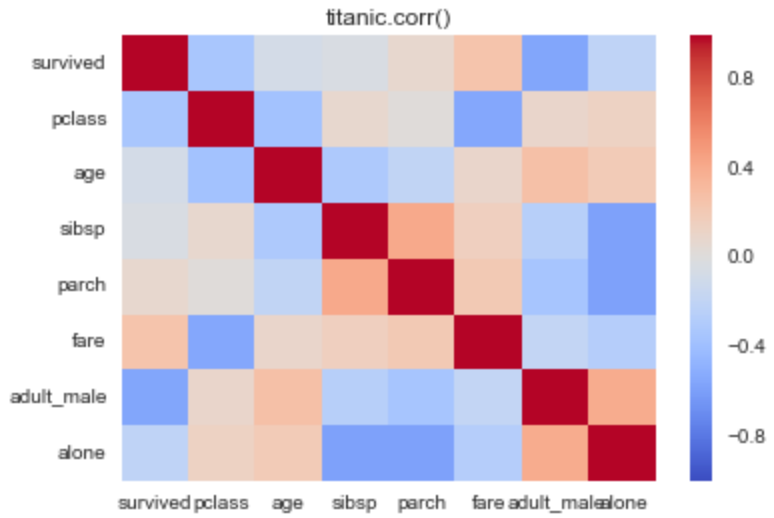
Out[47]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f207ef0>
```



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [48]:
```

```

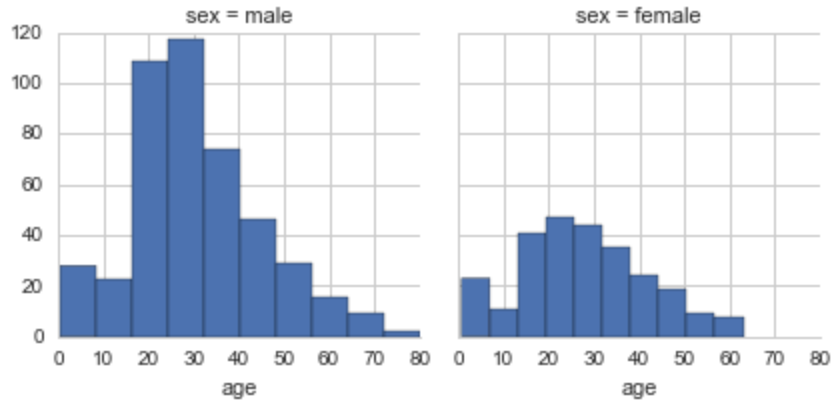
Out[48]:
```
<matplotlib.text.Text at 0x11d72da58>
```



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [49]:
```

```

Out[49]: <seaborn.axisgrid.FacetGrid at 0x11d81c240>



Great Job!

That is it for now! We'll see a lot more of seaborn practice problems in the machine learning section!


(http://www.pieriandata.com)

Seaborn Exercises - Solutions

Time to practice your new seaborn skills! Try to recreate the plots below (don't worry about color schemes, just the plot itself.

The Data

We will be working with a famous titanic data set for these exercises. Later on in the Machine Learning section of the course, we will revisit this data, and use it to predict survival rates of passengers. For now, we'll just focus on the visualization of the data with seaborn:

In [19]:
```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [27]:
```
sns.set_style('whitegrid')
```

In [28]:
```
titanic = sns.load_dataset('titanic')
```

In [40]:
```
titanic.head()
```

Out[40]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	d
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	I
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	I
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	I

Exercises

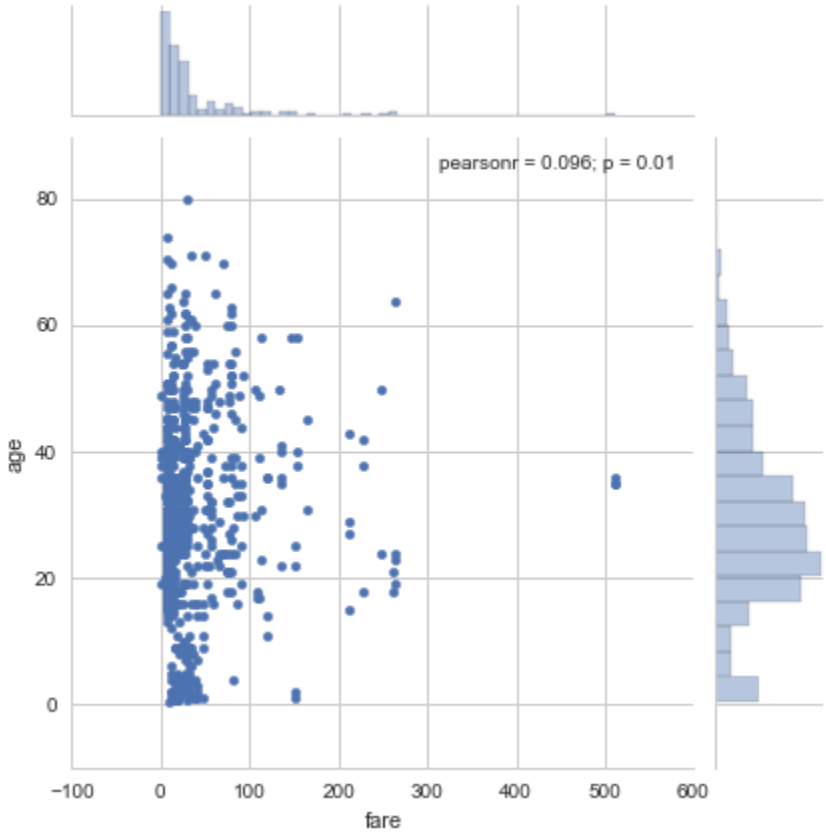
** Recreate the plots below using the titanic dataframe. There are very few hints since most of the plots can be done with just one or two lines of code and a hint would basically give away the solution. Keep careful attention to the x and y labels for hints.**

*** *Note! In order to not lose the plot image, make sure you don't code in the cell that is directly above the plot, there is an extra cell above that one which won't overwrite that plot! ***

In [42]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [41]:
```
sns.jointplot(x='fare',y='age',data=titanic)
```

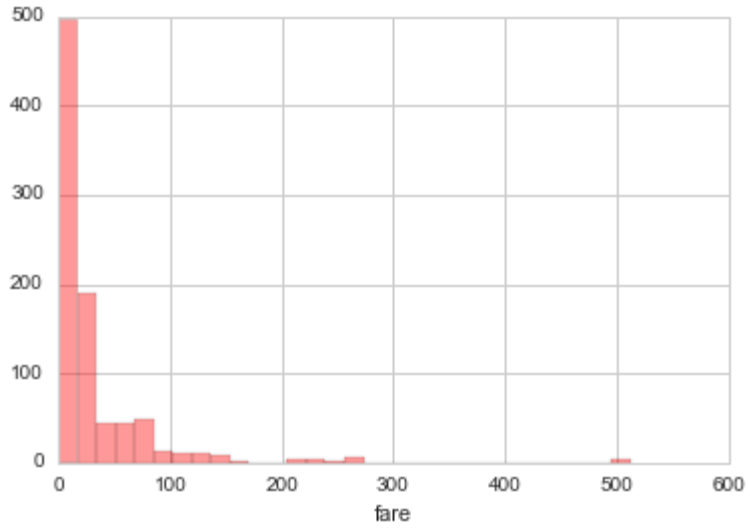
Out[41]:
```
<seaborn.axisgrid.JointGrid at 0x11d0389e8>
```



In [43]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [44]:
```
sns.distplot(titanic['fare'],bins=30,kde=False,color='red')
```

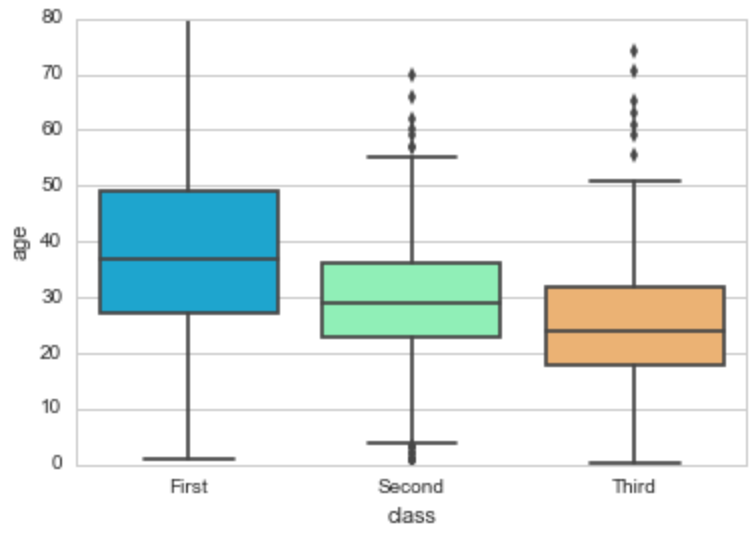
Out[44]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11fc5ca90>
```



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [45]:
```
sns.boxplot(x='class',y='age',data=titanic,palette='rainbow')
```

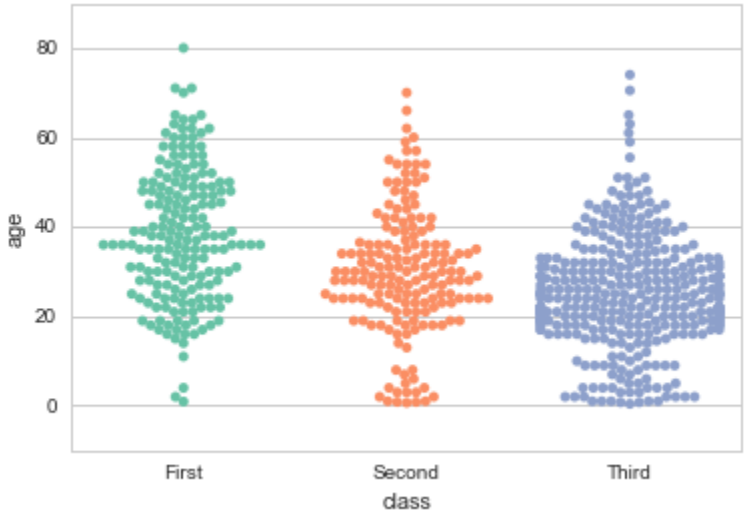
Out[45]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f23da90>
```



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [46]:
```
sns.swarmplot(x='class',y='age',data=titanic,palette='Set2')
```

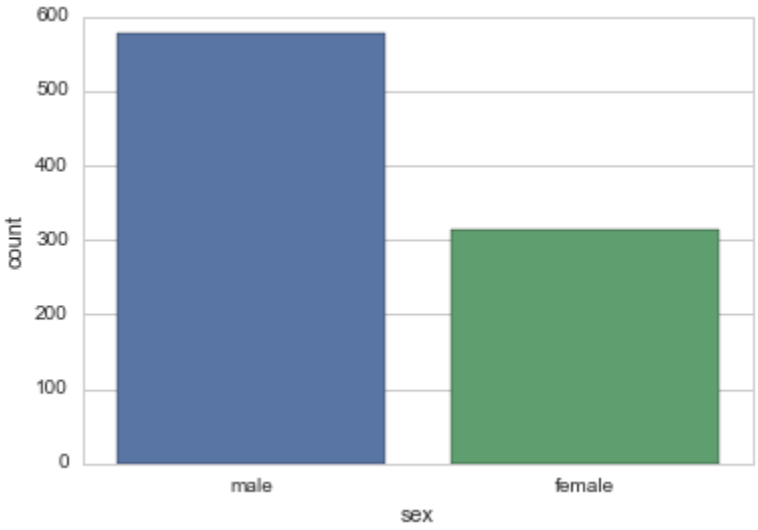
Out[46]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f215320>
```



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [47]:
```
sns.countplot(x='sex',data=titanic)
```

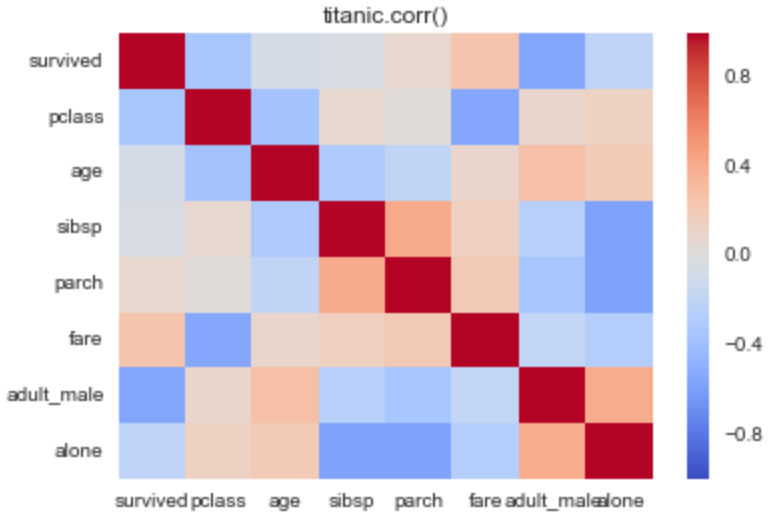
Out[47]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f207ef0>
```



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [48]:
```
sns.heatmap(titanic.corr(),cmap='coolwarm')
plt.title('titanic.corr()')
```

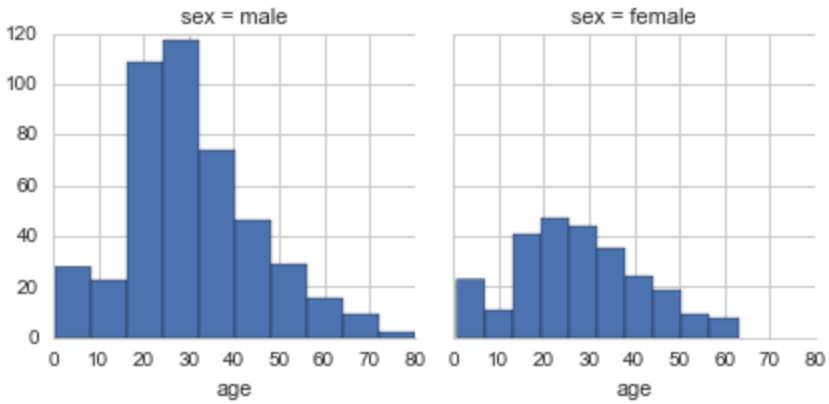
Out[48]: <matplotlib.text.Text at 0x11d72da58>



In [ ]:
```
# CODE HERE
# REPLICATE EXERCISE PLOT IMAGE BELOW
# BE CAREFUL NOT TO OVERWRITE CELL BELOW
# THAT WOULD REMOVE THE EXERCISE PLOT IMAGE!
```

In [49]:
```
g = sns.FacetGrid(data=titanic,col='sex')
g.map(plt.hist,'age')
```

Out[49]: <seaborn.axisgrid.FacetGrid at 0x11d81c240>



Great Job!

That is it for now! We'll see a lot more of seaborn practice problems in the machine learning section!