Disclaimer:

- *This document is prepared using jupyter notebook, outputs are printed out on the screen automatically without using a print() function. If you were using Pycharm to study this material, remember to use print().*
- *Most of the definitions here are taken directly from [scipy.org (https://docs.scipy.org/)](https://docs.scipy.org/) , please feel free to refer to this source for more examples if you find any of these explanations hard to understand.*
- *Each function given here will not utilize all the parameters it has. Only the base-cases are illustrated. If you want a function to do more than what is shown here, please refer to the above link to see all possible configurations.*

In [5]:
```
import numpy as np
```

Numpy is the building block of other frequently used data analysis libraries such as Pandas and Matplotlib. While we rarely use numpy for data wrangling and analysis, understanding the core concepts of numpy is highly beneficial. The 4 most important concepts are: Arrays (vectors, matrices) and number generation.

Native Python data type (e.g list) generally operates 10 to 100 times slower than Numpy array (ndarray). Therefore, algorithms frequently used for data demanding tasks such as machine learning are often written using Numpy.

Whenever you see NumPy array, ndarray or array in this document, they all mean the same thing.

Numpy Arrays

A NumPy array by definition should hold only 1 type of data. If you put more types into this array, there will be no error, instead this numpy array will work similar to a Python list.

Numpy arrays usually come in two forms: vectors and matrices. A vector is a 1-dimensional array and a matrix is a multi-dimensional array (e.g. 2-dimensional). Note that in terms of appearance and usage, NumPy array is very similar to Python list.

Creating numpy array

Using Python List

In [6]:
```
list1 = [1,2,3]
# print(list1)
```

In [7]:
```
np.array(list1)
```

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

In [8]:
```
matrix1 = [[1,2], [2,1]]
np.array(matrix1)
```

Out[8]:
```
array([[1, 2],
       [2, 1]])
```

Using built-in functions/methods

arange()
somewhat similar to range(), the last value won't be counted. This create a vector with evenly spaced integer values given an interval.

In [9]:
```
np.arange(0,10)
```

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

Very similar to Python range(), you can also specify the step or the space between values

In [10]:
```
np.arange(0,10,2)
```

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

zeros() and ones()

In [11]:
```
np.zeros(5)
```

Out[11]:
```
array([0., 0., 0., 0., 0.])
```

In [12]:
```
np.zeros((5,5))
```

Out[12]:
```
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 [13]:
```
np.ones(5)
```

Out[13]:
```
array([1., 1., 1., 1., 1.])
```

In [14]:
```
np.ones([3,2])
```

Out[14]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

linspace()
linspace(*start_point: end_point: num_of_points*)
linspace() will create evenly spaced values (including non-integers)

```
In [15]: np.linspace(0,5,2)
```

```
Out[15]: array([0., 5.])
```

```
In [16]: np.linspace(0,5,10)
```

```
Out[16]: array([0.        , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
                2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.
                ])
```

eye()

Return a 2-D array with ones on the diagonal and zeros elsewhere. Also called *identity matrix*, please consider taking a course in (refresh your knowledge in) **linear algebra** if you are not familiar with this term.

```
In [17]: np.eye(5)
```

```
Out[17]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 1.]])
```

Generate 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 [18]: np.random.rand(5)
```

```
Out[18]: array([0.90786068, 0.85968328, 0.64460595, 0.68535809, 0.8824119
                9])
```

```
In [19]: np.random.rand(5,5)
```

```
Out[19]: array([[0.05668443, 0.44786956, 0.07465729, 0.62584164, 0.3358745
         2],
                [0.31148919, 0.11637716, 0.59422559, 0.46372511, 0.2003902
         3],
                [0.64099892, 0.16461666, 0.99802665, 0.24501757, 0.2636202
         4],
                [0.0671434 , 0.25314822, 0.63646262, 0.66592077, 0.4960125
         6],
                [0.75392293, 0.45971332, 0.32054065, 0.01789525, 0.6548293
         ]])
```

randn()
Return a sample (or samples) from the "standard normal" (Gaussian) distribution (unlike rand() which is uniform)

```
In [20]: np.random.randn(5)
```

```
Out[20]: array([-0.55168228,  0.52344607,  1.50237886, -1.35539839, -0.8137
         2705])
```

```
In [21]: np.random.randn(5,5)
```

```
Out[21]: array([[-1.18557661,  1.05233302, -0.99887634,  0.17381758, -2.507
         5908 ],
                [ 3.17617035,  0.08936769, -0.63845657, -0.32605797,  0.501
         20836],
                [ 1.72043531, -0.71764019,  0.72242327, -1.22667615, -0.611
         01106],
                [-0.79377199,  0.14666167,  1.94254429,  0.47164733, -1.827
         09366],
                [ 0.07317495,  1.41964951, -0.33601913, -0.53844352, -1.124
         33328]])
```

randint(low, high=None, size=None)
Return random integers from low (inclusive) to high (exclusive).

```
In [22]: np.random.randint(100)
```

```
Out[22]: 62
```

```
In [23]: np.random.randint(10, 20, 5)
```

```
Out[23]: array([12, 14, 10, 14, 15])
```

Methods and Attributes of Numpy Arrays

Let's first create a few arrays before diving in

In [24]:
```
array1 = np.arange(36)
array2 = np.random.randint(0,30,10)
```

Methods

Reshape

Change the shape of your array, e.g change from 1-d to 2-d.

In [25]:
```
array1.reshape(6,6)
```

Out[25]:
```
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, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

Max and min attributes

max, min, argmax (index of max value), argmin (index of min value)

In [26]:
```
array2
```

Out[26]:
```
array([16, 13,  7,  0, 21, 27,  0,  2, 26,  8])
```

In [27]:
```
array2.max()
```

Out[27]:
```
27
```

In [28]:
```
array2.min()
```

Out[28]:
```
0
```

In [29]:
```
array2.argmax()
```

Out[29]:
```
5
```

In [30]:
```
array2.argmin()
```

Out[30]:
```
3
```

Attributes

Shape
Return the shape of the array

In [31]:
```
array1.shape #note the method reshaped() above has no impact on the actual array1
```

Out[31]:
```
(36,)
```

In [32]:
```
array2.shape
```

Out[32]:
```
(10,)
```

In [33]:
```
array1.reshape(6,6)
```

Out[33]:
```
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, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

In [34]:
```
array1.reshape(6,6).shape
```

Out[34]:
```
(6, 6)
```

In [35]:
```
array2.reshape(10,1)
```

Out[35]:
```
array([[16],
       [13],
       [ 7],
       [ 0],
       [21],
       [27],
       [ 0],
       [ 2],
       [26],
       [ 8]])
```

In [36]:
```
array2.reshape(10,1).shape
```

Out[36]:
```
(10, 1)
```

dtype
Show the datatype of an array, you will find this very useful when cleaning and learning about your data

In [37]:
```
array1.dtype
```

Out[37]:
```
dtype('int32')
```

NumPy Indexing and Selection

You will find in this section there are a lot of similarities between indexing and selection of numpy array and a normal Python list

In [38]:
```
array1 = np.arange(10,21)
array1
```

Out[38]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

Indexing and Selection with Bracket

In [39]:
```
#Select a specific element using index number
array1[6]
```

Out[39]:
```
16
```

In [40]:
```
#Select values in a specific range of index
array1[1:6]
```

Out[40]:
```
array([11, 12, 13, 14, 15])
```

In [41]:
```
#Another example
array1[:8]
```

Out[41]:
```
array([10, 11, 12, 13, 14, 15, 16, 17])
```

Broadcasting

Unlike Python list, where you are not allowed to reassign value to a slice, elements in numpy array can be reassigned in bulk

In [42]:
```
array1
```

Out[42]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

In [43]:
```
array1[0:5] = 999
array1
```

Out[43]:
```
array([999, 999, 999, 999, 999,  15,  16,  17,  18,  19,  20])
```

Copy an array

Similar to list , numpy array is mutable, thus, make sure you make an actual copy of an array and not *referencing* them

In [44]:
```
array1 = np.arange(10,21)
array1
```

Out[44]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

In [45]:
```
slice_array1 = array1[0:6]
slice_array1
```

Out[45]:
```
array([10, 11, 12, 13, 14, 15])
```

In [46]:
```
slice_array1[:] = 999
slice_array1
```

Out[46]:
```
array([999, 999, 999, 999, 999, 999])
```

In [47]:
```
array1 #notice that array1 also changed
```

Out[47]:
```
array([999, 999, 999, 999, 999, 999,  16,  17,  18,  19,  20])
```

In [48]:
```
#To properly copy an array, you cannot use the slicing method simila
r to python list,
# there's a special method in numpy for this purpose.
array2 = array1.copy()
array2
```

Out[48]:
```
array([999, 999, 999, 999, 999, 999,  16,  17,  18,  19,  20])
```

In [49]:
```
array2[0:6] = np.arange(10,16)
array2
```

Out[49]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

In [50]:
```
array1
```

Out[50]:
```
array([999, 999, 999, 999, 999, 999,  16,  17,  18,  19,  20])
```

Working with 2D Array (table)

Generally, there are two approach: (i) **table1[row][column]** or (ii) **table1[row, column]**. I recommend using the later. However, at the beginning, using the former will make things a bit clearer for you. The choice is yours.

Let's first create a 2d array :

In [51]:
```
table1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
table1
```

Out[51]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Common indexing and selection for 2d array

In [52]:
```
#Get a row, index of row also starts from 0
table1[1]
```

Out[52]:
```
array([4, 5, 6])
```

In [53]:
```
#Get a column
table1[:,1]
```

Out[53]:
```
array([2, 5, 8])
```

In [54]:
```
#Get a specific cell
table1[2,2]
```

Out[54]:
```
9
```

In [55]:
```
#Get a specific region
table1[:2,1:]
```

Out[55]:
```
array([[2, 3],
       [5, 6]])
```

In [56]:
```
#Very similar to Python list we also have slicing rule [start:stop:s
tep]. Step can also be negative.
table1[::-1]
```

Out[56]:
```
array([[7, 8, 9],
       [4, 5, 6],
       [1, 2, 3]])
```

Looks simple enough so far, how about 3d array or 4d array? Try it yourself

Fancy indexing

Fancy indexing allows you to select any rows or columns irrespective of their orders.
Suppose we have a 2d array named X. Fancy indexing follow this format: **X[[row],[column]]**
Let us first create a bigger 2d array

In [57]:
```
table2 = np.arange(0,100).reshape((10,10))
table2
```

Out[57]:
```
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, 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, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

In [58]:
```
#To select rows
table2[[1,3,5,7]]
```

Out[58]:
```
array([[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79]])
```

In [59]:
```
#To select a few cells. Note that in this case the cell are at [1,
1], [2,2], [3,3] and [8,2]
table2[[1,2,3,8],[1,2,3,2]]
```

Out[59]:
```
array([11, 22, 33, 82])
```

In [60]:
```
#If you want to select different columns
table2[:,[1,7,3]]
```

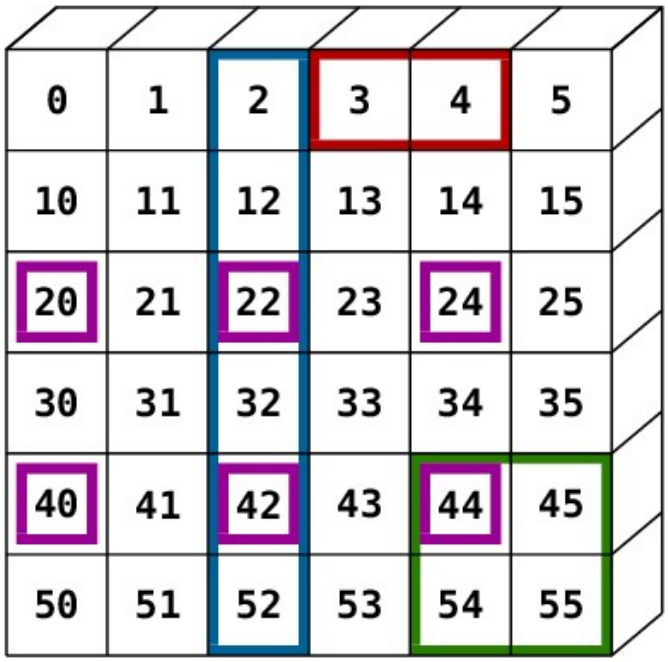
Out[60]:
```
array([[ 1,  7,  3],
       [11, 17, 13],
       [21, 27, 23],
       [31, 37, 33],
       [41, 47, 43],
       [51, 57, 53],
       [61, 67, 63],
       [71, 77, 73],
       [81, 87, 83],
       [91, 97, 93]])
```

Unlike slicing, fancy indexing always return a copy of data! And it will always return 1 dimension array (without using slicing (:))

More on numpy indexing help

The image below summarize the key points in numpy common indexing.

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



Conditional selection

Sometimes you might want to select cells that satisfy specific condition(s)

In [61]:
```
student_score = np.random.randint(0, 10, 20)
student_score
```

Out[61]:
```
array([0, 3, 7, 1, 1, 5, 4, 8, 6, 4, 1, 8, 3, 2, 8, 5, 3, 5, 3,
3])
```

In [62]:
```
# To select those that score above 5, you can:
above_5 = student_score > 5
student_score[above_5]
```

Out[62]:
```
array([7, 8, 6, 8, 8])
```

In [63]:
```
# You can also one-liner it, this is commonly used:
student_score[student_score>5]
```

Out[63]:
```
array([7, 8, 6, 8, 8])
```

Arithmentic Operations with numpy Arrays

In [64]:
```
array1 = np.arange(0,5)
```

In [65]:
```
array1 + array1
```

Out[65]:
```
array([0, 2, 4, 6, 8])
```

In [66]:
```
array1 - array1
```

Out[66]:
```
array([0, 0, 0, 0, 0])
```

In [67]:
```
array1 * array1
```

Out[67]:
```
array([ 0,  1,  4,  9, 16])
```

In [68]:
```
array1 / array1
#When divide by zero, there will only be a warning, not an error! Wh
en 0 / 0 the result will be NaN
```

```
C:\Users\Admin\Anaconda3\lib\site-packages\ipykernel_launcher.py:
1: RuntimeWarning: invalid value encountered in true_divide
  """Entry point for launching an IPython kernel.
```

Out[68]:
```
array([nan,  1.,  1.,  1.,  1.])
```

In [69]:
```
1 / array1
#If a number is divided by zero, infinity will be returned instead
```

```
C:\Users\Admin\Anaconda3\lib\site-packages\ipykernel_launcher.py:
1: RuntimeWarning: divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
```

Out[69]:
```
array([       inf, 1.        , 0.5       , 0.33333333, 0.25
])
```

In [70]:
```
array1**3
# Similar to native Python, the double-asterisk sign is for "to the
power of"
```

Out[70]:
```
array([ 0,  1,  8, 27, 64], dtype=int32)
```

Universal Functions ufunc

ufunc mostly contains mathematic operations can you can perform in an element-by-element fashion. More operations can be found at https://docs.scipy.org/doc/numpy/reference/ufuncs.html#math-operations (https://docs.scipy.org/doc/numpy/reference/ufuncs.html#math-operations). Here, we just briefly go through some common ones.

In [71]:
```
array1 = np.arange(0,10)
# Square roots
np.sqrt(array1)
```

Out[71]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.
])
```

In [72]:
```
#Exponential
np.exp(array1)
```

Out[72]:
```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369
e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316
e+03,
       2.98095799e+03, 8.10308393e+03])
```

In [73]:
```
#Return max num, this is the same as the built in method of a numpy
array .max()
np.max(array1)
```

Out[73]:
```
9
```

In [74]:
```
#Calculate the absolute value
np.absolute(array1)
```

Out[74]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [75]:
```
#Calculate natural log
np.log(array1)
```

```
C:\Users\Admin\Anaconda3\lib\site-packages\ipykernel_launcher.py:
2: RuntimeWarning: divide by zero encountered in log
```

Out[75]:
```
array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436,
       1.60943791, 1.79175947, 1.94591015, 2.07944154, 2.1972245
8])
```

In [76]:
```
#Trigonometric sine
np.sin(np.pi/2)
```

Out[76]:
```
1.0
```

In [77]:
```
np.sin(array1)
```

Out[77]:
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568
025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.4121
1849])
```

Numpy Excercises

The excercises are ordered from easy to hard

1. Import Numpy as np
2. Create an array of 15 elements that are zeros
3. Create an array of 15 elements that are 1
4. Create an array of 15 elements that are 8
5. Create an array of integer from 1 to 100
6. Create an array of odd integer from 1 to 100
7. Create a matrix of size 5x5, which has value from 0 to 24
8. Create an identity matrix of size 5x5
9. Use numpy to generate random number between 0 and 1
10. Use numpy to generate 5x5 matrix, which has values sampled from a Gaussian distribution
11. Create a matrix of size 10x10, which has value from 0.001 to 0.1
12. Create an array of 15 elements. Each points are evenly spaced (linearly spaced) between 0 and 1.
13. You are given this matrix:

In [78]:
```
matrix1 = np.arange(50,75).reshape(5,5)
matrix1
```

Out[78]:
```
array([[50, 51, 52, 53, 54],
       [55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64],
       [65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74]])
```

In [79]:
```
#Write a selection code that produce the below matrix:
```

In [80]:
```
# Write a selection code that output the number 68 from matrix1:
```

In [81]:
```
# Write a selection code that output the following array from matrix
1:
```

In [82]:
```
# Write a selection code that output the last row in matrix1:
```

In [83]:
```
# Write a selection code that output the columns that have the follo
wing index numbers: 1, 3, 4, from matrix1
```

In [84]:
```
# Write a selection code that output the rows that have the followin
g index numbers: 2, 3, 4, from matrix1
```

In [85]:
```
# Use built-in methods of numpy array. Calculate sum of all the valu
es in matrix1
```

In [86]:
```
# Use built-in methods of numpy array. Calculate standard deviation
of all the values in matrix1
```

In [87]:
```
# Use built-in methods of numpy array. Calculate the sum of each col
umns in matrix1
```

Disclaimer:

- *This document is prepared using jupyter notebook, outputs are printed out on the screen automatically without using a print() function. If you were using Pycharm to study this material, remember to use print().*
- *Most of the definitions here are taken directly from pandas.pydata.org (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html) , please feel free to refer to this source for more examples if you find any of these explanations hard to understand.*
- *Each function given here will not utilize all the parameters it has. Only the base-cases are illustrated. If you want a function to do more than what is shown here, please refer to the above link to see all possible configurations.*

Before exploring Pandas, make sure that you have installed Pandas (https://pandas.pydata.org/) in Pycharm or your system. You might find a library called "Panda", this has nothing to do with data analysis.

Series

A Series is very similar to a numpy array (in fact it is built on top of the numpy array object). Unlike a numpy array, a Series can have axis labels. This means index can be a label instead of just a number. In fact, it can even hold any arbitrary Python Object.

In [2]:
```
#You can of course, name these libraries differently, however, avoid doing so in a team. People are used to "np" and "pd"
import numpy as np
import pandas as pd
```

Creating Series

Any Python list, dictionary or numpy array can be converted into a Series

In [17]:
```
labels = ['a','b','c']
list1 = [3,7,9]
array1 = np.array([3,7,9])
dictionary1 = {'a':3,'b':7,'c':9}
```

List

In [16]:
```
pd.Series(list1) #or pd.Series(data=list1)
```

Out[16]:
```
0    3
1    c
2    9
dtype: object
```

```
In [6]: pd.Series(data=list1, index=labels)
```

```
Out[6]: a    3
        b    7
        c    9
        dtype: int64
```

```
In [7]: pd.Series(list1, labels)
```

```
Out[7]: a    3
        b    7
        c    9
        dtype: int64
```

Numpy Array

```
In [11]: pd.Series(array1)
```

```
Out[11]: 0    3
         1    7
         2    9
         dtype: int32
```

```
In [12]: pd.Series(array1, labels)
```

```
Out[12]: a    3
         b    7
         c    9
         dtype: int32
```

Dictionary

```
In [14]: pd.Series(dictionary1)
```

```
Out[14]: a    3
         b    7
         c    9
         dtype: int64
```

Types of Data in Series

Series can hold a variety of object types. Although a Series *(I am not making a grammar mistake here)* usually hold 1 type of data, it can hold different types, in this case, it is just like a Python list. Some online materials will tell you that "Series will always contain data of the same type", this is simply not true and misleading.

In [18]:
```
list1=[1, 'a', 'b', 2, 'c']
pd.Series(list1)
```

Out[18]:
```
0    1
1    a
2    b
3    2
4    c
dtype: object
```

In [22]:
```
#Series can also even hold Python functions, however, there is usual
ly no reason to do so
pd.Series([print, sum, min, max])
```

Out[22]:
```
0    <built-in function print>
1      <built-in function sum>
2      <built-in function min>
3      <built-in function max>
dtype: object
```

Index vs Label

Series index is attached to the label itself (also called index and index name), similar to Python dictionary key-value pair. Here're some examples to illustrate this point:

In [23]:
```
series1 = pd.Series([1,2,3,4],index = ['Alex', 'Ada','Mary', 'James
'])
```

In [24]:
```
series1
```

Out[24]:
```
Alex     1
Ada      2
Mary     3
James    4
dtype: int64
```

In [27]:
```
# While there still exists an index (a position number)
series1[1]
```

Out[27]:
```
2
```

In [28]:
```
# You should use the label to retrieve your data, just like dictiona
ry
series1['Ada']
```

Out[28]:
```
2
```

In [32]:
```
series2 = pd.Series([1,3,3,4],index = ['Alan', 'Mary', 'Ada', 'James
'])
```

In [30]:
```
series2
```

Out[30]:
```
Alan     1
Ada      3
Mary     3
James    4
dtype: int64
```

In [35]:
```
# Operations are done based off index names, not index
series1 + series2
```

Out[35]:
```
Ada      5.0
Alan     NaN
Alex     NaN
James    8.0
Mary     6.0
dtype: float64
```

Dataframe

DataFrames are the concept inspired by R programming language. Dataframes in Pandas are just a bunch of Series next to each other to share the same index.

In [42]:
```
from numpy.random import randn
np.random.seed(999) #Having a seed number will ensure the same resul
t for our random function.
```

In [39]:
```
df = pd.DataFrame(randn(6,4),index='A B C D E F'.split(),columns='W
X Y Z'.split())
```

In [40]:
```
df
```

Out[40]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

Indexing and selecting data

Let's look at some common way to select and index some data

In [43]:
```
df['X']
```

Out[43]:
```
A    0.661684
B    0.300543
C    0.099630
D   -0.202223
E    0.938794
F    0.911290
Name: X, dtype: float64
```

In [44]:
```
df[['X','Y']]
```

Out[44]:

	X	Y
A	0.661684	-0.708234
B	0.300543	-0.251229
C	0.099630	-0.787672
D	-0.202223	-1.564813
E	0.938794	1.831650
F	0.911290	-1.148981

In [45]:
```
# You can also do this, but please don't do it:
df.X
```

Out[45]:
```
A    0.661684
B    0.300543
C    0.099630
D   -0.202223
E    0.938794
F    0.911290
Name: X, dtype: float64
```

In [46]:
```
# You can see that the dataframe is just a bunch of Series next to e
ach other:
type(df['X'])
```

Out[46]:
```
pandas.core.series.Series
```

Creating columns

In [62]:
```
df['new_col'] = df['X'] + df['Y']
```

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

Out[63]:

	W	X	Y	Z	new_col
A	0.847024	0.661684	-0.708234	-0.498605	-0.046549
B	0.604876	0.300543	-0.251229	0.556110	0.049314
C	0.554606	0.099630	-0.787672	0.631009	-0.688042
D	1.284490	-0.202223	-1.564813	1.833163	-1.767036
E	-0.132046	0.938794	1.831650	0.883132	2.770444
F	-0.487038	0.911290	-1.148981	-0.449367	-0.237690

Dropping columns

In [58]:
```
df.drop('new_col',axis=1)
```

Out[58]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

In [53]:
```
# This is actually not done on df, you have to specify it
df
```

Out[53]:

	W	X	Y	Z	new_col
A	0.847024	0.661684	-0.708234	-0.498605	-0.046549
B	0.604876	0.300543	-0.251229	0.556110	0.049314
C	0.554606	0.099630	-0.787672	0.631009	-0.688042
D	1.284490	-0.202223	-1.564813	1.833163	-1.767036
E	-0.132046	0.938794	1.831650	0.883132	2.770444
F	-0.487038	0.911290	-1.148981	-0.449367	-0.237690

In [54]:
```
df.drop('new_col', axis=1, inplace=True)
```

In [55]:
```
df
```

Out[55]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

In [64]:
```
#You can also reassign df to a new version of itself
df = df.drop('new_col', axis=1)
```

In [65]:
```
df
```

Out[65]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

Dropping rows

In [67]:
```
df.drop('D',axis=0) #Again this is not inplace
```

Out[67]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

In [68]:
```
df
```

Out[68]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

Selecting rows

In [70]:
```
df.loc['C']
```

Out[70]:
```
W    0.554606
X    0.099630
Y   -0.787672
Z    0.631009
Name: C, dtype: float64
```

In [72]:
```
#You can also select rows by its index (iloc is for index location)
df.iloc[2]
```

Out[72]:
```
W    0.554606
X    0.099630
Y   -0.787672
Z    0.631009
Name: C, dtype: float64
```

Selecting cell, or subset of cells

In [73]:
```
df.loc['A','W']
```

Out[73]:
```
0.847024067573893
```

In [74]:
```
df.loc[['A','B'],['W','X']]
```

Out[74]:

	W	X
A	0.847024	0.661684
B	0.604876	0.300543

In [75]:
```
df.iloc[0,0]
```

Out[75]:
```
0.847024067573893
```

In [76]:
```
df.iloc[[0,1],[0,1]]
```

Out[76]:

	W	X
A	0.847024	0.661684
B	0.604876	0.300543

Conditional Selection

Conditional selection is very similar to numpy

In [86]:
```
df
```

Out[86]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

In [87]:
```
df>0
```

Out[87]:

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

In [88]:
```
df[df>0]
```

Out[88]:

	W	X	Y	Z
A	0.847024	0.661684	NaN	NaN
B	0.604876	0.300543	NaN	0.556110
C	0.554606	0.099630	NaN	0.631009
D	1.284490	NaN	NaN	1.833163
E	NaN	0.938794	1.83165	0.883132
F	NaN	0.911290	NaN	NaN

In [89]:
```
df[df['Y']>0]
```

Out[89]:

	W	X	Y	Z
E	-0.132046	0.938794	1.83165	0.883132

In [90]:
```
df[df['X']>0]['Z']
```

Out[90]:
```
A   -0.498605
B    0.556110
C    0.631009
E    0.883132
F   -0.449367
Name: Z, dtype: float64
```

In [91]:
```
df[df['X']>0][['Y','Z']]
```

Out[91]:

	Y	Z
A	-0.708234	-0.498605
B	-0.251229	0.556110
C	-0.787672	0.631009
E	1.831650	0.883132
F	-1.148981	-0.449367

In [95]:
```
# For multiple conditions you can use & (for the Python "and") and |
(for the Python "or")
df[(df['Y']>1) & (df['X'] > 0)]
```

Out[95]:

	W	X	Y	Z
E	-0.132046	0.938794	1.83165	0.883132

More about indexing

Sometimes we want to reset the index, or rename it

In [96]:
```
df
```

Out[96]:

	W	X	Y	Z
A	0.847024	0.661684	-0.708234	-0.498605
B	0.604876	0.300543	-0.251229	0.556110
C	0.554606	0.099630	-0.787672	0.631009
D	1.284490	-0.202223	-1.564813	1.833163
E	-0.132046	0.938794	1.831650	0.883132
F	-0.487038	0.911290	-1.148981	-0.449367

In [98]:
```
df.reset_index()
# Again this is not done inplace
# The index column is not the index, but the old index (however, it's named "index"). Might be confusing here!
```

Out[98]:

	index	W	X	Y	Z
0	A	0.847024	0.661684	-0.708234	-0.498605
1	B	0.604876	0.300543	-0.251229	0.556110
2	C	0.554606	0.099630	-0.787672	0.631009
3	D	1.284490	-0.202223	-1.564813	1.833163
4	E	-0.132046	0.938794	1.831650	0.883132
5	F	-0.487038	0.911290	-1.148981	-0.449367

In [100]:
```
stud_names= "Ada Adam Marley Rob Morgan Alan".split(" ")
df['student_name'] = stud_names
```

In [101]:
```
df
```

Out[101]:

	W	X	Y	Z	student_name
A	0.847024	0.661684	-0.708234	-0.498605	Ada
B	0.604876	0.300543	-0.251229	0.556110	Adam
C	0.554606	0.099630	-0.787672	0.631009	Marley
D	1.284490	-0.202223	-1.564813	1.833163	Rob
E	-0.132046	0.938794	1.831650	0.883132	Morgan
F	-0.487038	0.911290	-1.148981	-0.449367	Alan

In [102]:
```
df.set_index('student_name') #this is also not done inplace
```

Out[102]:

	W	X	Y	Z
student_name				
Ada	0.847024	0.661684	-0.708234	-0.498605
Adam	0.604876	0.300543	-0.251229	0.556110
Marley	0.554606	0.099630	-0.787672	0.631009
Rob	1.284490	-0.202223	-1.564813	1.833163
Morgan	-0.132046	0.938794	1.831650	0.883132
Alan	-0.487038	0.911290	-1.148981	-0.449367

In [103]:
```
df.set_index('student_name', inplace=True)
```

In [104]:
```
df
```

Out[104]:

student_name	W	X	Y	Z
Ada	0.847024	0.661684	-0.708234	-0.498605
Adam	0.604876	0.300543	-0.251229	0.556110
Marley	0.554606	0.099630	-0.787672	0.631009
Rob	1.284490	-0.202223	-1.564813	1.833163
Morgan	-0.132046	0.938794	1.831650	0.883132
Alan	-0.487038	0.911290	-1.148981	-0.449367

Multi-Index and Index Hierarchy

In economics, you will often see this in clustered data or experimental data (factorial design).
If this looks confusing, you can always use dummy variable instead of methods listed in this section.

In [13]:
```
# Index Levels
outter_index = "G1 G1 G1 G2 G2 G2".split()
inner_index = [1,2,3,1,2,3]
hier_index = list(zip(outter_index,inner_index)) #zip just put eleme
nts in pair in a tuple
```

In [14]:
```
hier_index
```

Out[14]:
```
[('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2), ('G2', 3)]
```

In [15]:
```
hier_index = pd.MultiIndex.from_tuples(hier_index)
```

In [16]:
```
hier_index
```

Out[16]:
```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           )
```

In [20]:
```
#You might see something like this (depending on version). This is e
ssentially the same as the above.
```

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

In [108]:
```
df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
```

Out[108]:

		A	B
G1	1	-1.110061	-0.584822
	2	-0.188410	0.813024
	3	-0.161305	1.600872
G2	1	0.984343	-0.835447
	2	-0.186649	-0.858067
	3	-0.759778	-1.512054

Indexing and selecting data from this multi-index dataframe is very similar with the normal dataframe discussed above

In [111]:
```
# Select outter rows
df.loc['G1']
```

Out[111]:

	A	B
1	-1.110061	-0.584822
2	-0.188410	0.813024
3	-0.161305	1.600872

In [115]:
```
#Select outter rows and inner rows
group1 = df.loc['G1']
group1.loc[1]
```

Out[115]:
```
A   -1.110061
B   -0.584822
Name: 1, dtype: float64
```

In [116]:
```
#Or a faster way
df.loc['G1'].loc[1]
```

Out[116]:
```
A   -1.110061
B   -0.584822
Name: 1, dtype: float64
```

In [117]:
```
#The indexes do not actually have names
df.index.names
```

Out[117]:
```
FrozenList([None, None])
```

In [118]:
```
#You can give them names by:
df.index.names = ['Group','Num']
```

In [119]:
```
df
```

Out[119]:

		A	B
Group	**Num**		
G1	1	-1.110061	-0.584822
	2	-0.188410	0.813024
	3	-0.161305	1.600872
G2	1	0.984343	-0.835447
	2	-0.186649	-0.858067
	3	-0.759778	-1.512054

In [120]:
```
#For a named multi-index data like this, you can select data using index names via method .xs()
df.xs('G1')
```

Out[120]:

	A	B
Num		
1	-1.110061	-0.584822
2	-0.188410	0.813024
3	-0.161305	1.600872

In [121]:
```
df.xs(['G1',1])
```

Out[121]:
```
A   -1.110061
B   -0.584822
Name: (G1, 1), dtype: float64
```

In [122]:
```
df.xs(1,level='Num') #You can specify the level if index name that you are looking for
```

Out[122]:

	A	B
Group		
G1	-1.110061	-0.584822
G2	0.984343	-0.835447

Dealing with missing data

Depending on your situation and purposes, missing data can convey a meaning or can be converted into a meaningful value. For example, if you have a pile of graded exams, a few students didn't show up, thus their grades are missing. These students' grades are recorded as missing or `nan` when you type the data into a spreadsheet. You can then, turn these missing entries into 5.0 (a failing grade) or keep them as `nan` to remind yourself that these were the missing exams.

You can also drop all of these students with missing exams out of your data. However, doing so means losing potential valuable data.

This section will provide you with some basic ways to handle missing data.

In [158]:
```
df = pd.DataFrame(
    { 'Biology':[5.0,2,np.nan],
      'Math':[1,np.nan,np.nan],
      'Physics':[1.0,2.0,3.0],
      'Economics':[1.3,2.0,2.0] }
)
```

In [159]:
```
df
```

Out[159]:

	Biology	Math	Physics	Economics
0	5.0	1.0	1.0	1.3
1	2.0	NaN	2.0	2.0
2	NaN	NaN	3.0	2.0

In [160]:
```
df.dropna() #Drop any rows that has any missing value
```

Out[160]:

	Biology	Math	Physics	Economics
0	5.0	1.0	1.0	1.3

In [161]:
```
df.dropna(axis=1)
```

Out[161]:

	Physics	Economics
0	1.0	1.3
1	2.0	2.0
2	3.0	2.0

In [166]:
```
df.dropna(thresh=3) #"thresh = N" define a threshold, a row has to h
ave at least N amount of data in order to survive
```

Out[166]:

	Biology	Math	Physics	Economics
0	5.0	1.0	1.0	1.3
1	2.0	NaN	2.0	2.0

In [168]:
```
df.fillna(value='') #Fill NaN with an empty string
```

Out[168]:

	Biology	Math	Physics	Economics
0	5	1	1.0	1.3
1	2		2.0	2.0
2			3.0	2.0

In [170]:
```
df['Biology'].fillna(value=df['Biology'].mean())
```

Out[170]:
```
0    5.0
1    2.0
2    3.5
Name: Biology, dtype: float64
```

Groupby

Groupby helps you aggregate/melt data or group rows of data together

In [172]:
```
data = {'Company':['Google','Google','Amazon','Amazon','Facebook','Facebook'],
        'Person':['Tom','Carl','Ada','Alan','Cornie','Oliver'],
        'Sales':[220,150,330,129,233,370]}
```

In [174]:
```
df = pd.DataFrame(data)
df
```

Out[174]:

	Company	Person	Sales
0	Google	Tom	220
1	Google	Carl	150
2	Amazon	Ada	330
3	Amazon	Alan	129
4	Facebook	Cornie	233
5	Facebook	Oliver	370

You can try to group the rows together based on a column, in this case, we might be interested in the sales of each company

In [175]:
```
df.groupby('Company')
```

Out[175]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000024EC227A4C8>
```

Pandas return an object instead of a dataframe table because we did not specify what to do with the data, if you group by company, what about the Person? or what to do with sales?

In [177]:
```
# Finding out total sales
df.groupby('Company').sum()
```

Out[177]:

Company	Sales
Amazon	459
Facebook	603
Google	370

In [178]:
```
#Finding out average sales
df.groupby('Company').mean()
```

Out[178]:

Company	Sales
Amazon	229.5
Facebook	301.5
Google	185.0

In [179]:
```
df.groupby('Company').min()
```

Out[179]:

Company	Person	Sales
Amazon	Ada	129
Facebook	Cornie	233
Google	Carl	150

In [180]:
```
df.groupby('Company').max()
```

Out[180]:

Company	Person	Sales
Amazon	Alan	330
Facebook	Oliver	370
Google	Tom	220

In [181]:
```
df.groupby('Company').std()
```

Out[181]:

Company	Sales
Amazon	142.128463
Facebook	96.873629
Google	49.497475

In [182]:
```
df.groupby('Company').count()
```

Out[182]:

Company	Person	Sales
Amazon	2	2
Facebook	2	2
Google	2	2

In [183]:
```
df.groupby('Company').describe()
```

Out[183]:

	Sales							
	count	mean	std	min	25%	50%	75%	max
Company								
Amazon	2.0	229.5	142.128463	129.0	179.25	229.5	279.75	330.0
Facebook	2.0	301.5	96.873629	233.0	267.25	301.5	335.75	370.0
Google	2.0	185.0	49.497475	150.0	167.50	185.0	202.50	220.0

In [184]:
```
df.groupby('Company').describe().transpose()
```

Out[184]:

	Company	Amazon	Facebook	Google
Sales	count	2.000000	2.000000	2.000000
	mean	229.500000	301.500000	185.000000
	std	142.128463	96.873629	49.497475
	min	129.000000	233.000000	150.000000
	25%	179.250000	267.250000	167.500000
	50%	229.500000	301.500000	185.000000
	75%	279.750000	335.750000	202.500000
	max	330.000000	370.000000	220.000000

In [185]:
```
df.groupby('Company').describe().transpose()['Facebook']
```

Out[185]:
```
Sales  count      2.000000
       mean     301.500000
       std       96.873629
       min      233.000000
       25%      267.250000
       50%      301.500000
       75%      335.750000
       max      370.000000
Name: Facebook, dtype: float64
```

Merge, Join and Concatenate

In [186]:
```
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])
df1
```

Out[186]:

	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 [187]:
```
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])
df2
```

Out[187]:

	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 [188]:
```
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])
df3
```

Out[188]:

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

Concatenating

Concatenation basically sticks DataFrames together. Keep in mind that the dimensions should match.

In [190]:
```
pd.concat([df1,df2,df3])
```

Out[190]:

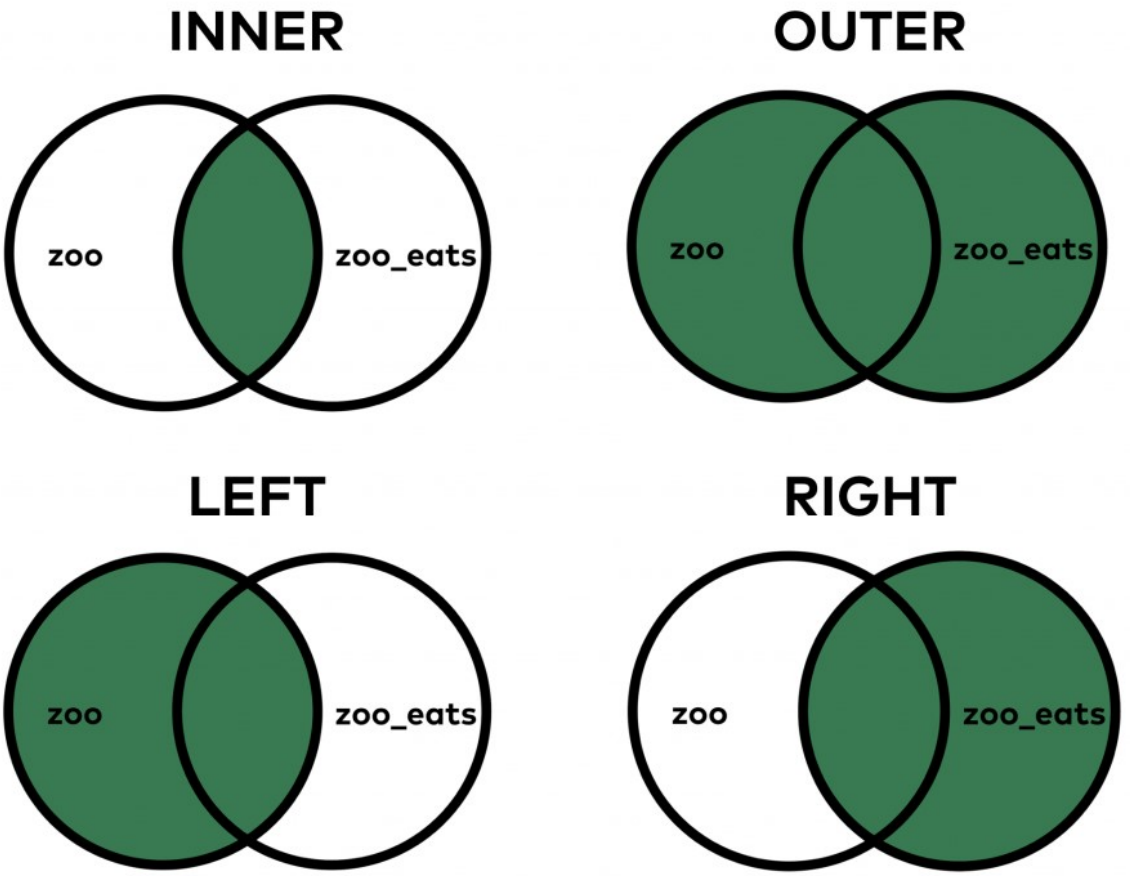
	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

In [191]:
```
pd.concat([df1,df2,df3],axis=1) #In this case it doesn't make sense, but sometimes you want to concat by columns
```

Out[191]:

	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

Merging



In [210]:
```
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 [194]:
```
left
```

Out[194]:

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

In [195]:
```
right
```

Out[195]:

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

In [211]:
```
#Often you want to merge data using a unique identification number.
In this case, key is that number
pd.merge(left,right,how='inner',on='key') #default is inner join
```

Out[211]:

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

In [197]:
```
#Sometimes you want to merge on multiple unique keys
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 [198]:
```
left
```

Out[198]:

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

In [199]:
```
right
```

Out[199]:

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

In [200]:
```
pd.merge(left, right, on=['key1', 'key2'])
```

Out[200]:

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

In [201]:
```
pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

Out[201]:

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

In [202]:
```
pd.merge(left, right, how='right', on=['key1', 'key2'])
```

Out[202]:

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

In [203]:
```
pd.merge(left, right, how='left', on=['key1', 'key2'])
```

Out[203]:

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

Joining

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

Usually, you will want to use merge() than join(). The same result of join can always be achieved using merge()

In [212]:
```
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 [205]:
```
left
```

Out[205]:

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

In [206]:
```
right
```

Out[206]:

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

In [213]:
```
left.join(right) #default is left join
```

Out[213]:

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

In [214]:
```
left.join(right, how='outer')
```

Out[214]:

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

Useful Operations

These are just collection of useful operations that I think will be frequently used. There is no particular categories in this section.

In [220]:
```
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[333,222,777,333],'col3
':['abc','def','ghi','xyz']})
df.head(2)
```

Out[220]:

	col1	col2	col3
0	1	333	abc
1	2	222	def

In [221]:
```
df['col2'].unique()
```

Out[221]:
```
array([333, 222, 777], dtype=int64)
```

In [222]:
```
df['col2'].nunique()
```

Out[222]:
```
3
```

In [223]:
```
df['col2'].value_counts()
```

Out[223]:
```
333    2
777    1
222    1
Name: col2, dtype: int64
```

In [227]:
```
#Select from DataFrame using criteria from multiple columns
df_new = df[(df['col1']<3) & (df['col2']==222)]
```

In [228]:
```
df_new
```

Out[228]:

	col1	col2	col3
1	2	222	def

In [231]:
```
def square_cell(x):
    return x*2
```

In [232]:
```
df['col1'].apply(square_cell)
```

Out[232]:
```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

In [233]:
```
df['col3'].apply(len)
```

Out[233]:
```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

In [234]:
```
df['col1'].sum()
```

Out[234]:
```
10
```

In [235]:
```
del df['col1']
```

In [236]:
```
df
```

Out[236]:

	col2	col3
0	333	abc
1	222	def
2	777	ghi
3	333	xyz

In [237]:
```
df.columns #Get column names
```

Out[237]:
```
Index(['col2', 'col3'], dtype='object')
```

In [238]:
```
df.index #Get index numbering/names
```

Out[238]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [239]:
```
df.sort_values(by='col2') #inplace is False by default
```

Out[239]:

	col2	col3
1	222	def
0	333	abc
3	333	xyz
2	777	ghi

In [240]:
```
df.isnull() #check if something is null or NaN
```

Out[240]:

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

In [245]:
```
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 [242]:
```
df
```

Out[242]:

	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 [246]:
```
#Pivot table is a powerful tool to transform your data
df.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

Out[246]:

		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

Data Input and Output

CSV

In [251]:
```
# Read in a file named "example" from the folder of your .py file. Y
ou can use relative path of course
df = pd.read_csv('example.csv')
df
```

Out[251]:

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

In [252]:
```
# Output a file named "example", ignore index
df.to_csv('example.csv',index=False)
```

Excel

Pandas can read and write excel files, keep in mind, this only imports simple data. Images or macros may cause this read_excel method to crash.

In [254]:
```
pd.read_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
```

Out[254]:

	Unnamed: 0	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

In [255]:
```
df.to_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
```

HTML

Pandas offers you an easy way to read table off a website. If an error occurs that requires you to install some libraries, just install them. (Libraries such as htmllib5,lxml, and BeautifulSoup4)

In [13]:
```
df = pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')
```

In [14]:
```
df2 = df[0]
df2
```

Out[14]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date
0	City National Bank of New Jersey	Newark	NJ	21111	Industrial Bank	November 1, 2019	November 7, 2019
1	Resolute Bank	Maumee	OH	58317	Buckeye State Bank	October 25, 2019	November 12, 2019
2	Louisa Community Bank	Louisa	KY	58112	Kentucky Farmers Bank Corporation	October 25, 2019	November 7, 2019
3	The Enloe State Bank	Cooper	TX	10716	Legend Bank, N. A.	May 31, 2019	August 22, 2019
4	Washington Federal Bank for Savings	Chicago	IL	30570	Royal Savings Bank	December 15, 2017	July 24, 2019
...	...	...	...	...	...	...	...
554	Superior Bank, FSB	Hinsdale	IL	32646	Superior Federal, FSB	July 27, 2001	August 19, 2014
555	Malta National Bank	Malta	OH	6629	North Valley Bank	May 3, 2001	November 18, 2002
556	First Alliance Bank & Trust Co.	Manchester	NH	34264	Southern New Hampshire Bank & Trust	February 2, 2001	February 18, 2003
557	National State Bank of Metropolis	Metropolis	IL	3815	Banterra Bank of Marion	December 14, 2000	March 17, 2005
558	Bank of Honolulu	Honolulu	HI	21029	Bank of the Orient	October 13, 2000	March 17, 2005

559 rows × 7 columns

Exercises

1. Import numpy and pandas
2. Import salaries.csv file and named it as sal. This is taken from https://www.kaggle.com/kaggle/sf-salaries (https://www.kaggle.com/kaggle/sf-salaries) . This is an uncleaned version of the dataset that I uploaded on moodle. If you want a challenge you can download it directly from Kaggle and clean the data yourself. This dataset comes with all the questions below. You might find solutions online. But try to do it yourself. Furthermore, one of the answers you found online could be wrong (I will show it in class).
3. Check the head of the dataframe
4. Use .info() to quickly see how many entries are there
5. Find the average BasePay
6. Find the highest amount of OvertimePay
7. What is the job title of Greg V Rebollo ? Note: Names are case sensitive
8. How much does Greg V Rebollo make (including benefits)?
9. What is the name of highest paid person (including benefits)?
10. What is the name of lowest paid person (including benefits)?
11. What was the average (mean) BasePay of all employees per year? (2011-2014) ?
12. How many unique job titles are there?
13. What are the top 5 most common jobs?
14. How many Job Titles were represented by only one person in 2013? (e.g. Job Titles with only one occurence in 2013?)
15. How many people have the word Chief in their job title?
16. Is there a correlation between length of the Job Title string and Salary?
17. How many entries are there for each year? (2011-2014)

In [ ]:

Matplotlib

Matplotlib is one of the oldest and a base library in Python for data visualization. In fact, if you want to use a more advanced data visualization tool such as Seaborn, you have to install Matplotlib.

Matplotlib will require Numpy library installed to function.

Similar to previous lecture on Numpy and Pandas, the best site for further information about Matplotlib is its official page: http://matplotlib.org/ (http://matplotlib.org/)

In [2]:
```
# We will start with importing Matplotlib
import matplotlib.pyplot as plt

# If you want to see the graph directly inside Jupyter Notebook, you
also have to use this command:
%matplotlib inline

# If you are using Pycharm or other IDE, you can use this function
(it is similar to print()), you should use it at
# the end of the .py file you are working with to avoid error (inclu
ding logical error as different plots may show up in
# one figure), in general, you are unsure, it is best to have differ
ent files/modules for different figures:
plt.show()
```

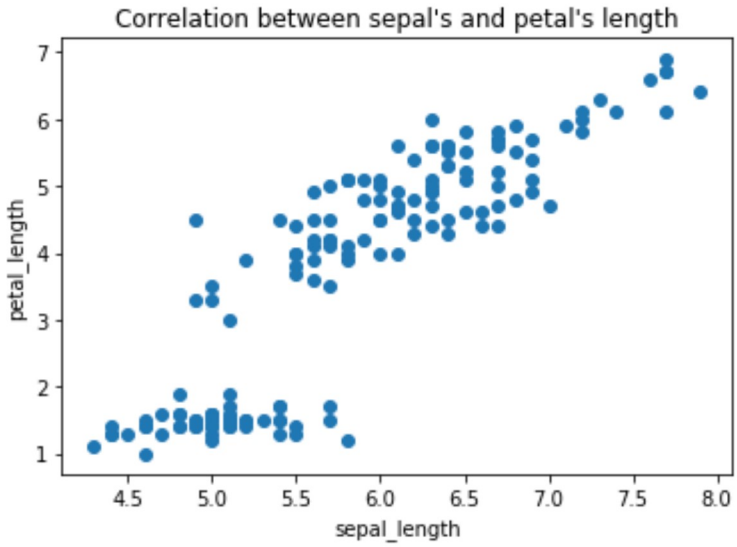
Basic Matplotlib

In [10]:
```
# Let's first create some data:
import numpy as np
import pandas as pd
import seaborn as sns

iris = sns.load_dataset('iris')
iris.head(5)
```

Out[10]:

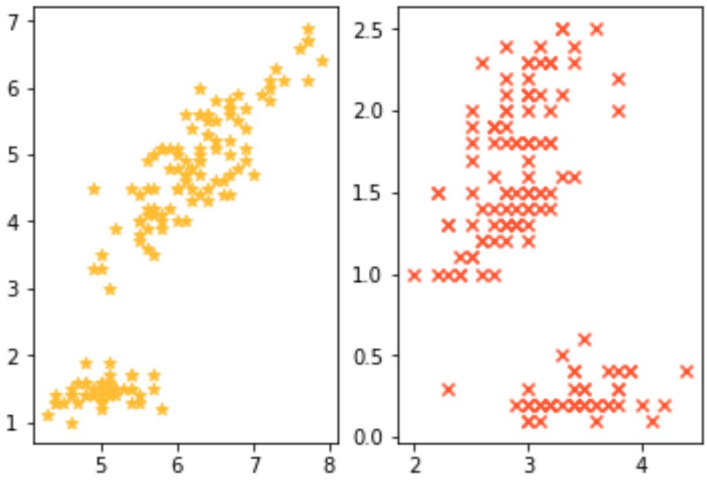
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [16]:
```
# You can create a simple line plot by:
plt.scatter(iris['sepal_length'], iris['petal_length']) # 'b' is the
color blue
plt.xlabel('sepal_length')
plt.ylabel('petal_length')
plt.title("Correlation between sepal's and petal's length")
plt.show()
```



Multiple plots in the same figure

In [24]:
```
# plt.subplot(num_rows, num_cols, plot_number)
plt.subplot(1,2,1)
plt.scatter(iris['sepal_length'], iris['petal_length'], marker='*',
c='#FFBD33')
plt.subplot(1,2,2)
plt.scatter(iris['sepal_width'], iris['petal_width'], marker='x', c=
'#FF5733');
```



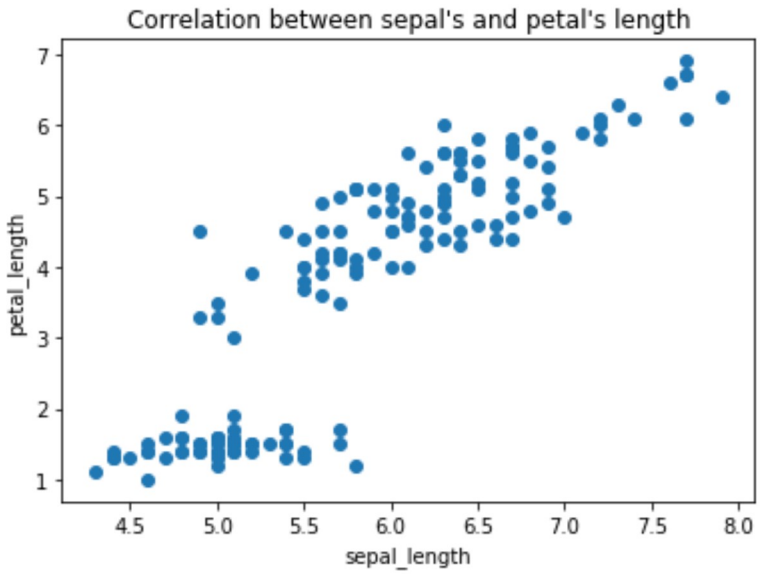
Object-oriented with Matplotlib

This is a more standard way to create graphs/figures with Matplotlib. The idea is to create an empty figure (like a blank canvas) as an object first then we draw on it using built-in methods.

Basic plot

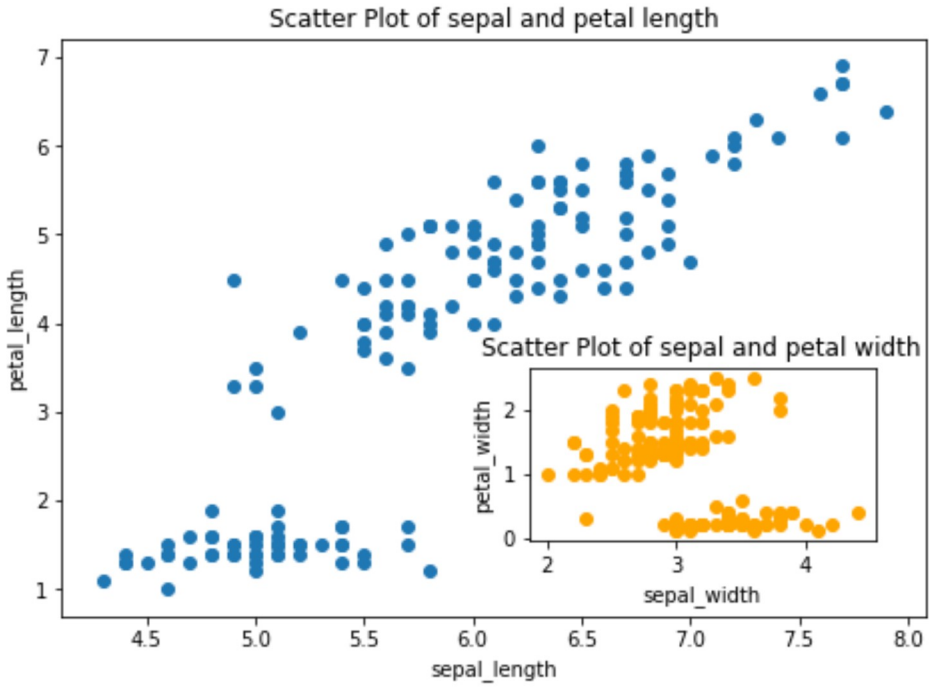
In [26]:
```
# Initialize an empty figure (blank canvas):
fig = plt.figure()

# Add dimensions to the figure (axes):
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, hei
ght (range 0 to 1)

# Now we just plot the data on those axes we've just created
axes.scatter(iris['sepal_length'], iris['petal_length'])

# If you want to add some decors, you can:
axes.set_xlabel('sepal_length') # Similar to object-oriented program
ing, remember the getters and setters?
axes.set_ylabel('petal_length')
axes.set_title("Correlation between sepal's and petal's length")
```

Out[26]:
```
Text(0.5,1,"Correlation between sepal's and petal's length")
```



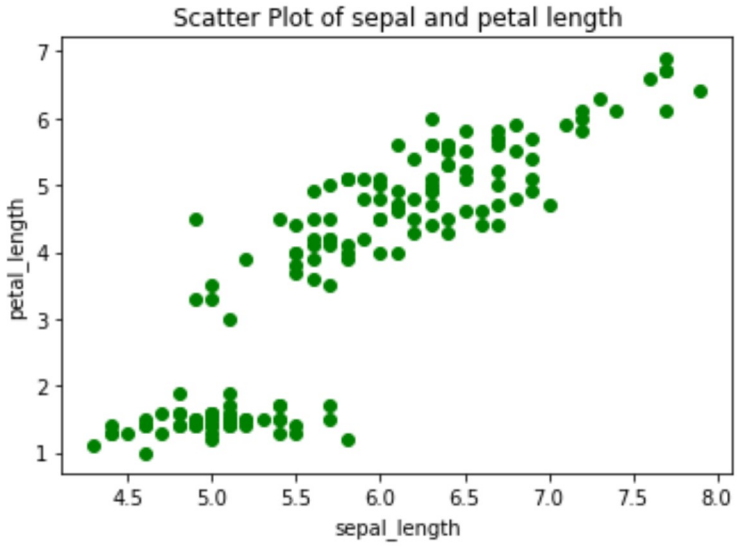
It seems to have more step to just create a simple plot like the above. But we can now have control over many elements in the figures such as the location of the axes or how many axes to create:

In [62]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1]) # 1st set of axes
axes2 = fig.add_axes([0.54, 0.13, 0.4, 0.3]) # 2nd set of axes, smal
ler axes


x = np.linspace(0, 5, 20)
y = x ** 2
# 1st set of axes
axes1.scatter(iris['sepal_length'], iris['petal_length'])
axes1.set_xlabel('sepal_length')
axes1.set_ylabel('petal_length')
axes1.set_title('Scatter Plot of sepal and petal length');

# 2nd set of axes
axes2.scatter(iris['sepal_width'], iris['petal_width'], c='orange')
axes2.set_xlabel('sepal_width')
axes2.set_ylabel('petal_width')
axes2.set_title('Scatter Plot of sepal and petal width');
```

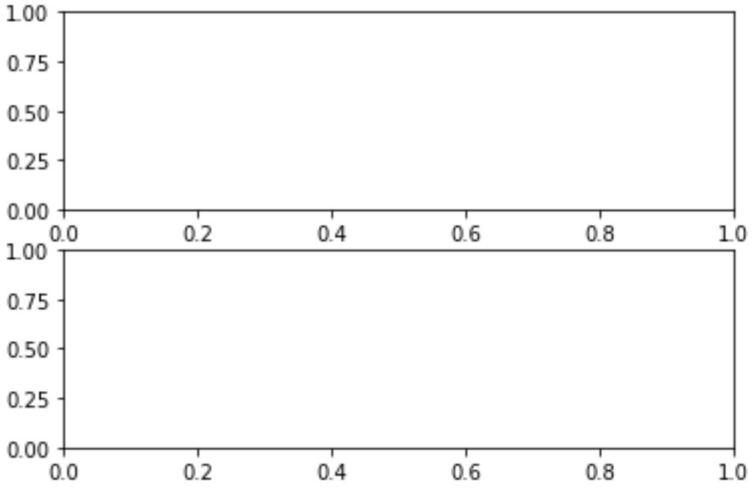


subplot()

In [63]:
```
# Firstly, you can use subplots() similar to plt.figure() except sub
plots() by default will create a tuple with the
# 1st element as figure and second element as axes on this figure. W
e grab these using 2 variables assignment:

fig, axes = plt.subplots()

# Now use the axes object to draw data on it
axes.scatter(iris['sepal_length'], iris['petal_length'], c='g')
axes.set_xlabel('sepal_length')
axes.set_ylabel('petal_length')
axes.set_title('Scatter Plot of sepal and petal length');
```



In [74]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=2, ncols=1) #You can use positional a
rguments here
```

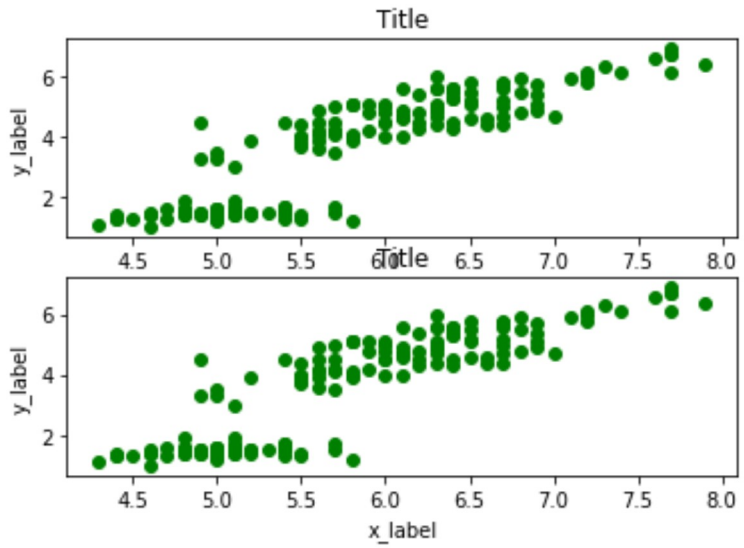


In [75]:
```
# axes is now an array with elements contain different set of axes
axes
```

Out[75]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x0000016D
5D9DB400>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x0000016D
5AC63C18>],
      dtype=object)
```

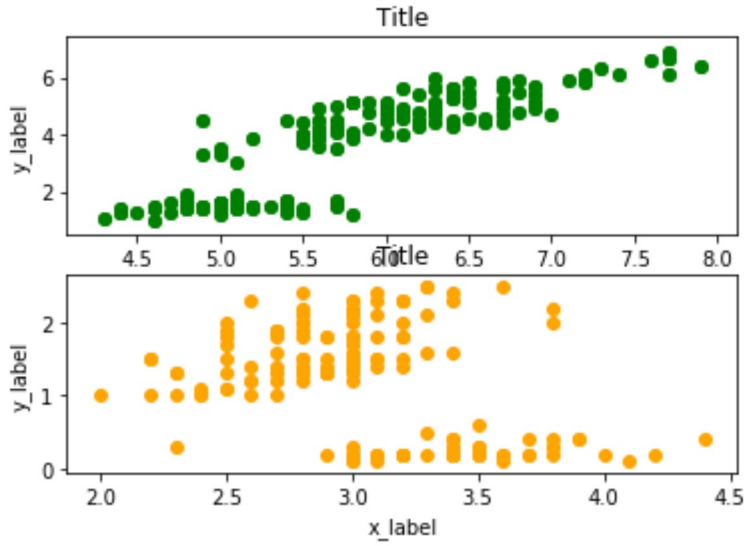
In [76]:
```
# Since axes is an array you can loop through it or access it with i
ndex number:
for ax in axes:
    ax.scatter(iris['sepal_length'], iris['petal_length'], c='g')
    ax.set_xlabel('x_label')
    ax.set_ylabel('y_label')
    ax.set_title('Title')

fig
```

Out[76]:

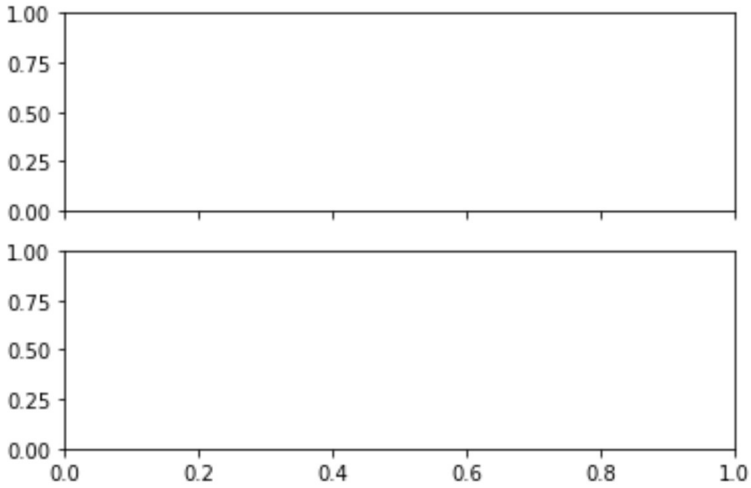


In [72]:
```
axes[1].clear() #clear the data from the last step. Otherwise, you'l
l have both green and orange dots in axes[1]
axes[1].scatter(iris['sepal_width'], iris['petal_width'], c='orange
')
axes[1].set_xlabel('x_label')
axes[1].set_ylabel('y_label')
axes[1].set_title('Title')

fig
```

Out[72]:

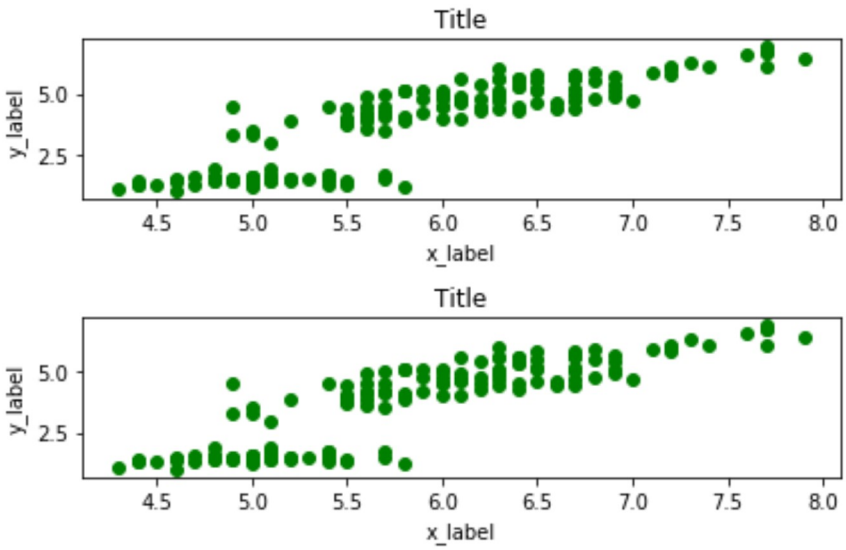


In [46]:
```
# You might also wants to get rid of the similar axes, by declaring
at the beginning:
fig, axes = plt.subplots(nrows=2, ncols=1, sharex='col', sharey='row
')
```



Instead of getting rid of the inner labels. We can 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 [73]:
```
fig, axes = plt.subplots(nrows=2, ncols=1)

for ax in axes:
    ax.scatter(iris['sepal_length'], iris['petal_length'], c='g')
    ax.set_xlabel('x_label')
    ax.set_ylabel('y_label')
    ax.set_title('Title')

fig
plt.tight_layout()
```



Size, ratio and DPI

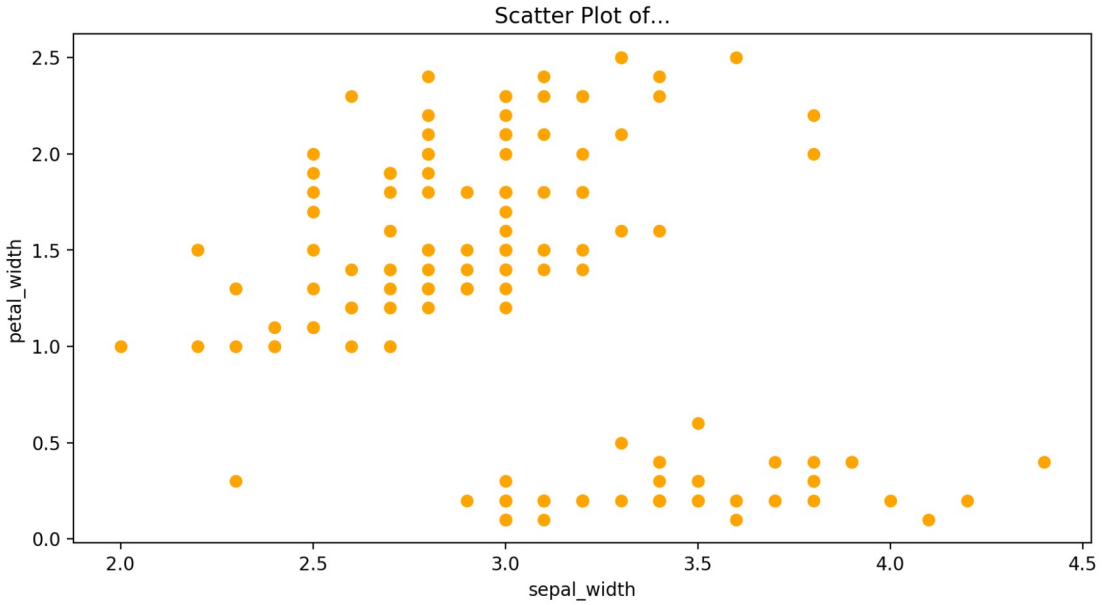
Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created. You can use the figsize and dpi keyword arguments.

- figsize is a tuple of the width and height of the figure in inches
- dpi is the dots-per-inch (pixel per inch).
- aspect ratio is the ratio between height and width of the data, but what you want is the display ratio (! This is a bit confusing, just follow the formula)

(!This makes more sense when you work with Pycharm, you won't see some features clearly in Jupyter Notebook due to auto-scaling for example)

Here's an example:

In [88]:
```
fig, axes = plt.subplots(dpi=200, figsize=(10,20))

axes.scatter(iris['sepal_width'], iris['petal_width'], c='orange')
axes.set_xlabel('sepal_width')
axes.set_ylabel('petal_width')
axes.set_title('Scatter Plot of...')
axes.set_aspect(0.5/axes.get_data_ratio())
# (ratio_you_want)/data_ratio = aspect, so if you want height = 0.5*
width, you can make a calculation:
# aspect = 0.5/axes.get_data_ratio()
```



In [97]:
```
x_lim = axes.get_xlim()
x_interval = abs(x_lim[0] - x_lim[1])
y_lim = axes.get_ylim()
y_interval = abs(y_lim[0] - y_lim[1])
data_ratio = y_interval/x_interval #this is equivalent to axes.get_d
ata_ratio()
data_ratio;
```

Out[97]:
```
0.9992161755766702
```

Saving figures

Matplotlib can save figure outputs in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF.

To save a figure, use the savefig method in the Figure class:

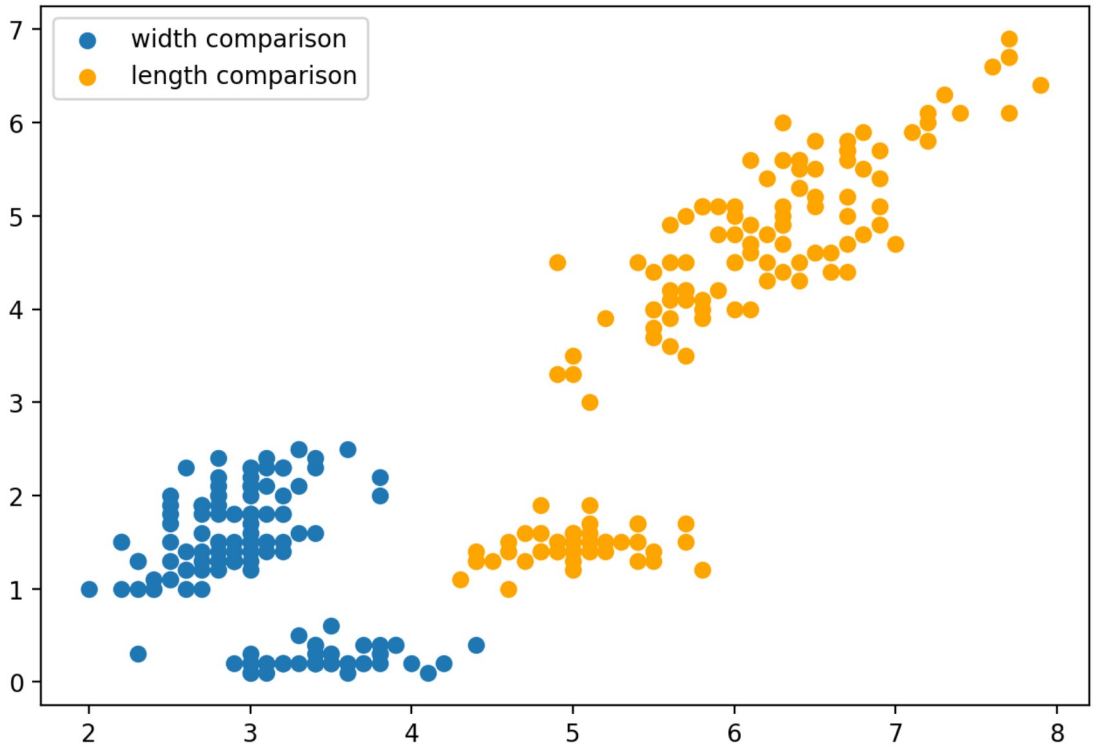
In [147]:
```
fig.savefig("filename.png", dpi=200) #specifying dpi is optional, ho
wever, it is a good practice to do so
```

Legends

You can use the label="label text" keyword argument when plotting data, and then using the .legend() method without arguments to add the legend to the figure:

In [100]:
```
fig = plt.figure(dpi=200)

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

ax.scatter(iris['sepal_width'], iris['petal_width'], label="width co
mparison")
ax.scatter(iris['sepal_length'], iris['petal_length'], label="length
comparison", c = 'orange')
ax.legend()
```

Out[100]:
```
<matplotlib.legend.Legend at 0x16d5db20908>
```



Sometimes the legend might overlap with the actual plot. You can tell Matplotlib where to put the legend by passing in the .legend() method the keyword argument loc.

This argument allows values of loc to be numerical codes for the various places the legend can be drawn. See the documentation page (https://matplotlib.org/tutorials/intermediate/legend_guide.html) for details. Some of the most common loc values are:

In [ ]:
```
ax.legend(loc=0) # let matplotlib decide the optimal location (mostly used)
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

fig
```

Setting colors, linewidths, linetypes

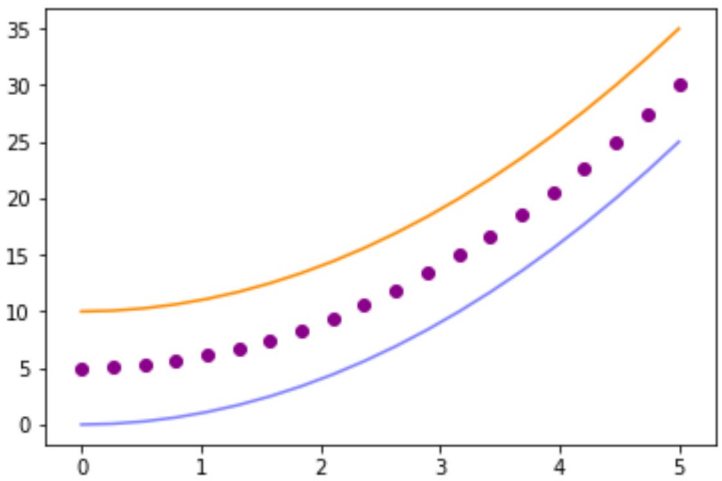
Matplotlib gives you many options to customize colors, linewidths, and linetypes.

In [ ]:
```
# Let first create some simple data so that we can see the color more clearing
x = np.linspace(0, 5, 11)
y = x ** 2
```

Color

In [106]:
```
fig, ax = plt.subplots()

ax.plot(x, y, color="blue", alpha=0.5)   # half-transparent
ax.scatter(x, y+5, color="#8B008B")        # RGB hex code
ax.plot(x, y+10, color="#FF8C00")         # RGB hex code
```

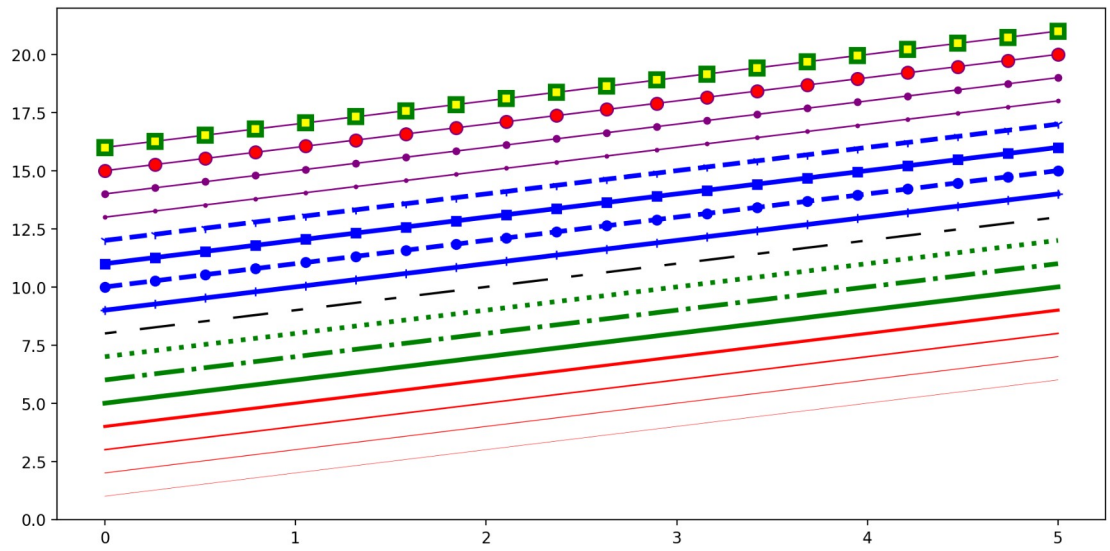
Out[106]:
```
[<matplotlib.lines.Line2D at 0x16d5aeccef0>]
```



Line style and marker style

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 [108]:
```
fig, ax = plt.subplots(figsize=(12,6), dpi=200)

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 lengt
h, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1
', '2', '3', '4', ...
ax.plot(x, x+ 9, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
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', markersiz
e=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersiz
e=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersiz
e=8, markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersiz
e=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor
="green");
```



And a lot more at: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html)

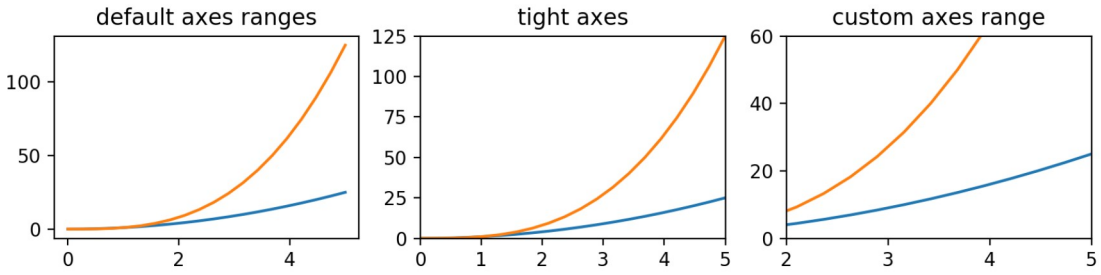
Controlling Axes

In this section we'll look at ways to control how Matplotlib draw the axes

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 [118]:
```
fig, axes = plt.subplots(1, 3, figsize=(10, 2), dpi=200)

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")
# There will be no visible differences between the first 2 axes beca
use Matplotlib recently adds padding to the graph.
# This line below will show the intended result, you might not need
this for different dataset:
axes[1].autoscale(enable=True, axis='both', tight=True)

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");
```

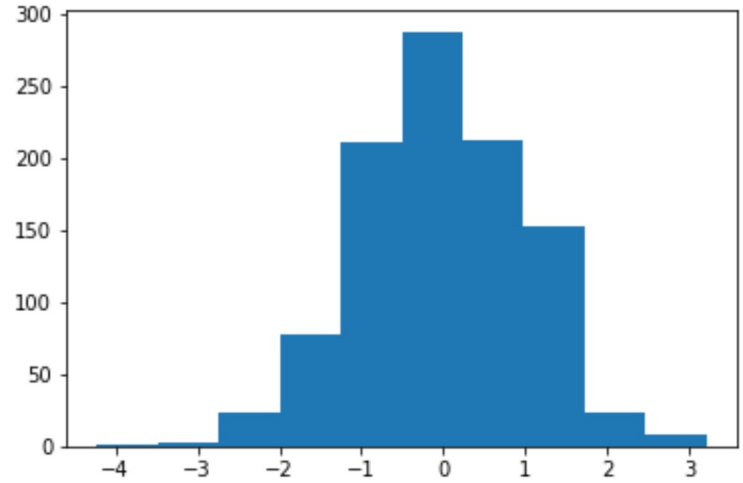


Other Common Plot Types

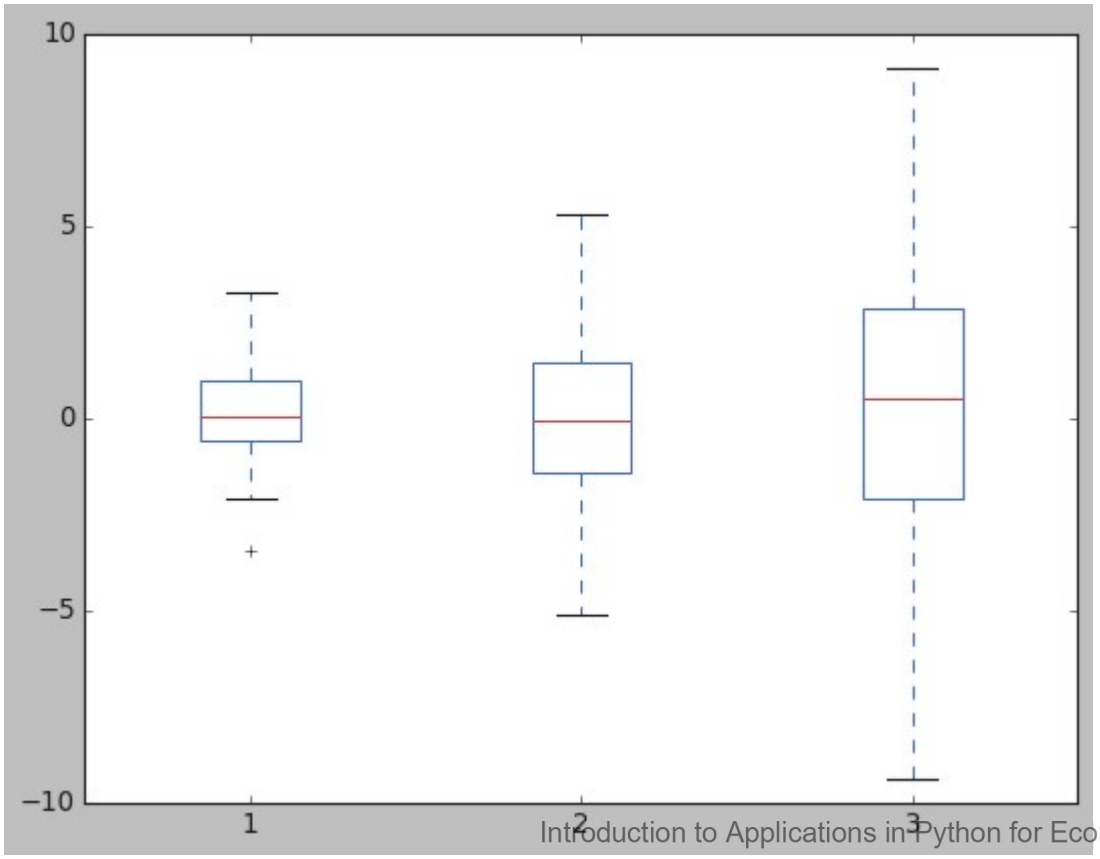
We will mostly use Seaborn for ploting, however, here are a few example should you desire to work with Matplotlib only:

In [8]:
```
import numpy as np
data = np.random.randn(1000)
plt.hist(data)
```

Out[8]:
```
(array([  1.,   2.,  24.,  78., 211., 288., 212., 153.,  23.,
8.]),
 array([-4.22669059, -3.48276093, -2.73883128, -1.99490162, -1.250
97197,
        -0.50704231,  0.23688735,  0.980817  ,  1.72474666,  2.468
67631,
         3.21260597]),
 <a list of 10 Patch objects>)
```



In [145]:
```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

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



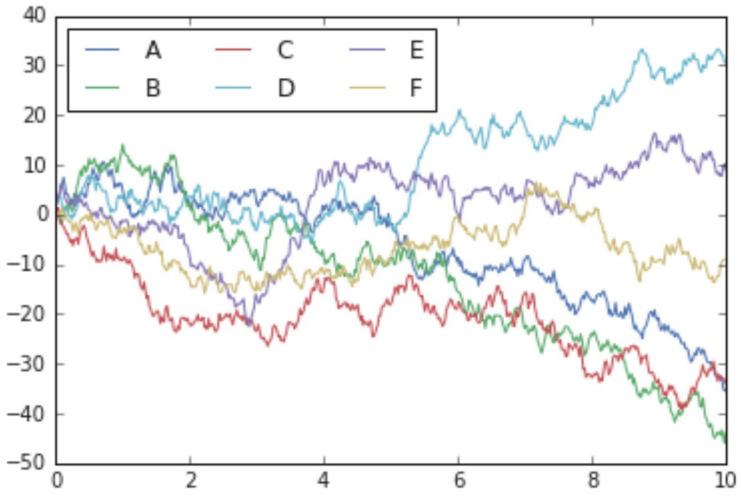
Seaborn

Matplotlib was created a decade before Pandas. Thus, it was not originally designed to work with DataFrame in Pandas. To answer this problem, Seaborn was created.
To be fair, Matplotlib has improved since the 2.0 release, we are now at 3.0, therefore, Matplotlib should work seamlessly with Pandas DataFrame.

Introduction

In [128]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

plt.style.use('classic') #Check https://matplotlib.org/ gallery for more styles, including ggplot
# This code below is only required if you works with jupyter notebook
%matplotlib inline
```

In [129]:
```
randgen = np.random.RandomState(0)
x = np.linspace(0,10,500)
y = np.cumsum(randgen.randn(500,6), 0) #cumsum(data=?,axis=?): cummulative sum.
plt.plot(x,y)
plt.legend('ABCDEF', ncol=3, loc='upper left');
```



Let's be frank, this does not look good, or at least on par with 21st century data visualization. The following codes show you how to do this in Seaborn

In [133]:
```
import seaborn as sns

sns.set() #this set the style as default of seaborn
```

In [134]:
```
plt.plot(x,y)
plt.legend('ABCDEF', ncol=2, loc='upper left')
#The quality of this graph is reduced when you use Jupyter Notebook.
```

Out[134]:
```
<matplotlib.legend.Legend at 0x16d5dc06f98>
```

You can think of Seaborn as adding a beautification len on top of Matplotlib. In fact, this is what Seaborn does, under the surface Seaborn use raw Matplotlib commands.

Distribution plots

In [15]:
```
import seaborn as sns
%matplotlib inline

tips = sns.load_dataset('tips')
tips.head()
```

Out[15]:

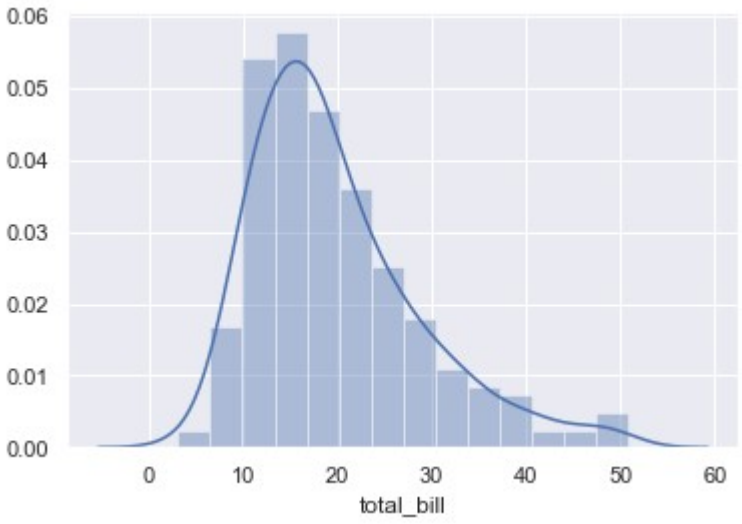
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

distplot

Plot a univariate distribution of observations.

```
In [20]: sns.distplot(tips['total_bill'])
```

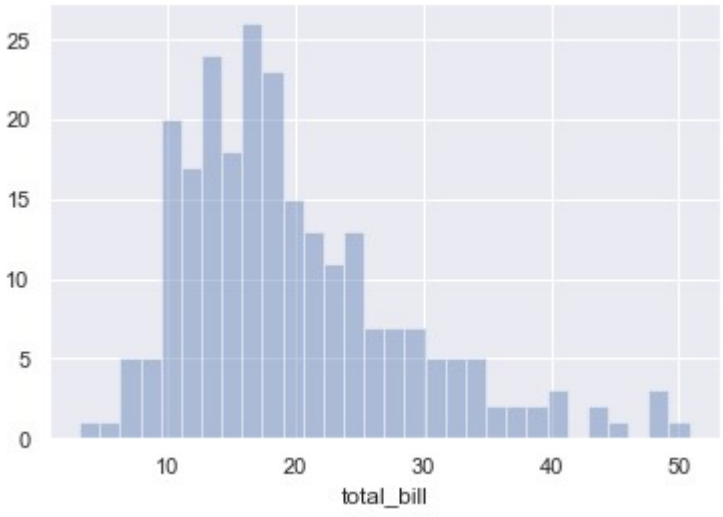
```
Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb1a52d6c8>
```



You can remove the kde line by simply turning it off in sns.distplot():

```
In [19]: sns.distplot(tips['total_bill'],kde=False,bins=30)
```

```
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb1a47fdc8>
```



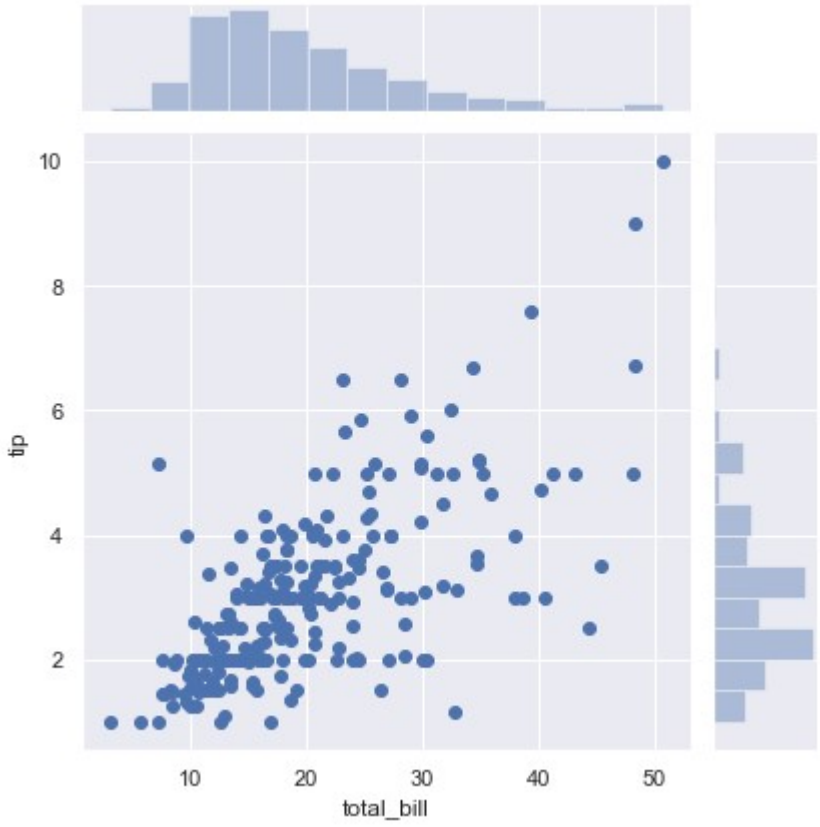
jointplot

As the name suggest, it is just a way to join 2 distplots together for a bivariate data plot. You can also think of it as a scatter plot but with added details. There are many settings for this type of plot:

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

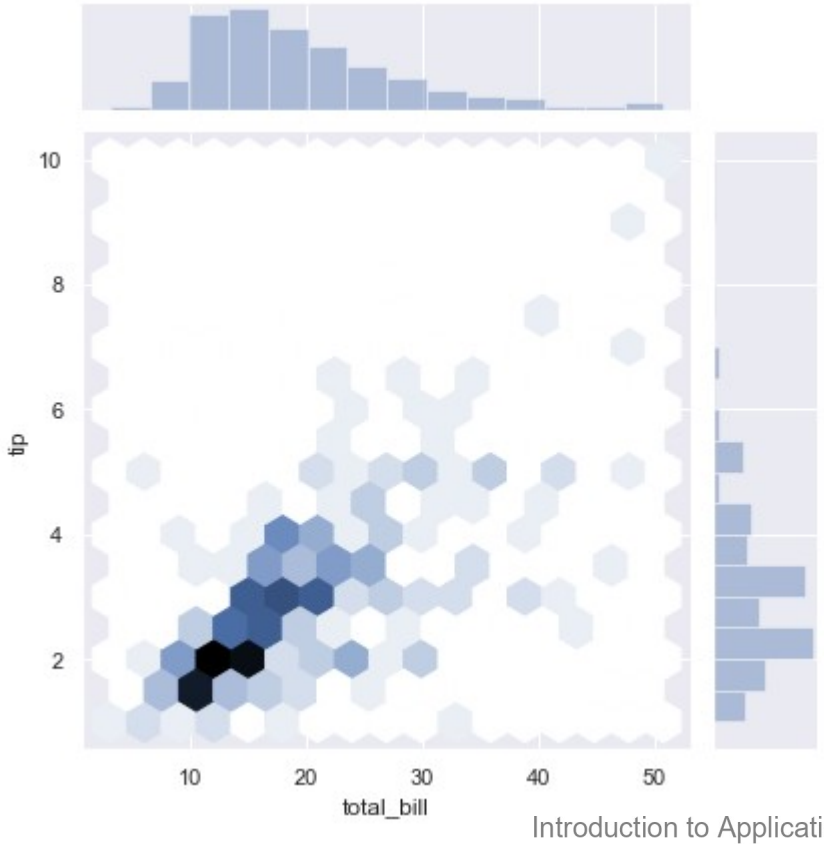
In [21]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='scatter')
```

Out[21]:
```
<seaborn.axisgrid.JointGrid at 0x1cb1a5d1f88>
```



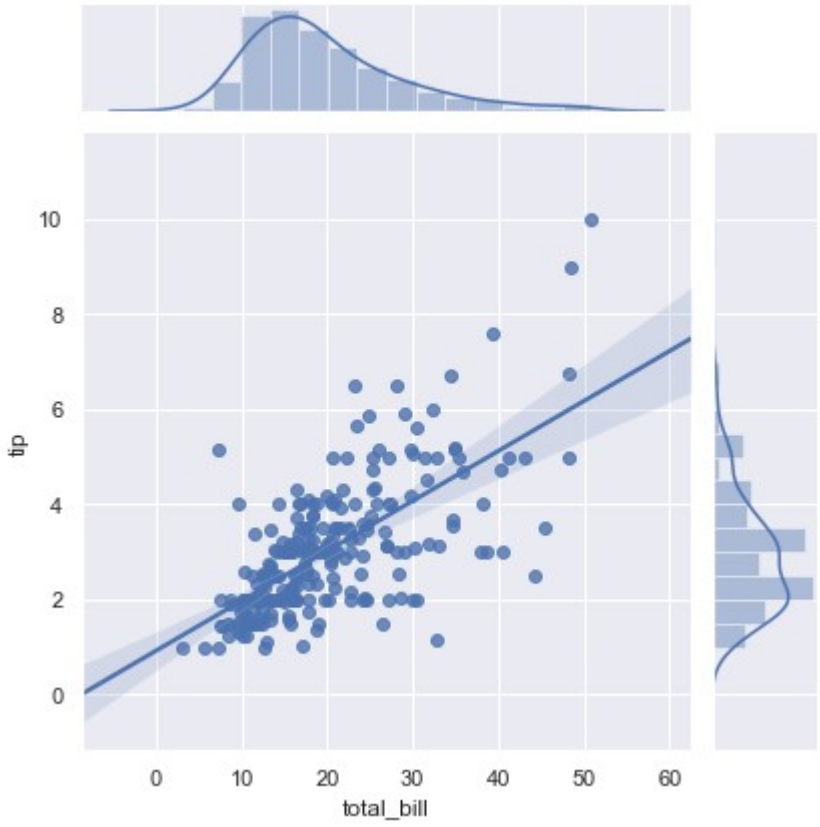
In [22]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='hex')
```

Out[22]:
```
<seaborn.axisgrid.JointGrid at 0x1cb1a72c488>
```



```
In [23]: sns.jointplot(x='total_bill',y='tip',data=tips,kind='reg')
```

```
Out[23]: <seaborn.axisgrid.JointGrid at 0x1cb1a88e048>
```



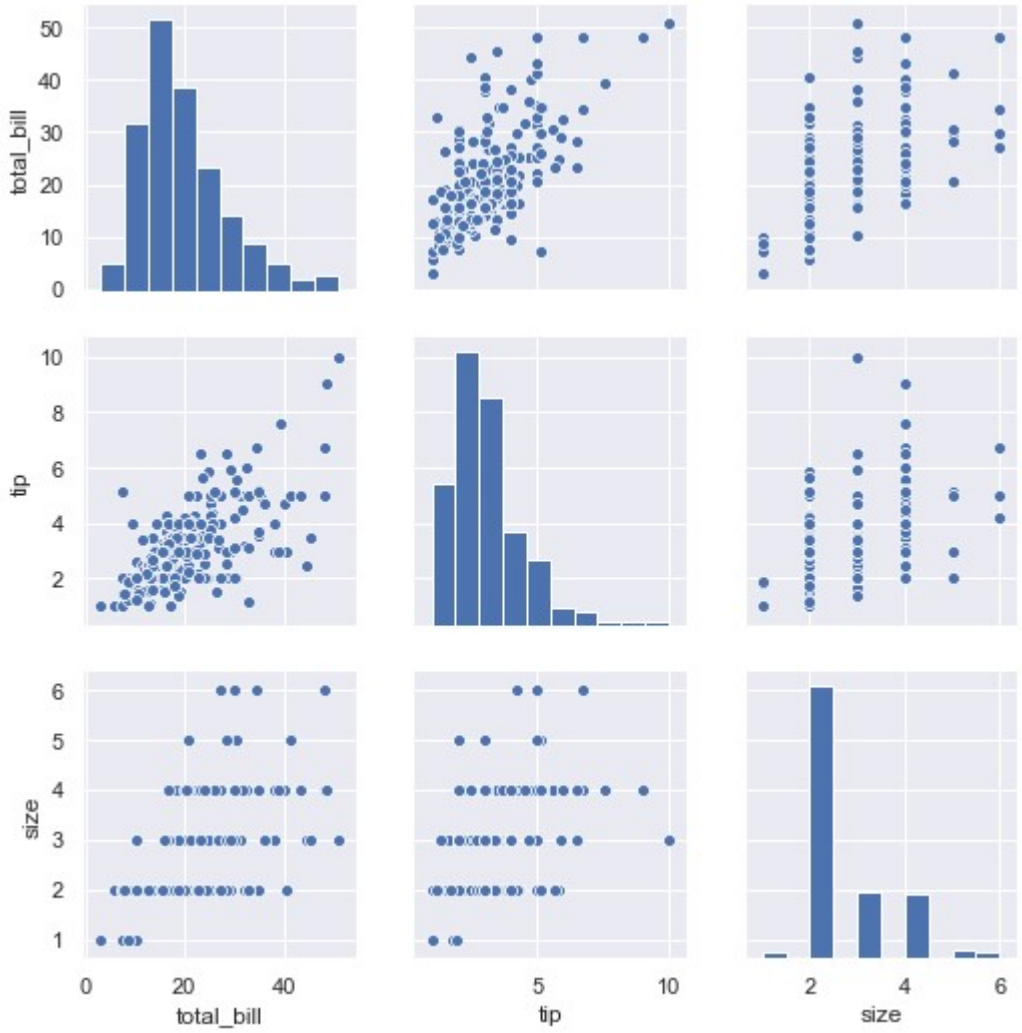
pairplot

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

usually, you want to create this figure to check for potential correlation between each and every variables in your dataset

```
In [24]: sns.pairplot(tips)
```

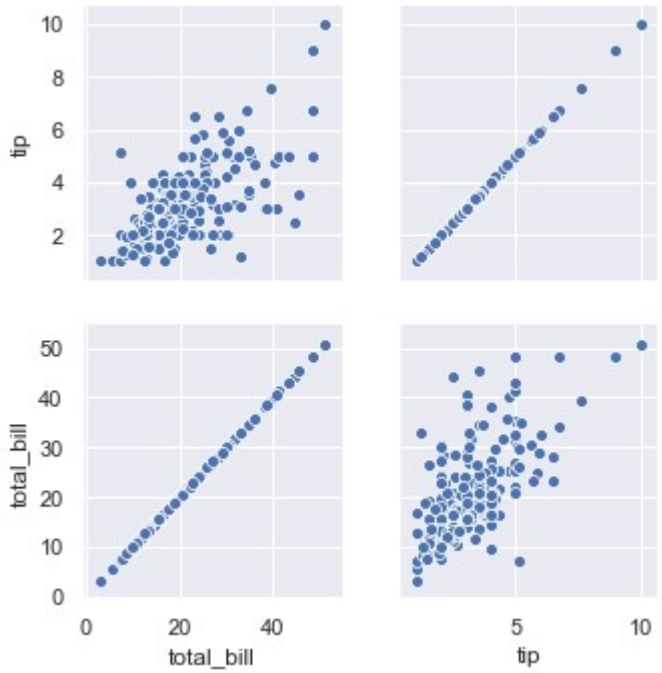
```
Out[24]: <seaborn.axisgrid.PairGrid at 0x1cb1b9b5648>
```



You can also selectively choose what columns to plot

In [29]:
```
sns.pairplot(tips, x_vars=["total_bill", "tip"], y_vars=["tip", "total_bill"])
```

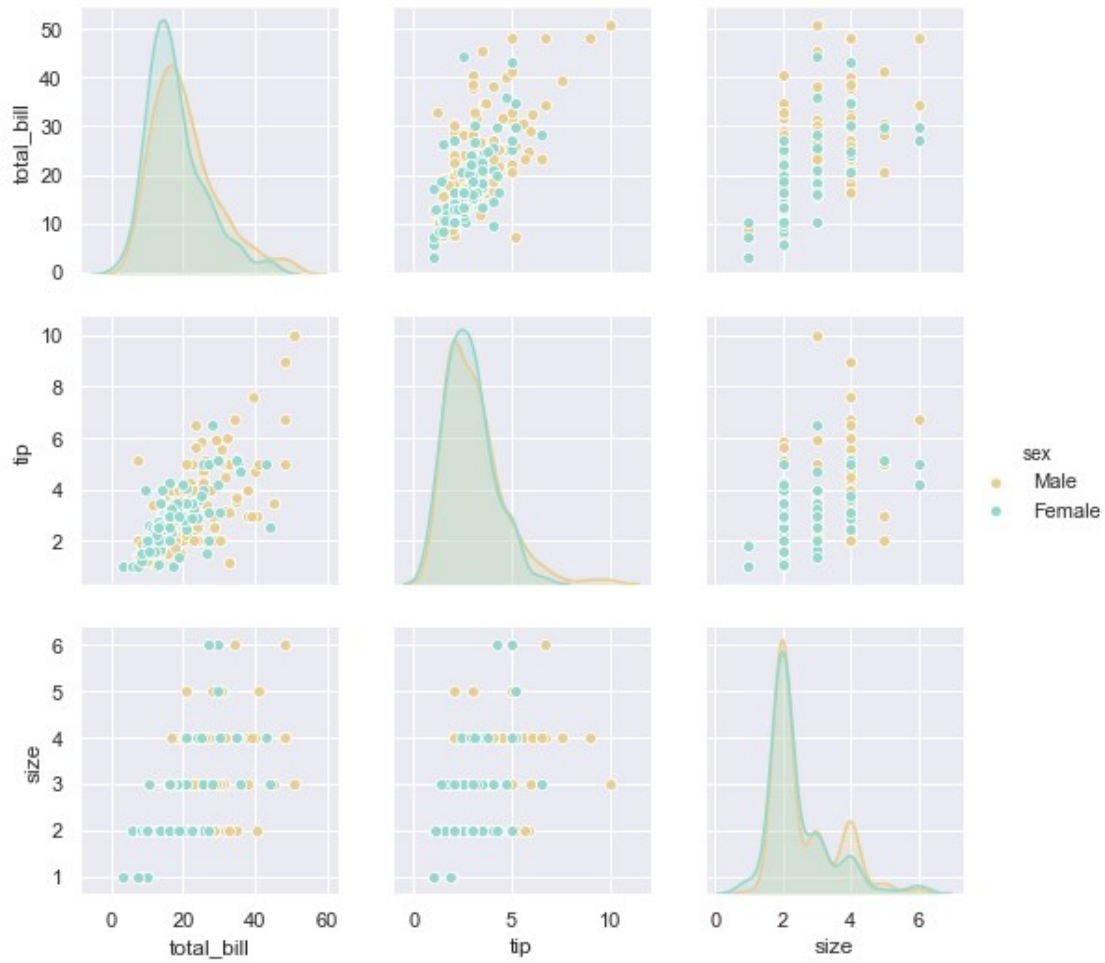
Out[29]:
```
<seaborn.axisgrid.PairGrid at 0x1cb1c2a45c8>
```



There are many color theme (palette) and it is easy to change

In [36]:
```
sns.pairplot(tips,hue='sex',palette='BrBG')
# Color is a field of itself, here is a link for those artistic soul
s:
# https://seaborn.pydata.org/tutorial/color_palettes.html
```

Out[36]:
```
<seaborn.axisgrid.PairGrid at 0x1cb1f2d0188>
```

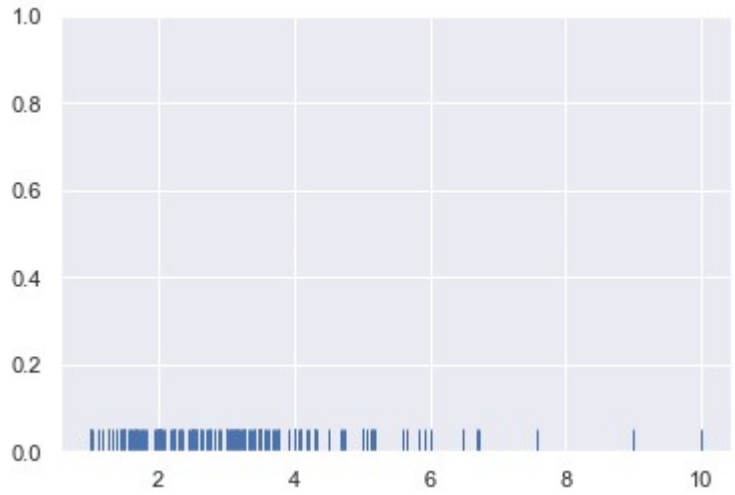


rugplot

In a rugplot every point on a univariate distribution is drawn using a stick mark. Somewhat looks like a point up strand of thread on a rug.

In [37]:
```
sns.rugplot(tips['tip'])
```

Out[37]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb1ec5c148>
```

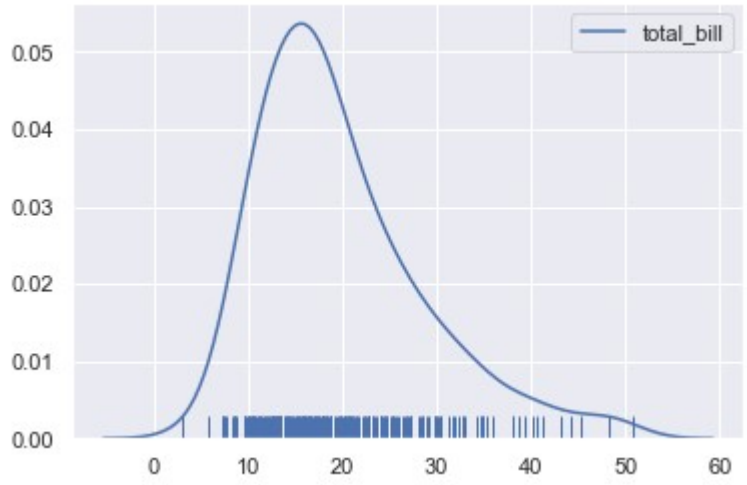


kdeplot

kdeplot is a non-parametric way to estimate the probability density function of a random variable. These KDE plots replace every single observation with a Gaussian (Normal) distribution centered around that value. For example:

In [40]:
```
sns.kdeplot(tips['total_bill'])
sns.rugplot(tips['total_bill'])
```

Out[40]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb1f8c1308>
```



Categorical Data Plots

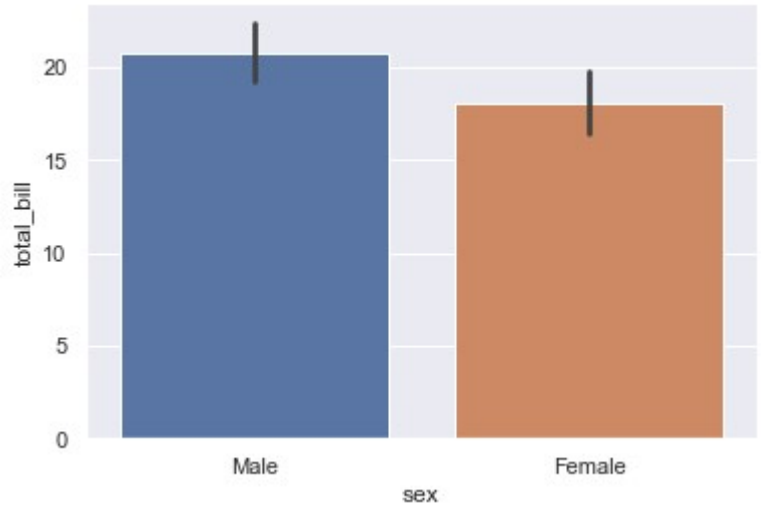
We now will take a look at plots that is used for discrete or categorical data

barplot

barplot is a general plot that allows you to aggregate the categorical data based off some function, by default the mean:

```
In [42]: sns.barplot(x='sex',y='total_bill',data=tips)
```

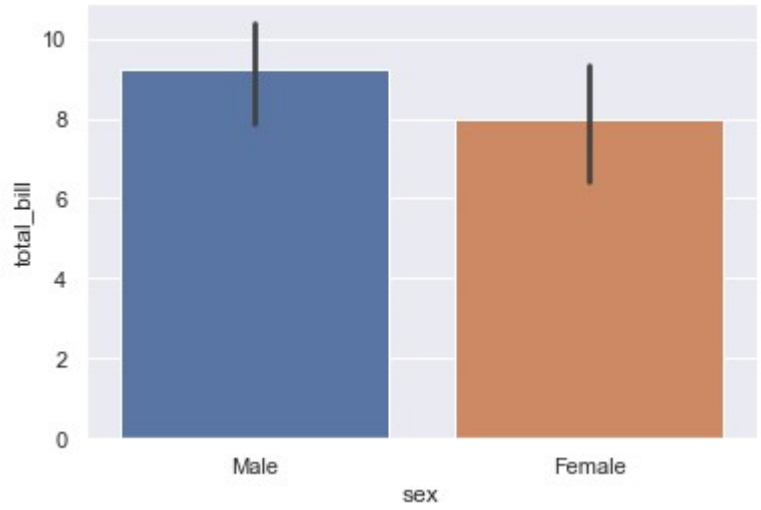
```
Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb1fb04b88>
```



The black line you are seeing on top of the bar is called "error bar" in Seaborn. This error bar provides some indication of the uncertainty around the estimate (e.g mean of that category).
You can change the estimate into something else instead of mean, for example standard deviation:

```
In [43]: sns.barplot(x='sex',y='total_bill',data=tips,estimator=np.std) #we a
         re using numpy standard deviation here
```

```
Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb1fcb43c8>
```

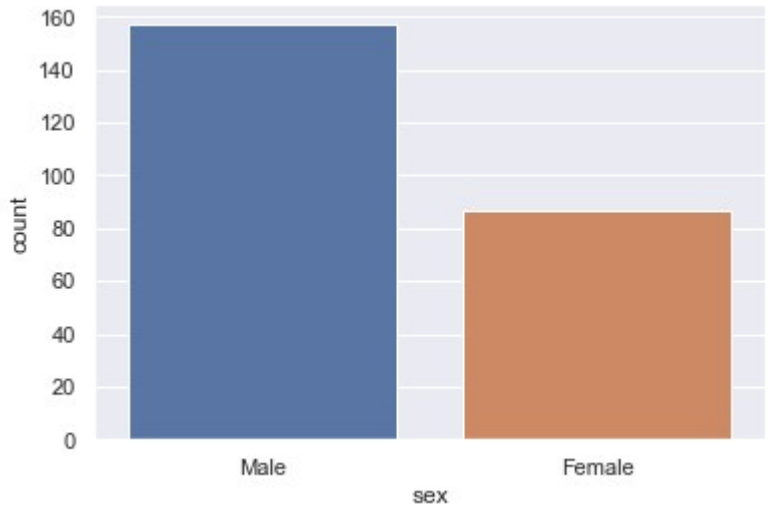


countplot

As the name suggest, it just counts the number of occurrences

```
In [44]: sns.countplot(x='sex',data=tips)
```

```
Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb1fcfa7c8>
```

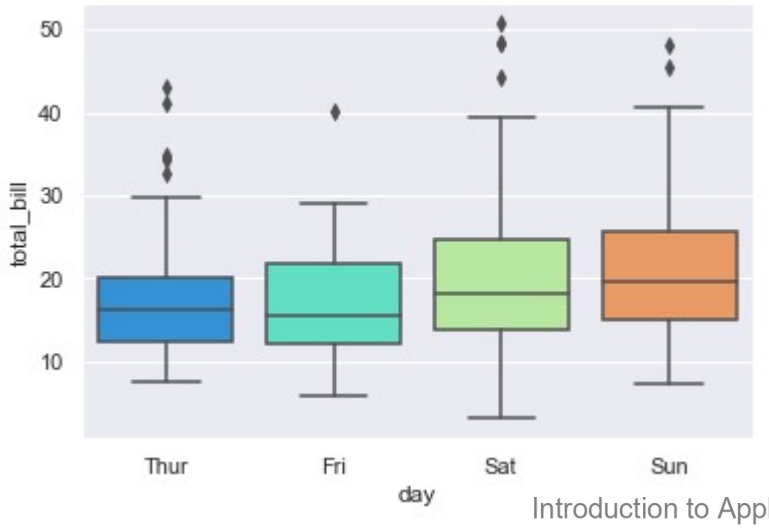


boxplot

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.

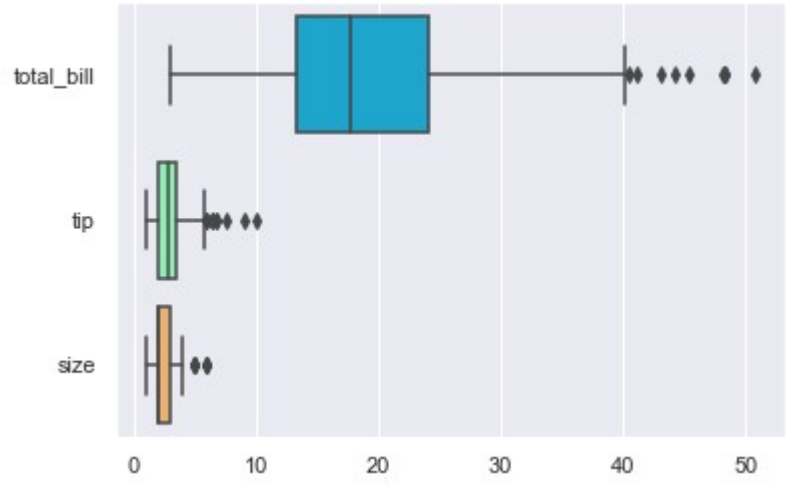
```
In [45]: sns.boxplot(x="day", y="total_bill", data=tips, palette='rainbow')
```

```
Out[45]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb1fd4ae88>
```



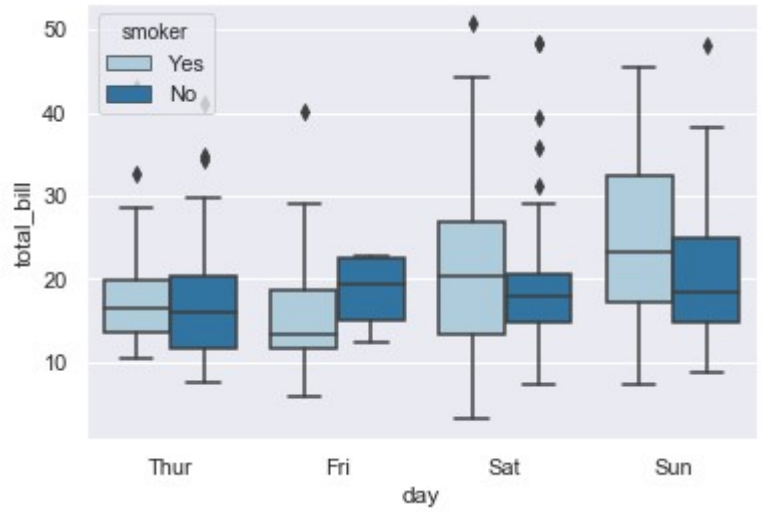
In [46]:
```
# You can also flip this 90 degree easily with Seaborn:
sns.boxplot(data=tips,palette='rainbow',orient='h') # h stands for h
orizontal
```

Out[46]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb1fdf8508>
```



In [47]:
```
sns.boxplot(x="day", y="total_bill", hue="smoker",data=tips, palette
="Paired")
```

Out[47]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb1fea3dc8>
```

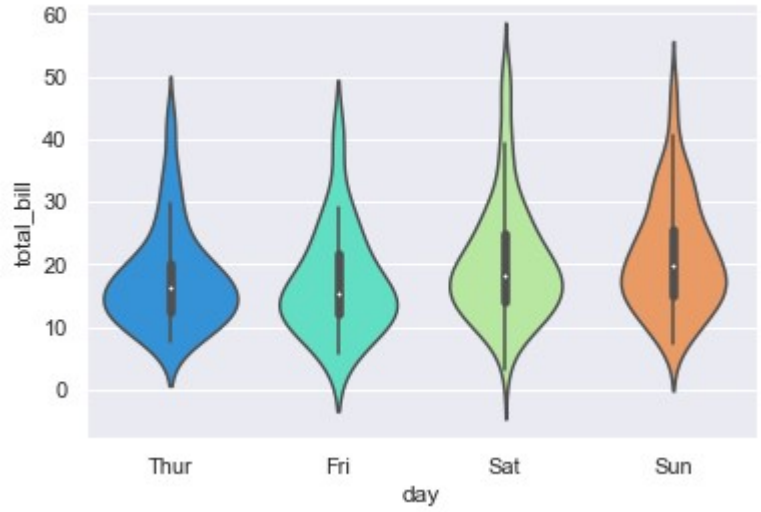


violinplot

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 (kde) of the underlying distribution.

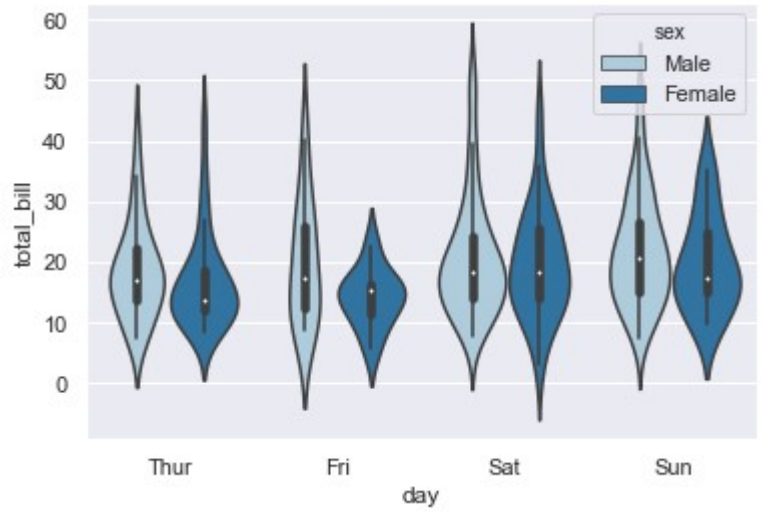
In [48]:
```
sns.violinplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

Out[48]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb1ff98c48>
```



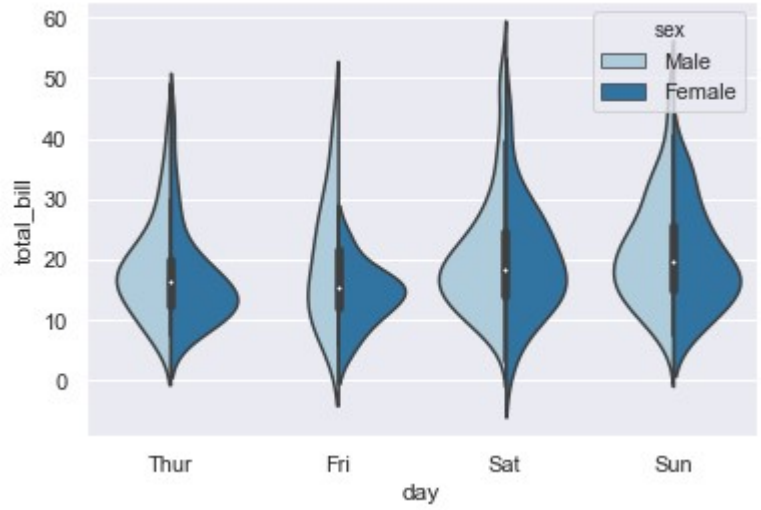
In [52]:
```
sns.violinplot(x="day", y="total_bill", data=tips, hue='sex', palett
e='Paired')
```

Out[52]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb2120b588>
```



In [53]:
```
sns.violinplot(x="day", y="total_bill", data=tips,hue='sex',split=True,palette='Paired')
```

Out[53]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb212bab08>
```



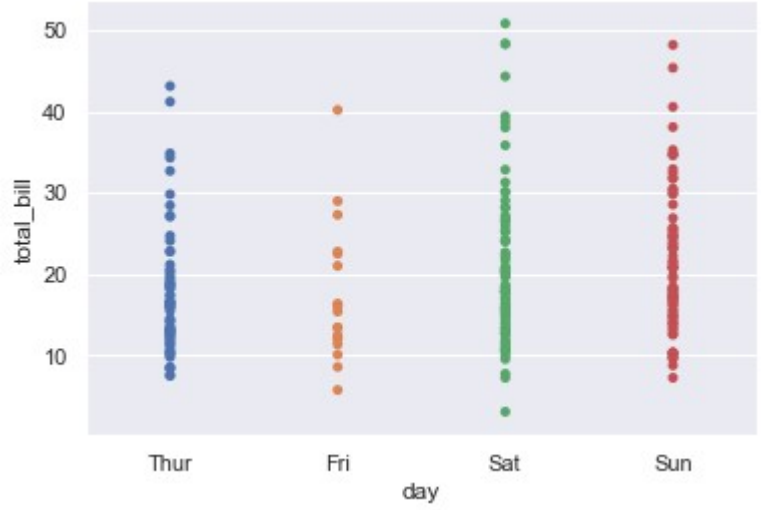
stripplot and swarmplot

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 [60]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=False)
# jitter can be from 0 to 1
```

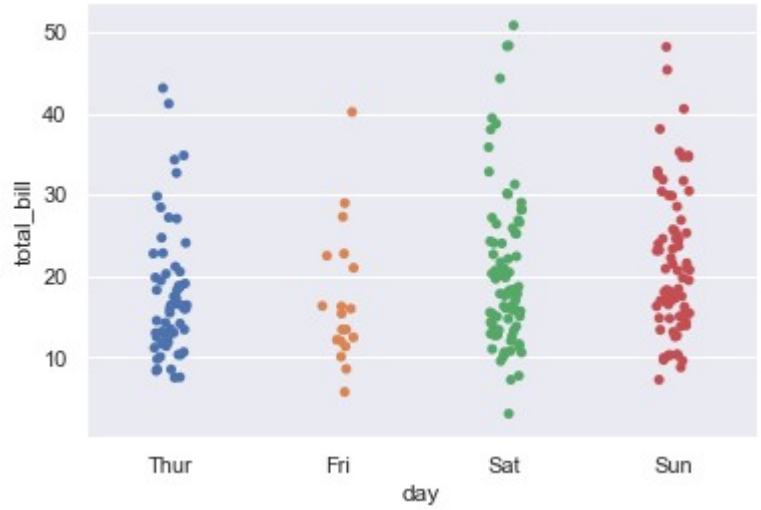
Out[60]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb2154d2c8>
```



Using jitter (by default, it's True), you can see the distribution a bit clearer, especially when the points are overlapping each other too much

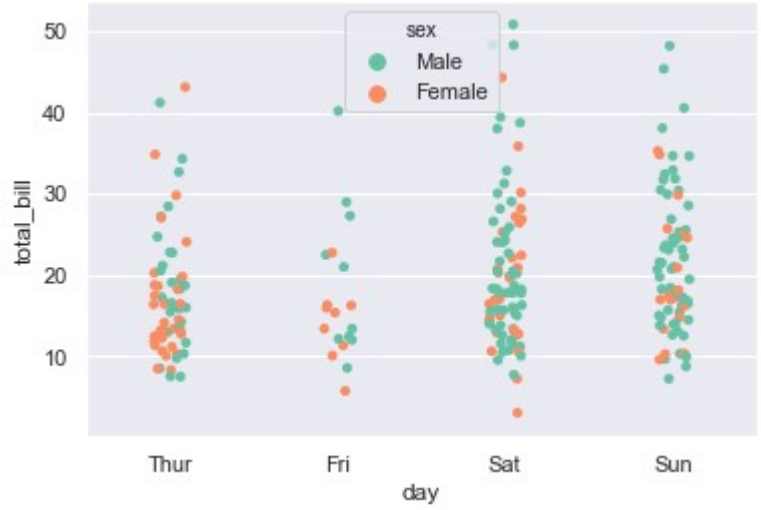
In [61]:
```
sns.stripplot(x="day", y="total_bill", data=tips)
```

Out[61]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb2159e588>
```



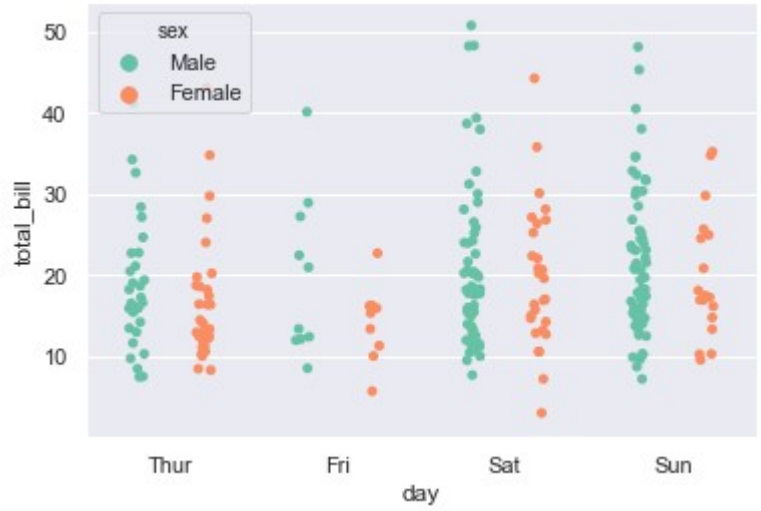
In [62]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=True, hue='
sex', palette='Set2')
```

Out[62]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb2165f8c8>
```



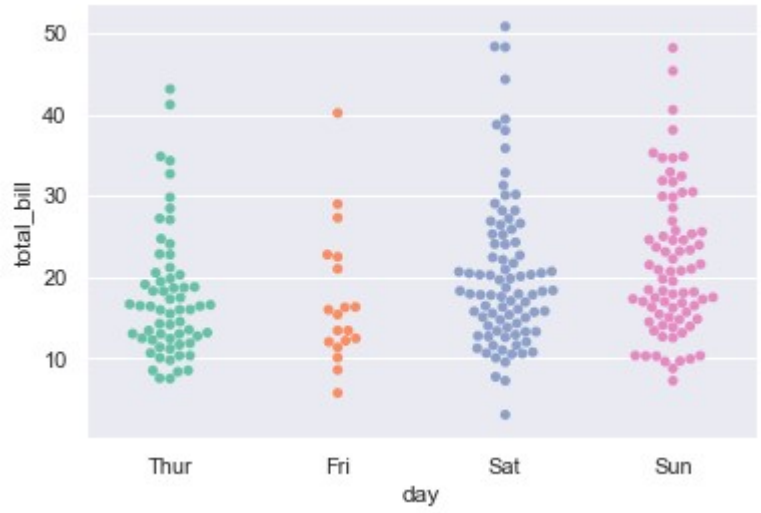
In [68]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True,hue='sex', palette='Set2', dodge=True)
#Recently, split has been changed to 'dodge'
```

Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb2194f508>



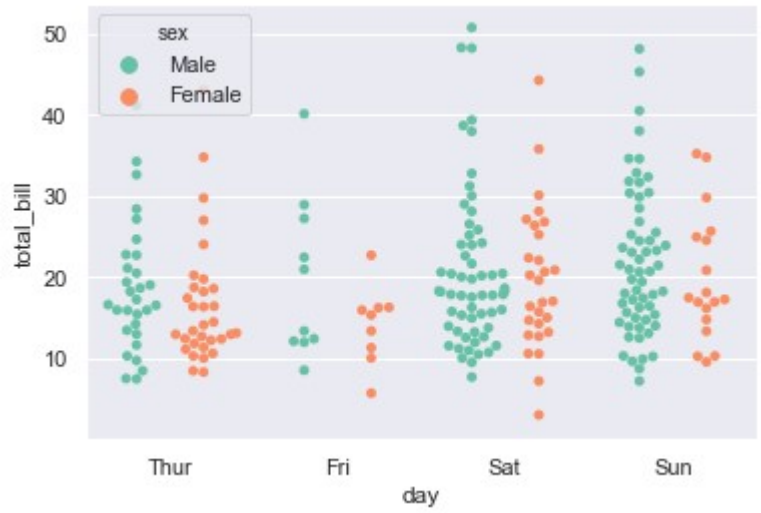
In [65]:
```
sns.swarmplot(x="day", y="total_bill", data=tips, palette='Set2')
```

Out[65]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb216d4808>



In [67]:
```
sns.swarmplot(x="day", y="total_bill", hue='sex', data=tips, palette
="Set2", dodge=True)
```

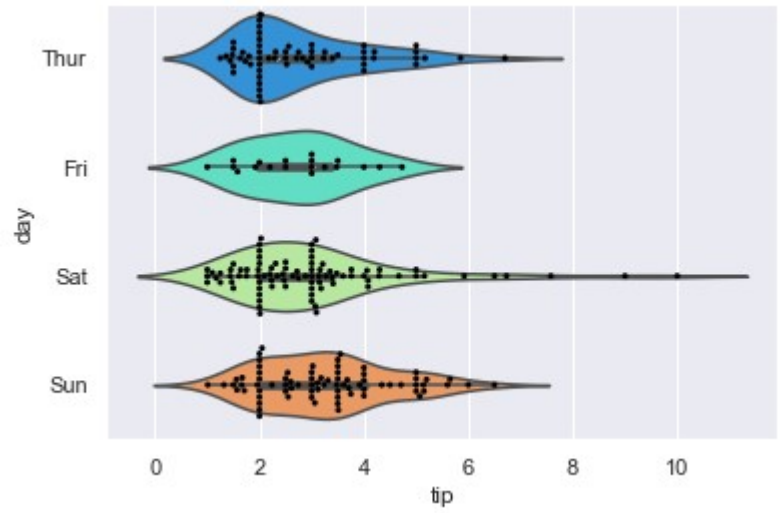
Out[67]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb218d5f08>
```



Combining Categorical Plots

In [71]:
```
sns.violinplot(x="tip", y="day", data=tips, palette='rainbow')
sns.swarmplot(x="tip", y="day", data=tips, color='black', size=3)
```

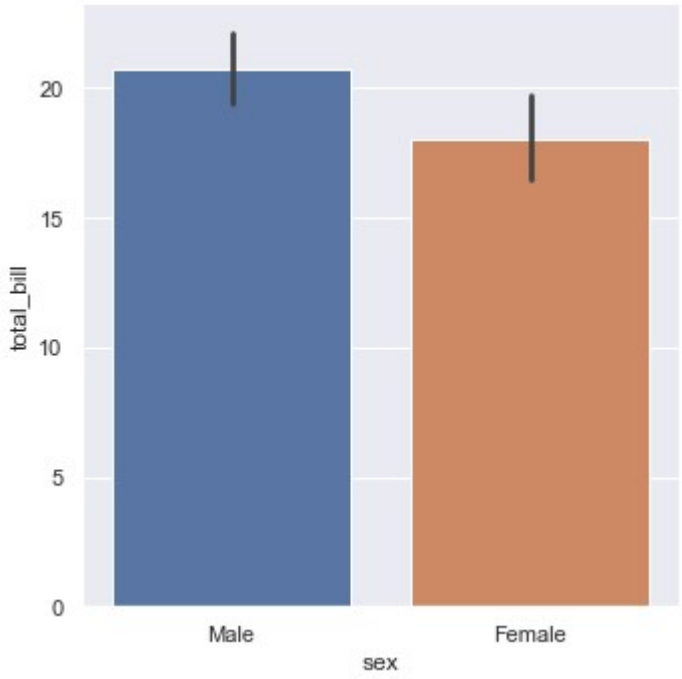
Out[71]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb22a34548>
```



catplot

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

In [96]:
```
sns.catplot(x='sex',y='total_bill',data=tips, kind='bar');
```



Grid

The pairplot above is a special case of grid. 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 [100]:
```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [101]:
```
iris = sns.load_dataset('iris')
iris.head()
```

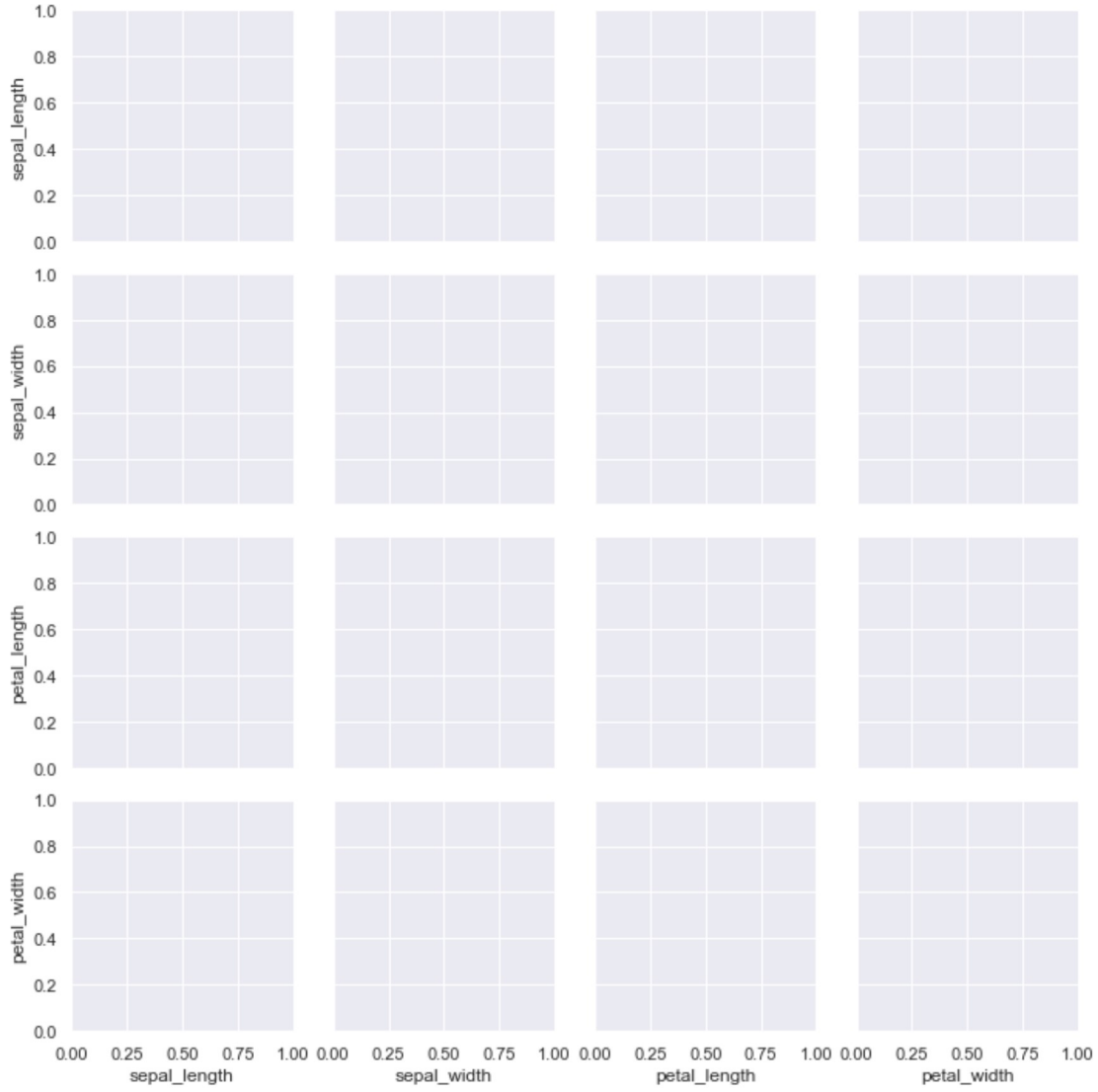
Out[101]:

	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

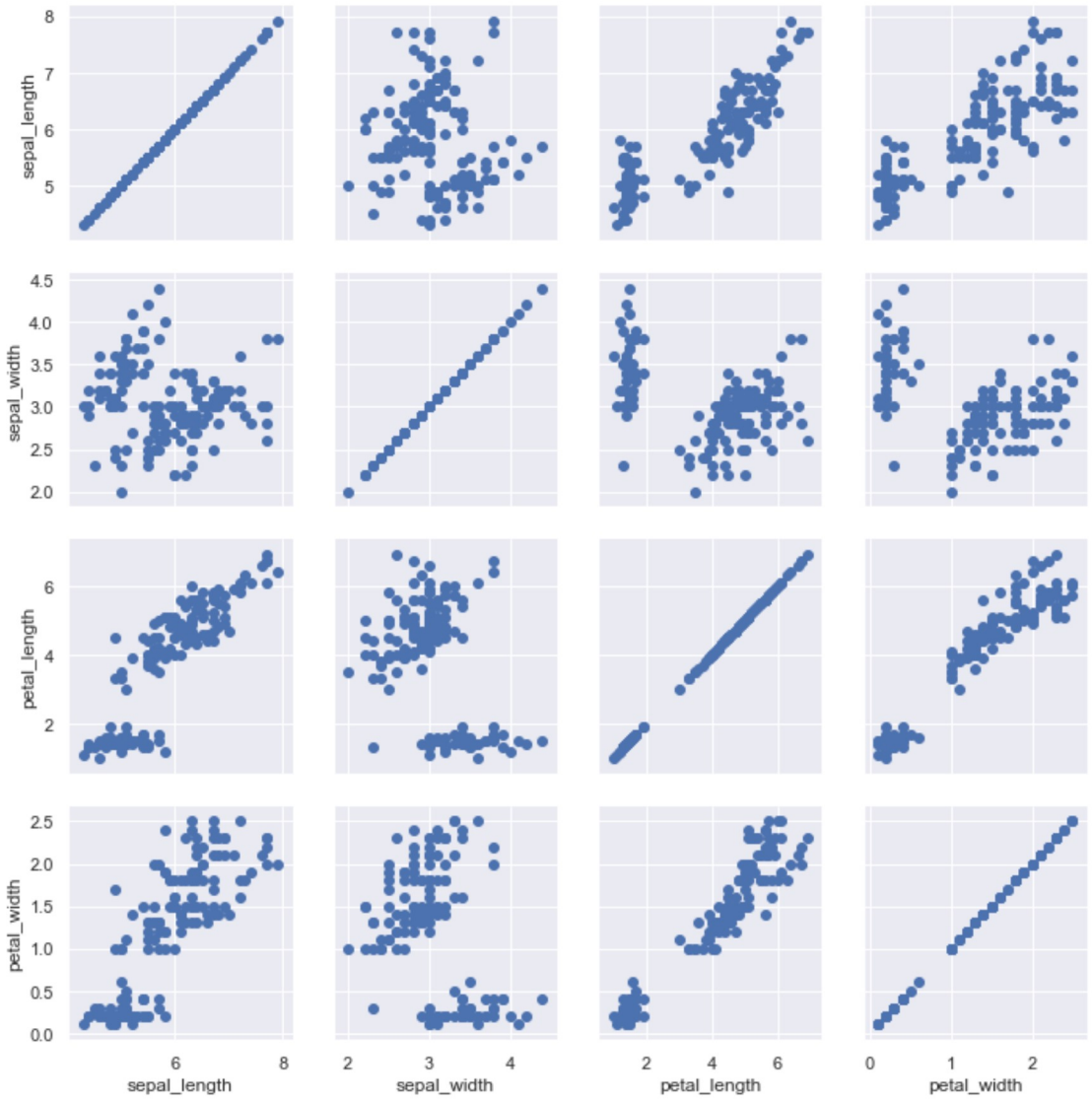
Pairgrid is a subplot grid for plotting pairwise relationships in a dataset.

In [105]:
```
# First, you create a pair, but not actually plotting anything yet.
g = sns.PairGrid(iris)
```



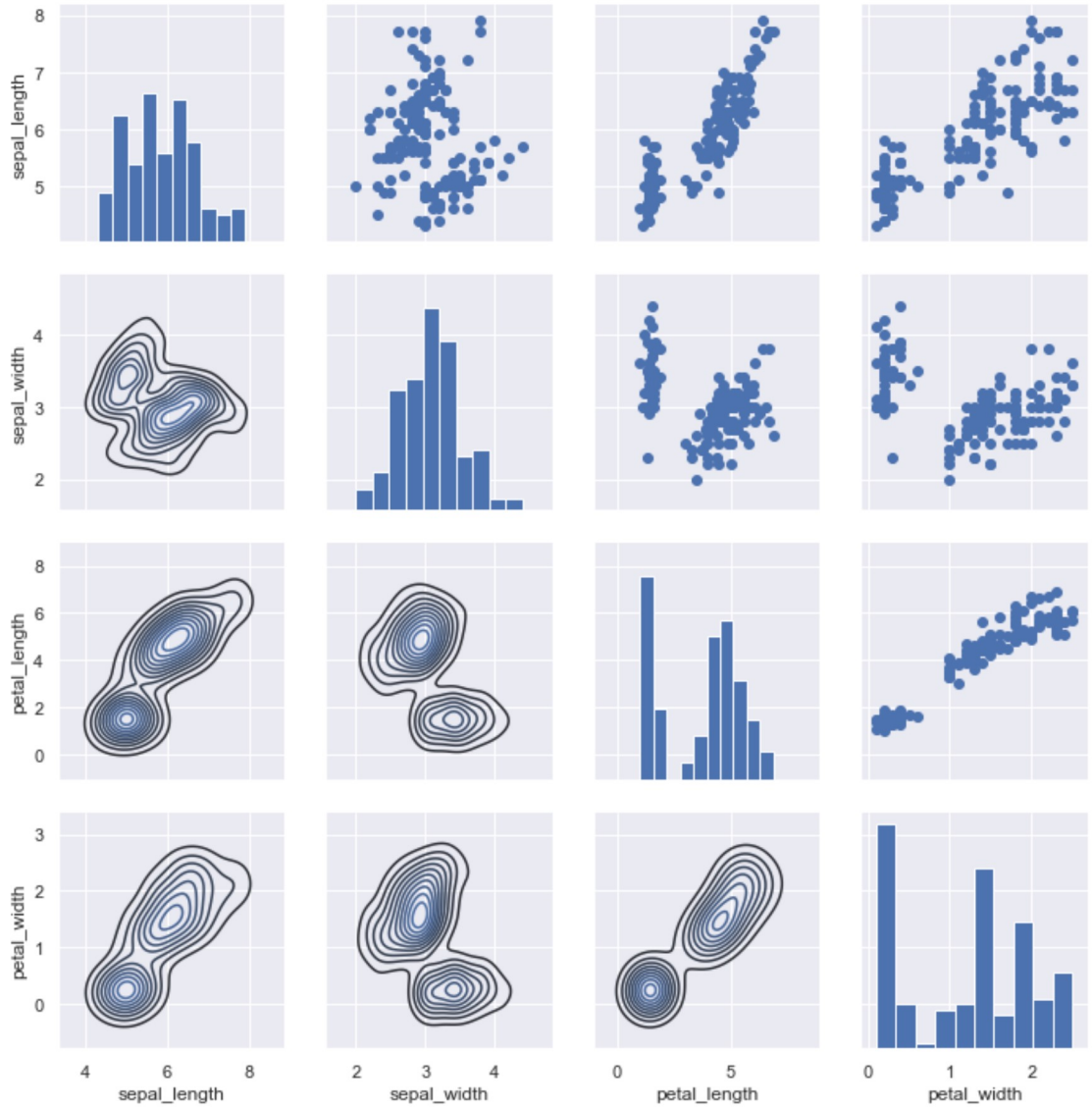
In [108]:
```
# Then you start plotting on this grid:
g = sns.PairGrid(iris) # If you are using Jupyter, you need to write this code in one cell with the below
g.map(plt.scatter)
```

Out[108]: <seaborn.axisgrid.PairGrid at 0x1cb284d3408>



In [109]:
```
# What makes grid different is that you can now customize the grid f
reely:
g = sns.PairGrid(iris)
g.map_diag(plt.hist)
g.map_upper(plt.scatter)
g.map_lower(sns.kdeplot)
```

Out[109]:
```
<seaborn.axisgrid.PairGrid at 0x1cb25878908>
```



Facet Grid

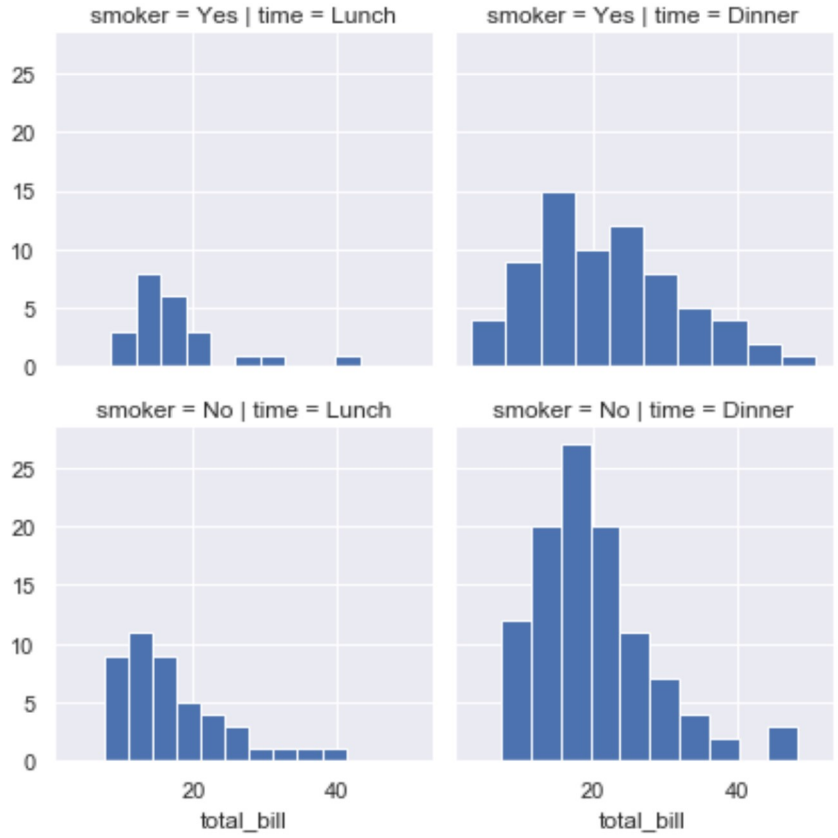
FacetGrid is the general way to create grids of plots based off of a feature:

In [111]:
```
tips = sns.load_dataset('tips')
tips.head()
```

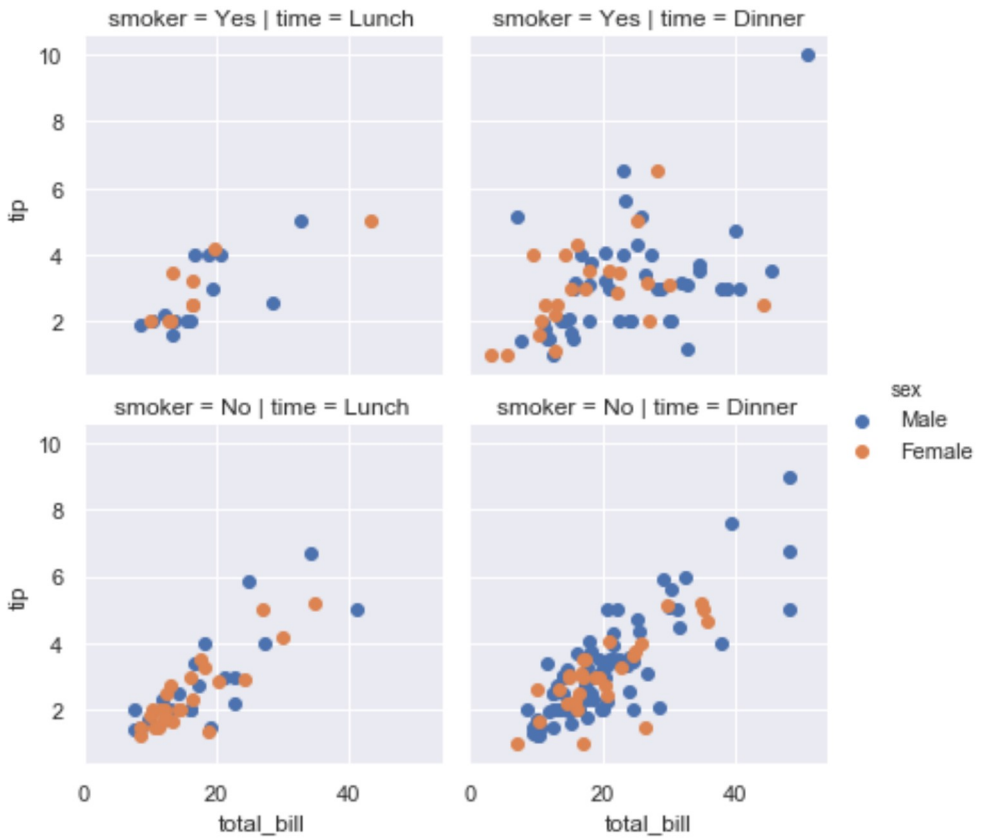
Out[111]:

	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 [112]:
```
g = sns.FacetGrid(tips, col="time",  row="smoker")
g = g.map(plt.hist, "total_bill")
```



In [113]:
```
g = sns.FacetGrid(tips, col="time", row="smoker", hue='sex')
# Notice how the arguments come after plt.scatter call
g = g.map(plt.scatter, "total_bill", "tip").add_legend()
```



Linear Regression Plots

Seaborn has many built-in capabilities for regression plots, however I'll discuss them during the next lecture. For now, we only look at linear regression.

Implot 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.

In [114]:
```
import seaborn as sns
%matplotlib inline
```

In [115]:
```
tips = sns.load_dataset('tips')
tips.head()
```

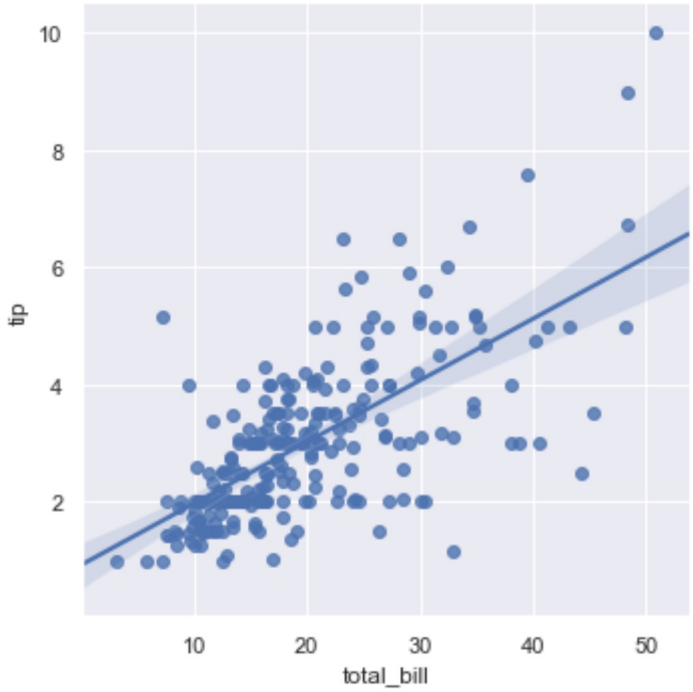
Out[115]:

	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

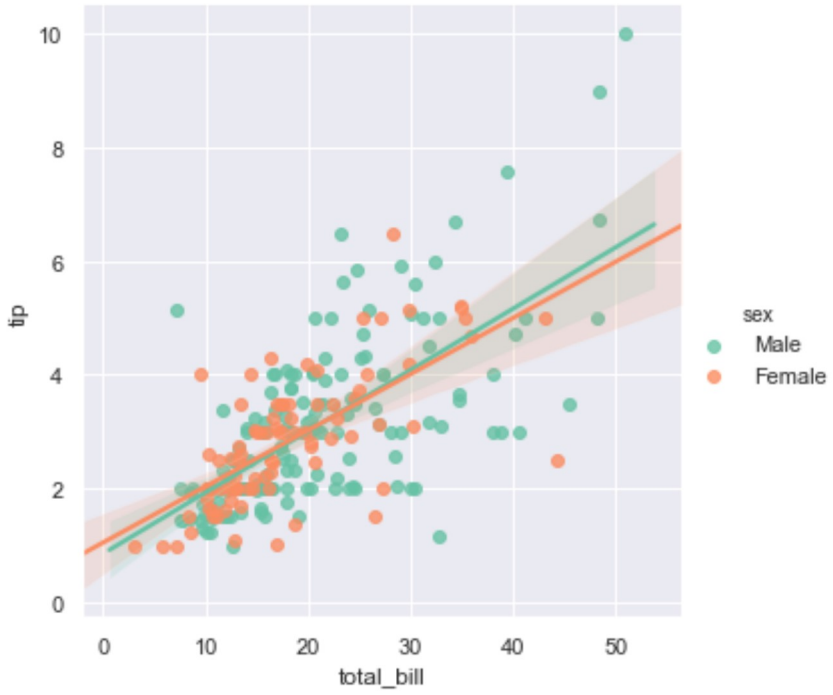
In [119]:
```
sns.lmplot(x='total_bill',y='tip',data=tips)
```

Out[119]:
```
<seaborn.axisgrid.FacetGrid at 0x1cb2b1bc448>
```



In [120]:
```
sns.lmplot(x='total_bill', y='tip', data=tips, hue='sex', palette='Set2')
```

Out[120]:
```
<seaborn.axisgrid.FacetGrid at 0x1cb2b20b8c8>
```



Markers

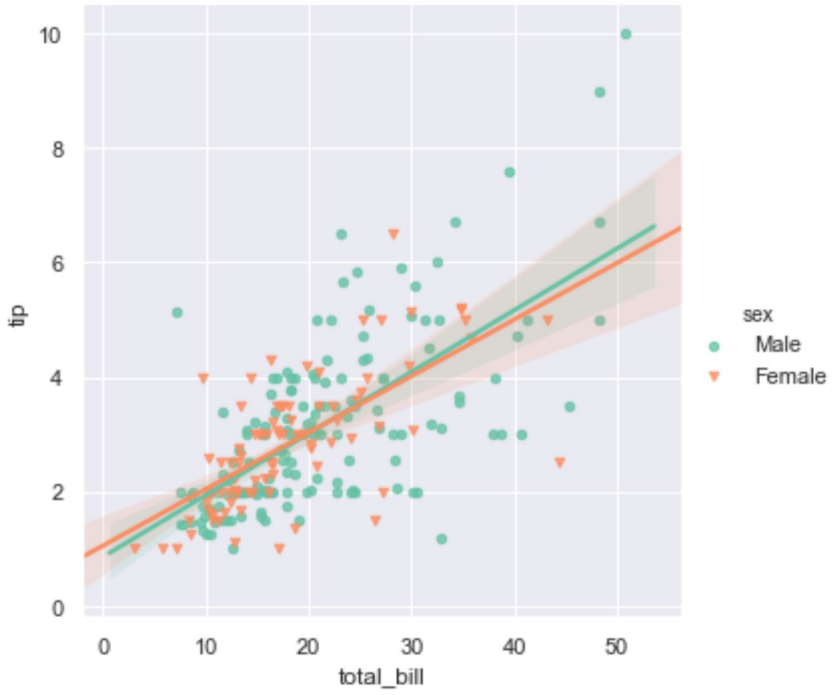
Fair warning: You probably won't remember this on top of your head, nor you are asked to do so, therefore, please refer to the document here: http://matplotlib.org/api/markers_api.html (http://matplotlib.org/api/markers_api.html)

In general, lmplot kwargs get passed through to **regplot** which is a more general form of lmplot(). regplot then pass these kwargs to plt.scatter. So you want to set the s parameter in that dictionary, which corresponds to the squared markersize (a bit confusingly).

The bottomline is if you want to pass argument to plt.scatter, you have to put them inside a dictionary with the base matplotlib arguments, in this case, s for size of the marker. In general, you probably won't remember this off the top of your head, but instead reference the documentation.

In [125]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,hue='sex',palette='Set2',
           markers=['o','v'],scatter_kws={'s':20})
# If you want to customize the regression line, you can for example add: line_kws={'lw': 2, 'color': 'b'}
```

Out[125]:
```
<seaborn.axisgrid.FacetGrid at 0x1cb2b433548>
```



Using a Grid

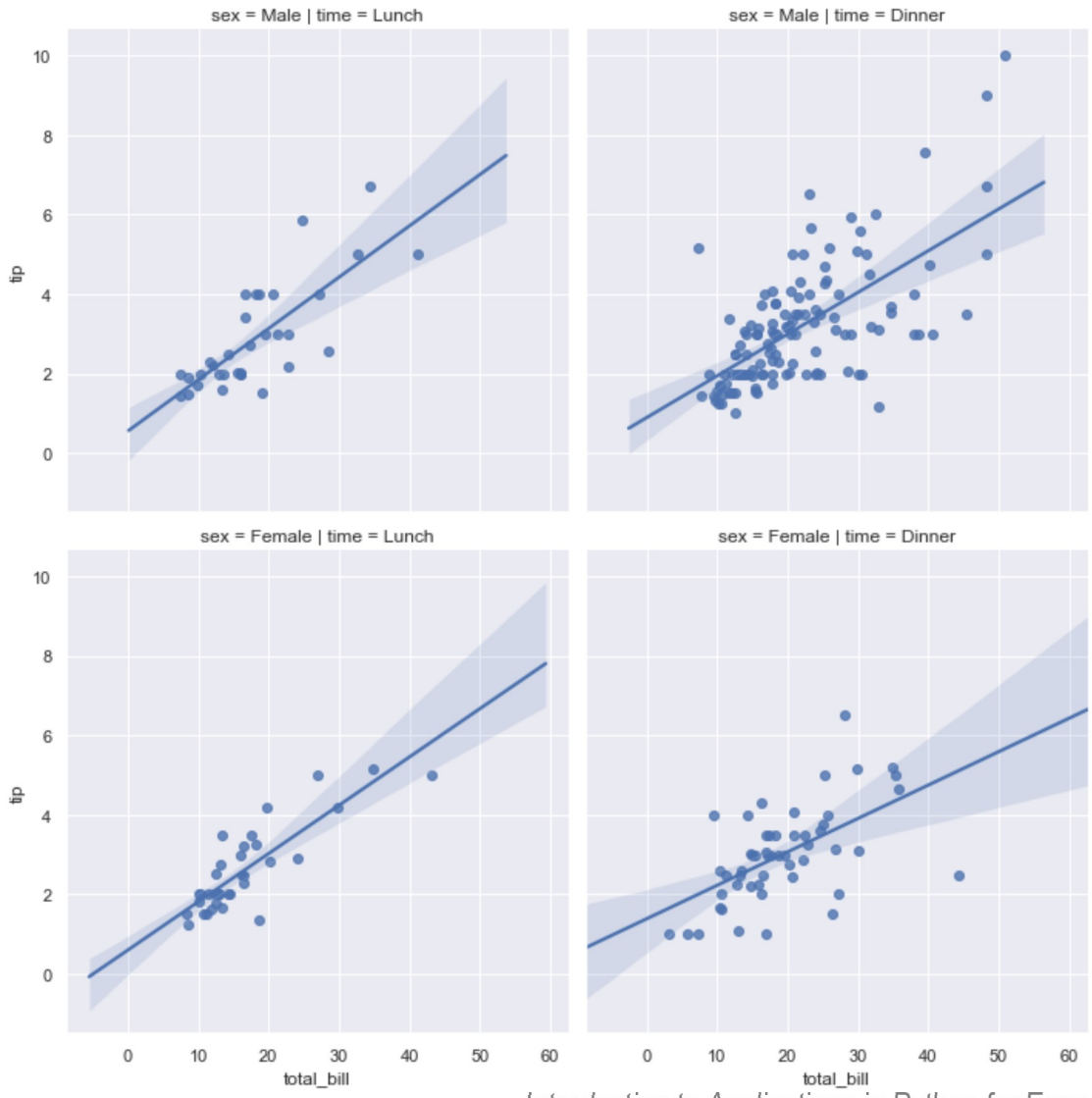
We can add more variable separation through columns and rows with the use of a grid. Just indicate this with the col or row arguments:

In [126]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,col='sex')
```

Out[126]:
```
<seaborn.axisgrid.FacetGrid at 0x1cb2b39b488>
```

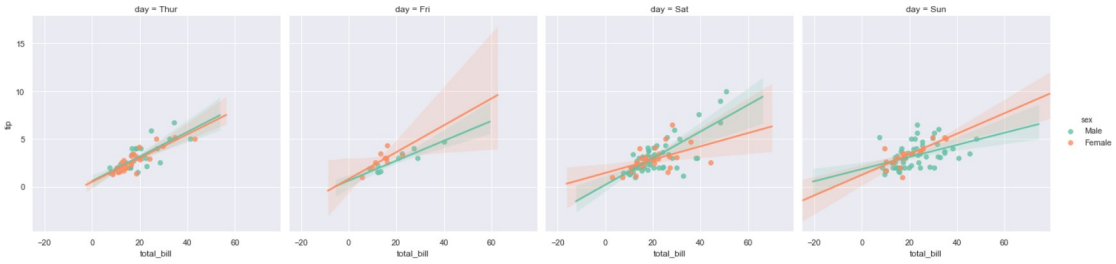
In [127]:
```
sns.lmplot(x="total_bill", y="tip", row="sex", col="time",data=tips)
```

Out[127]:
```
<seaborn.axisgrid.FacetGrid at 0x1cb2c5b1d08>
```



In [128]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,col='day',hue='sex',palette='Set2')
```

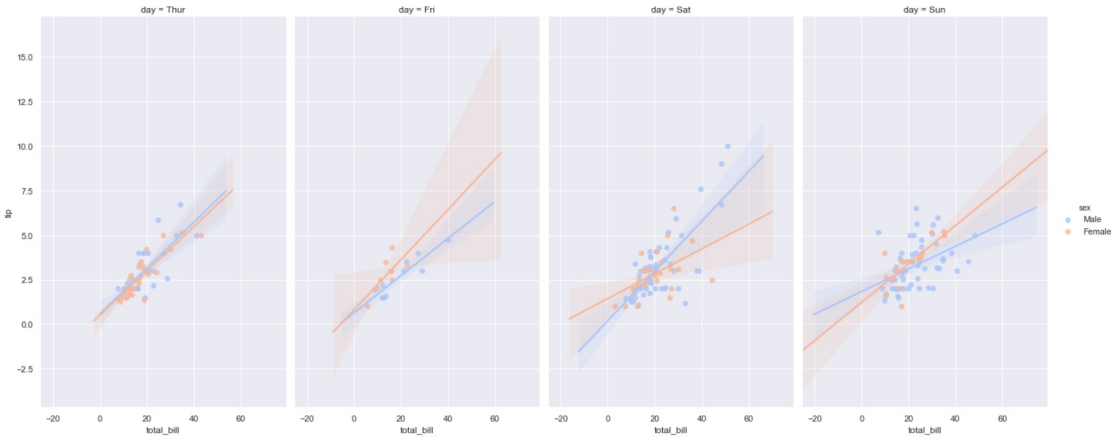
Out[128]: <seaborn.axisgrid.FacetGrid at 0x1cb2c941508>



Size and aspect ratio

In [132]:
```
sns.lmplot(x='total_bill',y='tip',data=tips,col='day',hue='sex',palette='coolwarm',
           aspect=0.6,height=8) # Aspect will change the width while keeping the height constant
```

Out[132]: <seaborn.axisgrid.FacetGrid at 0x1cb2d977308>



Short Introduction into Styling and Color

So...you spend a lot of time in analyzing your data, choosing the right graph, you think you convey the message clearly, then you do something like this:

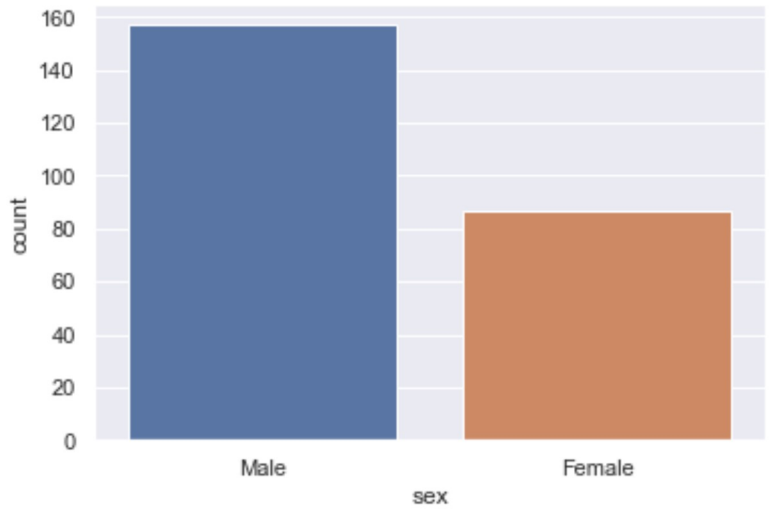
Here I don't really mean the font, what I meant was styling and color picking. It seems trivial and frankly a job of a designer (not me, I'm an economist). The truth is most of the time, you'll be in charge of your graph, it is unlikely a designer will help you do it. Picking a wrong color/style will greatly affect the audience perceptions of your graph, it might even make the data unreadible.

Further documents:
https://seaborn.pydata.org/tutorial/color_palettes.html (https://seaborn.pydata.org/tutorial/color_palettes.html)
http://seaborn.pydata.org/tutorial/aesthetics.html (http://seaborn.pydata.org/tutorial/aesthetics.html)

In [133]:
```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
tips = sns.load_dataset('tips')
```

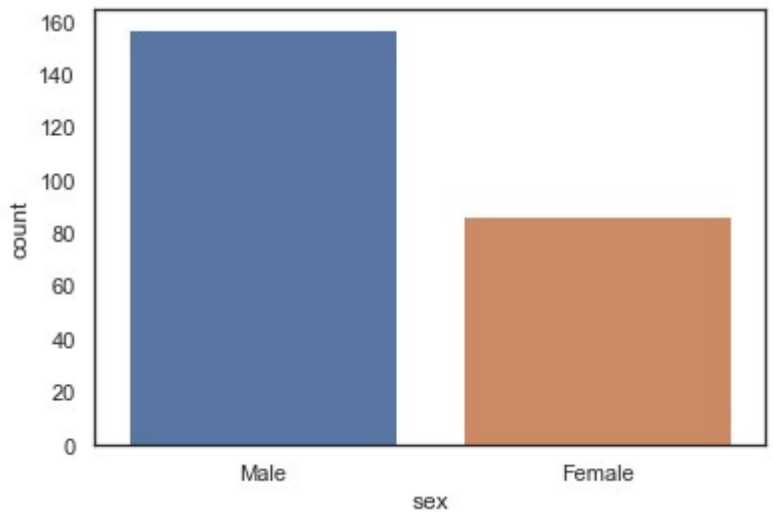
In [134]:
```
sns.countplot(x='sex',data=tips)
```

Out[134]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb2b0d8188>
```



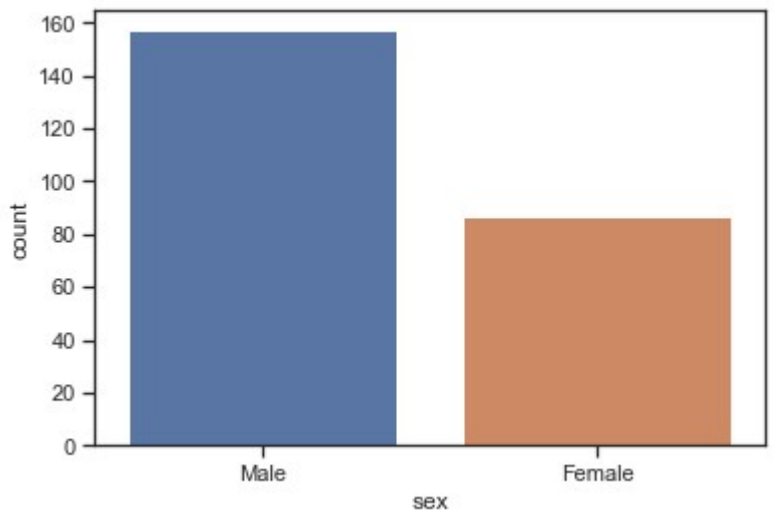
In [135]:
```
# Some designers actually make a collection of style of you, so you can just pick them:
sns.set_style('white')
sns.countplot(x='sex',data=tips)
```

Out[135]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb29502c88>
```

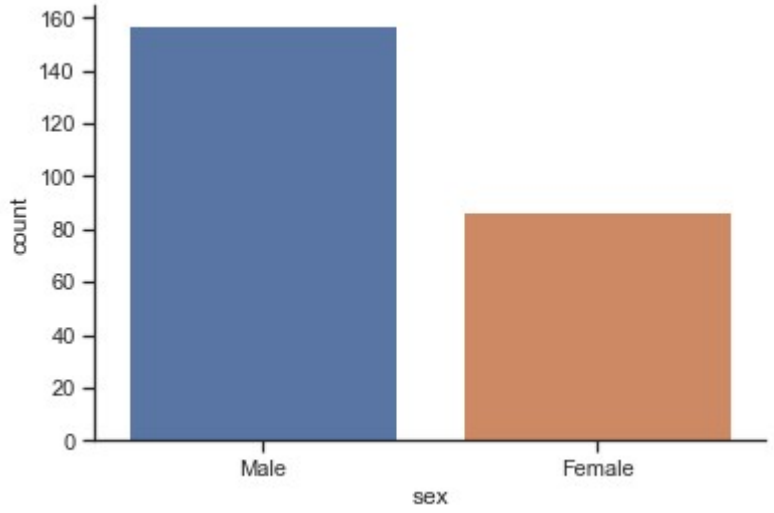


In [136]:
```
sns.set_style('ticks')
sns.countplot(x='sex',data=tips,palette='deep')
```

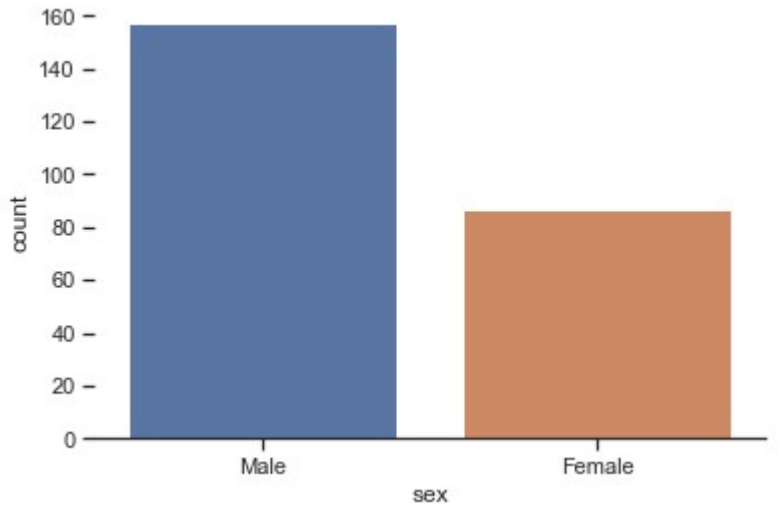
Out[136]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb29489cc8>
```



In [137]:
```
# Despine
sns.countplot(x='sex',data=tips)
sns.despine()
```

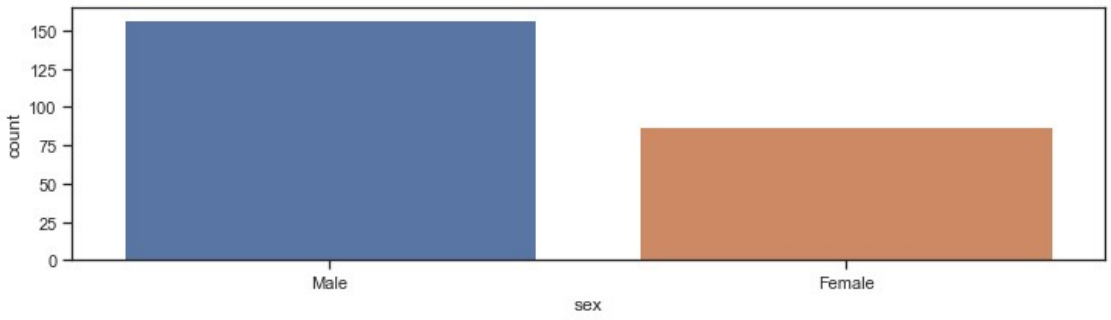


In [138]:
```
# Futher despine
sns.countplot(x='sex',data=tips)
sns.despine(left=True)
```



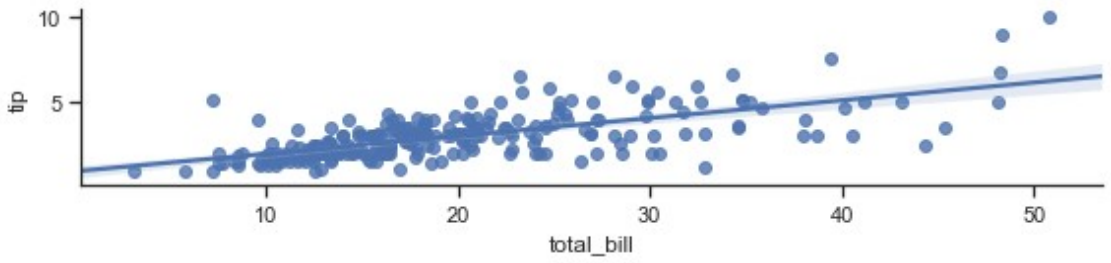
In [139]:
```
# Changing the size of a seaborn plot is done via matplotlib
plt.figure(figsize=(12,3))
sns.countplot(x='sex',data=tips)
```

Out[139]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cb26613bc8>
```



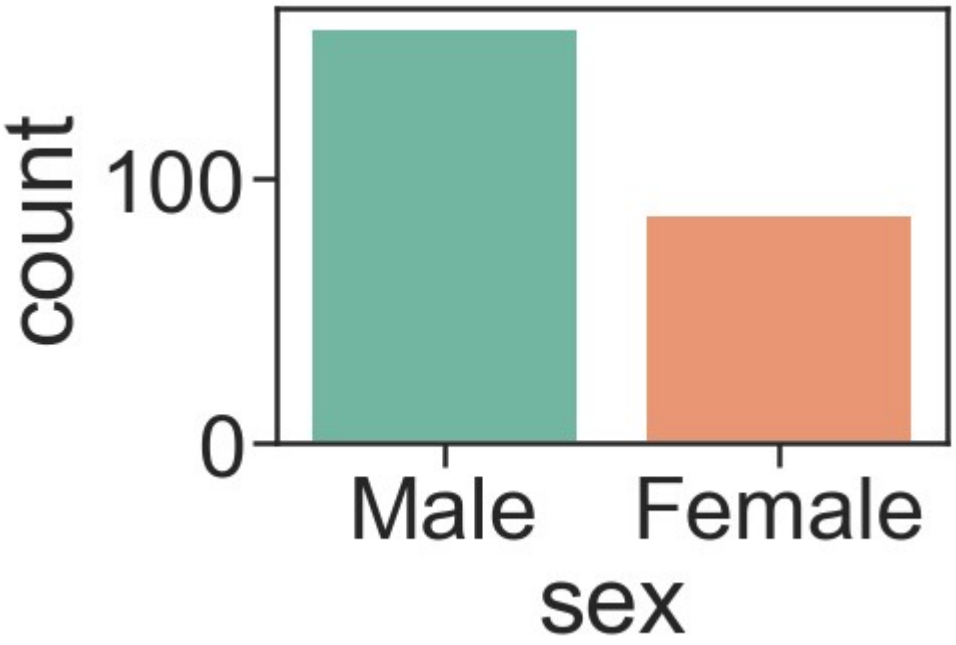
In [140]:
```
# However, some plot has height and aspect as arguments, such as the lm.plot above
sns.lmplot(x='total_bill',y='tip',size=2,aspect=4,data=tips)
```

Out[140]:
```
<seaborn.axisgrid.FacetGrid at 0x1cb26621908>
```



In [143]:
```
# If you want to quickly scale the figure, for example for a poster:
sns.set_context('poster',font_scale=2)
sns.countplot(x='sex',data=tips,palette='Set2')
```

Out[143]: <matplotlib.axes._subplots.AxesSubplot at 0x1cb24fc0788>



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

titanic = sns.load_dataset('titanic')
```

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

Out[3]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_n
0	0	3	male	22.0	1	0	7.2500	S	Third	man	
1	1	1	female	38.0	1	0	71.2833	C	First	woman	F
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	F
3	1	1	female	35.0	1	0	53.1000	S	First	woman	F
4	0	3	male	35.0	0	0	8.0500	S	Third	man	

In [4]:
```
# Let's start with what we know, groupby. We want to get the mean of
survival rate according to gender. We can:
titanic.groupby('sex')[['survived']].mean()
```

Out[4]:

	survived
sex	
female	0.742038
male	0.188908

In [5]:
```
# We now want to explore the survival rate much closer according to
class (ticket class).
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unst
ack()
```

Out[5]:

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

In [6]:
```
# You can do something like this or even without unstack():
titanic.groupby(['sex', 'class'])['survived'].mean().unstack()
```

Out[6]:

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

In [7]:
```
# Codes look a bit confusing. Pivot table is used for cases like thi
s
titanic.pivot_table('survived', index='sex', columns='class')
```

Out[7]:

sex	class First	Second	Third
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

Note that pivot table automatically take the mean for you by default. If you want a std() or min/max. You can declare the argument aggfunc= . Like this:

In [8]:
```
titanic.pivot_table('survived', index='sex', columns='class', aggfun
c='std')
```

Out[8]:

sex	class First	Second	Third
female	0.176716	0.271448	0.501745
male	0.484484	0.365882	0.342694

aggfunc can also be mapped like a dictionary!In this case, the data is unspecified

In [9]:
```
titanic.pivot_table(index='sex', columns='class', aggfunc={'survived
':'sum', 'fare':'mean'})
```

Out[9]:

	fare			survived		
class	First	Second	Third	First	Second	Third
sex						
female	106.125798	21.970121	16.118810	91	70	72
male	67.226127	19.741782	12.661633	45	17	47

Multilevel pivot table. Sometimes you might want to make a multilevel pivot table (multi-index). But some values must be defined in a bin first, such as age.

In [10]:
```
age = pd.cut(titanic['age'], [0,18,30,50,80])
titanic.pivot_table('survived', ['sex',age], 'class')
```

Out[10]:

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 30]	0.958333	0.900000	0.500000
	(30, 50]	0.972973	0.925926	0.272727
	(50, 80]	1.000000	0.666667	1.000000
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 30]	0.428571	0.027027	0.147541
	(30, 50]	0.448980	0.114286	0.126761
	(50, 80]	0.192308	0.083333	0.000000

In [12]:
```
fare = pd.qcut(titanic['fare'],2) #quantile cut, just cut into 2 quantile
titanic.pivot_table('survived', ['sex', age], [fare, 'class']) #Similar to normal dataframe rows, columns. So ['sex', age] is row, [fare, 'class'] is column
```

Out[12]:

	fare	(-0.001, 14.454]			(14.454, 512.329]		
	class	First	Second	Third	First	Second	Third
sex	age						
female	(0, 18]	NaN	1.000000	0.714286	0.909091	1.000000	0.318182
	(18, 30]	NaN	0.916667	0.500000	0.958333	0.888889	0.500000
	(30, 50]	NaN	0.916667	0.142857	0.972973	0.933333	0.333333
	(50, 80]	NaN	0.000000	1.000000	1.000000	1.000000	NaN
male	(0, 18]	NaN	0.000000	0.260870	0.800000	0.818182	0.178571
	(18, 30]	NaN	0.040000	0.138889	0.428571	0.000000	0.214286
	(30, 50]	0.0	0.176471	0.118644	0.488889	0.055556	0.166667
	(50, 80]	NaN	0.111111	0.000000	0.192308	0.000000	NaN

In [ ]:
```

```

In [59]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Remember to use plt.show() when you use pycharm!
%matplotlib inline
import statsmodels.api as sm
from statsmodels.iolib.summary2 import summary_col

from linearmodels.iv import IV2SLS
```

In [60]:
```
df = pd.read_csv('housing.csv')
```

Let's first import some data, you should download the data from Moodle, if you want a challenge with the unclean data you can try: https://www.kaggle.com/vedavyasv/usa-housing

Data Overview

The section below is for getting the overview of your data, you are usually not required to report these metrics or graphs in your term paper or master thesis for example.

In [61]:
```
df.head()
```

Out[61]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	
0	79545.458574	5.682861	7.009188	4.09	23086.800503	1.059034e+06	208 Michael F 674\nLaura
1	79248.642455	6.002900	6.730821	3.09	40173.072174	1.505891e+06	188 Johns Suite 0 Kathle
2	61287.067179	5.865890	8.512727	5.13	36882.159400	1.058988e+06	9127 Stravenue\nDa W
3	63345.240046	7.188236	5.586729	3.26	34310.242831	1.260617e+06	USS Barnett\
4	59982.197226	5.040555	7.839388	4.23	26354.109472	6.309435e+05	USNS Raymc A

In [62]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 7 columns):
Avg. Area Income                5000 non-null float64
Avg. Area House Age             5000 non-null float64
Avg. Area Number of Rooms       5000 non-null float64
Avg. Area Number of Bedrooms    5000 non-null float64
Area Population                 5000 non-null float64
Price                           5000 non-null float64
Address                         5000 non-null object
dtypes: float64(6), object(1)
memory usage: 273.6+ KB
```

In [63]:
```
df.describe()
```

Out[63]:

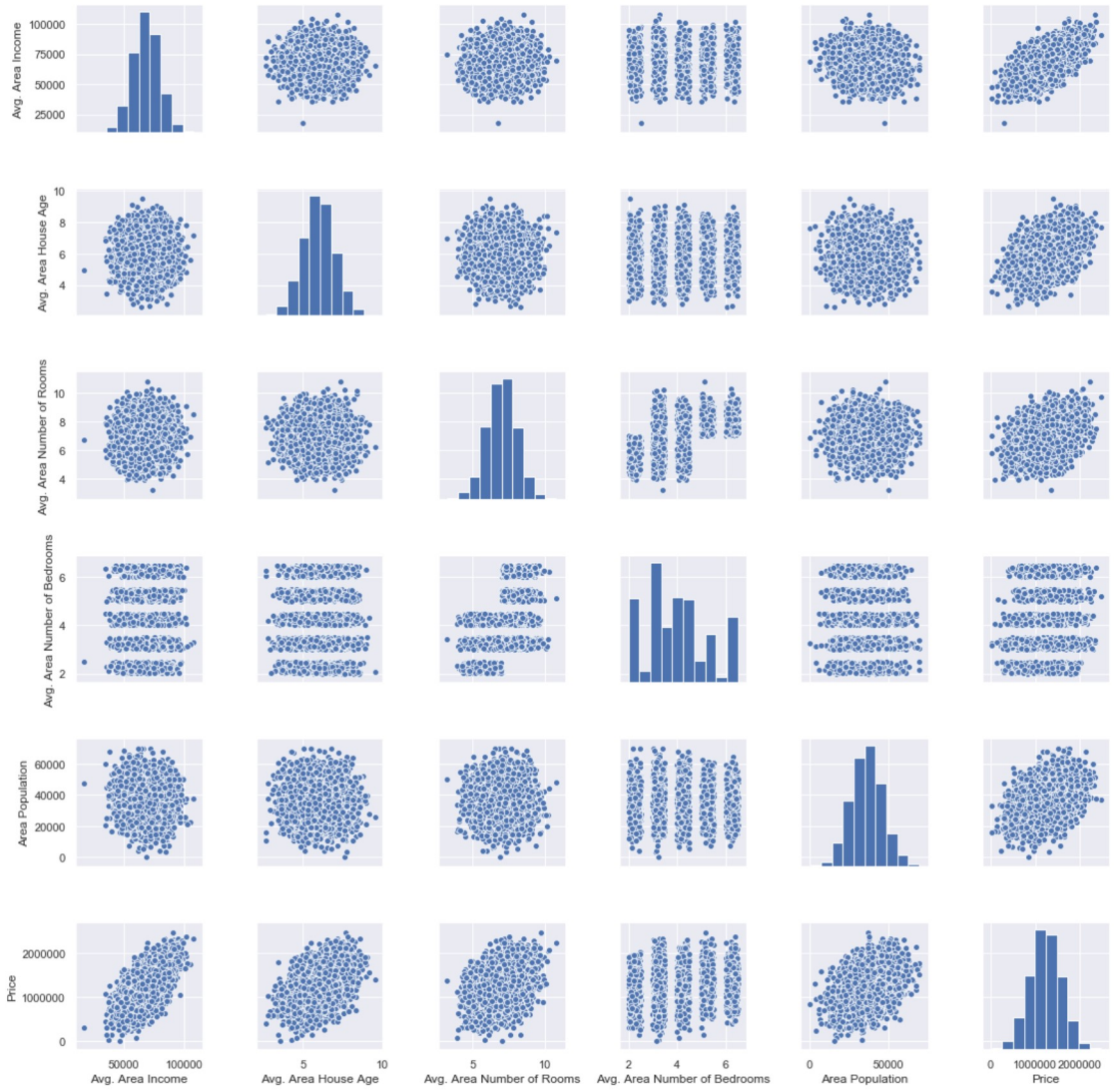
	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5.000000e+03
mean	68583.108984	5.977222	6.987792	3.981330	36163.516039	1.232073e+06
std	10657.991214	0.991456	1.005833	1.234137	9925.650114	3.531176e+05
min	17796.631190	2.644304	3.236194	2.000000	172.610686	1.593866e+04
25%	61480.562388	5.322283	6.299250	3.140000	29403.928702	9.975771e+05
50%	68804.286404	5.970429	7.002902	4.050000	36199.406689	1.232669e+06
75%	75783.338666	6.650808	7.665871	4.490000	42861.290769	1.471210e+06
max	107701.748378	9.519088	10.759588	6.500000	69621.713378	2.469066e+06

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

Out[64]:
```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Numbe
r of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population', 'Price',
'Address'],
      dtype='object')
```

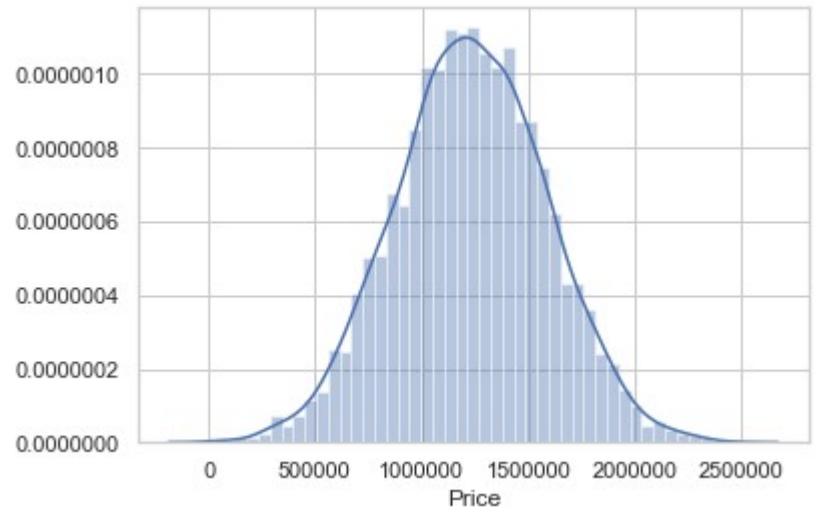
In [65]:
```
sns.pairplot(df)
```

Out[65]:
```
<seaborn.axisgrid.PairGrid at 0x2203dc1ae48>
```



In [66]:
```
sns.set_style("whitegrid")
sns.distplot(df['Price'])
```

Out[66]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2203eb16ec8>
```



If you see a warning in this step (in pink, error is in white). You should just ignore these warnings, when scipy is updating this warning will pop up but it is not at all a problem.

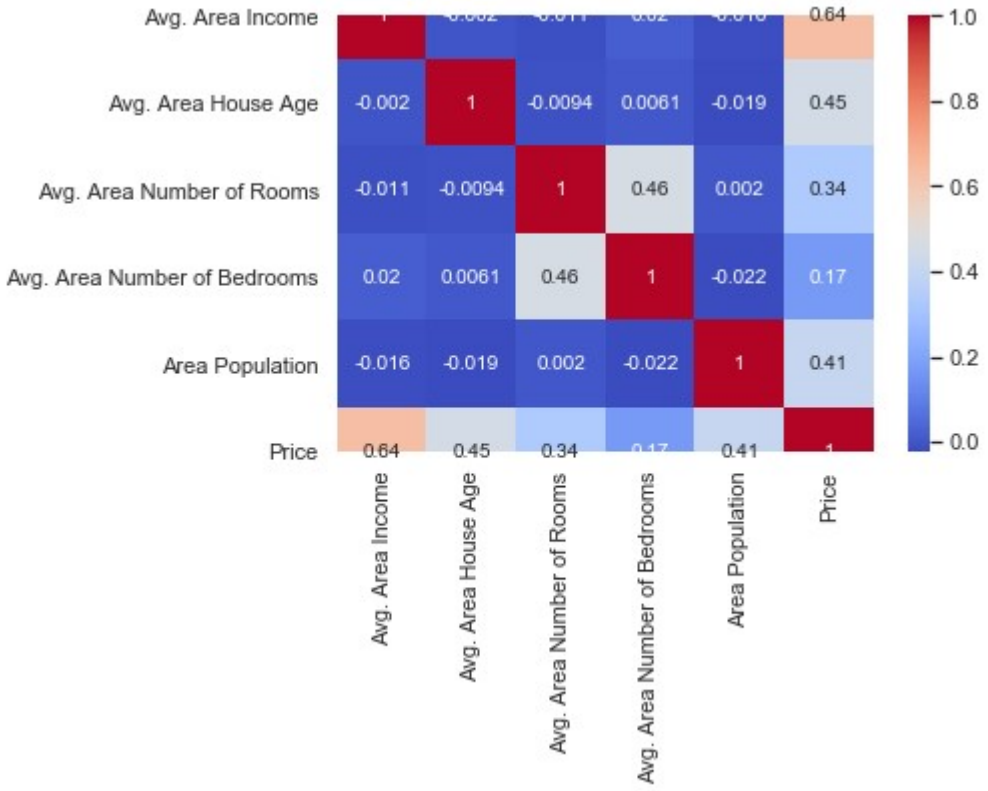
In [67]:
```
df.corr() #Shows correlation between all the columns
```

Out[67]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
Avg. Area Income	1.000000	-0.002007	-0.011032	0.019788	-0.016234	0.639734
Avg. Area House Age	-0.002007	1.000000	-0.009428	0.006149	-0.018743	0.452543
Avg. Area Number of Rooms	-0.011032	-0.009428	1.000000	0.462695	0.002040	0.335664
Avg. Area Number of Bedrooms	0.019788	0.006149	0.462695	1.000000	-0.022168	0.171071
Area Population	-0.016234	-0.018743	0.002040	-0.022168	1.000000	0.408556
Price	0.639734	0.452543	0.335664	0.171071	0.408556	1.000000

In [68]:
```
sns.heatmap(df.corr(), annot = True, cmap='coolwarm')
```

Out[68]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2203f050dc8>
```



In [69]:
```
df.columns
```

Out[69]:
```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Numbe
r of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population', 'Price',
'Address'],
      dtype='object')
```

In [70]:
```
X = df[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number
of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population']]
```

In [71]:
```
Y = df['Price']
```

Simple Linear Regression

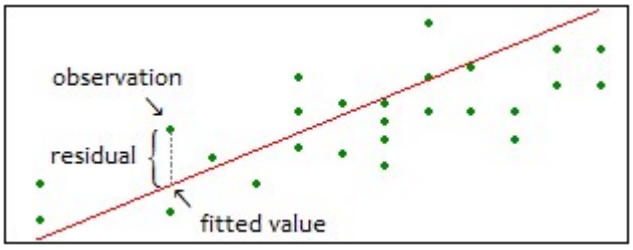
We'll use the Python package statsmodels to estimate, interpret, and visualize linear regression models.

You can also use sklearn to perform econometric applications, however, scikit-learn focuses on prediction (forecasting or training models) rather than explanation (which most econometricians focus on). Therefore, I will not be using scikit-learn in this lecture, for those who are interested, you can take a look here: https://scikit-learn.org/ (https://scikit-learn.org/) and here for linear regression https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html).

Again, the most important thing is the statistics (or econometrics) theories behind these models, it does not matter which tools you use to perform it. If you feel uncomfortable with the following materials, consider refreshing your knowledge in statistics or econometrics. These are just codes to implement statistical tools.

So let's get back to our data. From the above pairplot graph, it seems the 'Avg. Area Income' and the 'Price' has a good correlation. Let us check the linear regression of these two variables. The form of the model is: $y = \beta_0 + \beta x + \mu$ (β is the slope, β_0 is the intercept and μ is the random error term).

The goal of linear regression is to minimize the vertical distance between all the data points and the regression line. The most common one is the sum of least squared of residuals or OLS (ordinary least square). In this case, we're trying to draw a straight line so that the distance (μ) squared of each datapoint to this line is minimum (the least), hence the name " least squared of residuals".



To estimate β_0, we need to first add a constant variable columns with value of 1 into our data, you can think of it as $\beta_0 x_1$ with $x_1 = 1$.

In [72]:
```
df['dummy'] = 1
#We can divide by 1000 to only deal with thousand to make the number
looks smaller. This should not change the relationship between
#the variables in our models, i.e beta (which comprises of only 2 va
riables below)
#try not to do this step and see the differences yourself.
df['Avg. Area Income'] = df['Avg. Area Income']/1000
df['Price'] = df['Price']/1000
df.head()
```

Out[72]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Addı
0	79.545459	5.682861	7.009188	4.09	23086.800503	1059.033558	208 Michael Ferry 674\nLaurabury 37
1	79.248642	6.002900	6.730821	3.09	40173.072174	1505.890915	188 Johnson Vi Suite 079\nL Kathleen, (
2	61.287067	5.865890	8.512727	5.13	36882.159400	1058.987988	9127 Eliza Stravenue\nDanielt WI 064
3	63.345240	7.188236	5.586729	3.26	34310.242831	1260.616807	USS Barnett\nFPC 44
4	59.982197	5.040555	7.839388	4.23	26354.109472	630.943489	USNS Raymond\nI AE 09

Now we can run OLS regression using statsmodel package

In [170]:
```
# (*) specifying the model of choice, here I choose OLS.
reg_price_avgincome = sm.OLS(endog=df['Price'], exog=df[['dummy', 'A
vg. Area Income']], \
    missing='drop')
#Python allows you to specify if you want to drop NA values here, ho
wever, it is a good practice to deal with NA first
#(clean your data first!) before feeding your data into a model
type(reg_price_avgincome)
```

Out[170]:
```
statsmodels.regression.linear_model.OLS
```

This only specify our model $y = \beta_0 + \beta x + \mu$ or price = $\beta_0 + \beta^* avg\_area\_income + \mu$. In order to get the slope and intercept you have to use .fit() method

In [74]:
```
results_reg1 = reg_price_avgincome.fit() #this is when we actually try to calculate (fit) the numbers into our model
results_reg1.summary()
```

Out[74]:

OLS Regression Results

Dep. Variable:	Price	**R-squared:**	0.409
Model:	OLS	**Adj. R-squared:**	0.409
Method:	Least Squares	**F-statistic:**	3463.
Date:	Thu, 16 Jan 2020	**Prob (F-statistic):**	0.00
Time:	22:19:50	**Log-Likelihood:**	-35112.
No. Observations:	5000	**AIC:**	7.023e+04
Df Residuals:	4998	**BIC:**	7.024e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
dummy	-221.5795	25.000	-8.863	0.000	-270.591	-172.568
Avg. Area Income	21.1955	0.360	58.844	0.000	20.489	21.902

Omnibus:	0.527	**Durbin-Watson:**	1.976
Prob(Omnibus):	0.768	**Jarque-Bera (JB):**	0.572
Skew:	0.007	**Prob(JB):**	0.751
Kurtosis:	2.950	**Cond. No.**	452.

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Here are a few basic things we get from the summary table

- The intercept β_0= -221.5795 (thousands)
- The slope β is 21.1955 (wait! is this thousands ?)
- The positive β parameter estimate implies that the average income in the area has a positive effect on the housing prices in that area.
- The p-value of 0.000 for β implies that the effect of average income in the area is statistically significant (using 0.05 as a rejection rule).
- The R-squared value of 0.409 indicates that around 40.9% of variation in housing prices is explained by the average income in that area.

Our model now can be written as: $\widehat{price}$ = -221.5795 + 21.1955**avg_area_income*

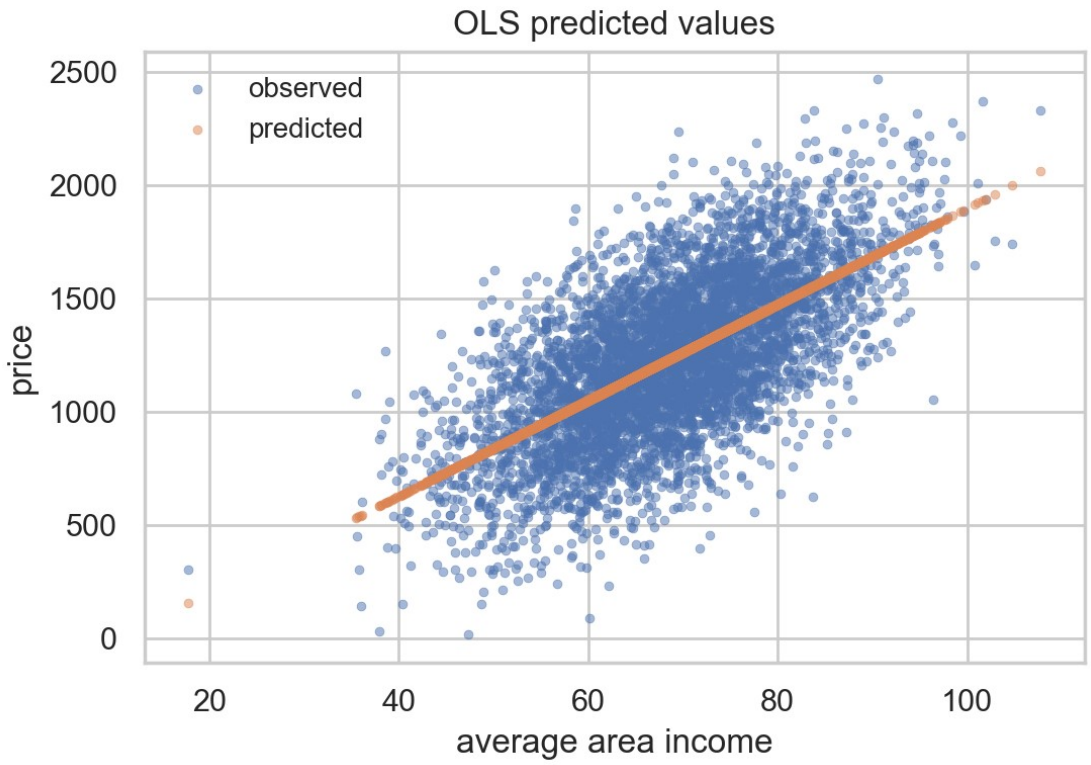
If you want to get a predicted price at any data point of average area income, you can use method .predict()

In [75]:
```
#I use the mean of the average area income as "the" data point of ch
oice here:
mean_avg_aincome = np.mean(df['Avg. Area Income'])
results_reg1.predict(exog=[1, mean_avg_aincome])
```

Out[75]:
```
array([1232.07265414])
```

We can also use .predict() to make a columns of predicted price value and plot it on the figure.

In [76]:
```
sns.set_style("whitegrid")
fig1, ax = plt.subplots(dpi=200)
plt.style.use('seaborn')

# Plot observed values

ax.scatter(df['Avg. Area Income'], df['Price'], alpha=0.5,
        label='observed', s=10)

# Plot predicted values
ax.scatter(df['Avg. Area Income'], results_reg1.predict(), alpha=0.
5,
        label='predicted', s=10)

ax.legend()
ax.set_title('OLS predicted values')
ax.set_xlabel('average area income')
ax.set_ylabel('price')
plt.show()
```



Multivariate linear regression

So far we were dealing with only 2 variables (bivariate), we can now try to extend our model by adding more variables (as an attempt to reduce omitted variable bias). Let us add all the the other variables into our models.

In [77]:
```
# Create lists of variables to be used in each model (regression).
X = df[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number
of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population']]

X1 = ['dummy', 'Avg. Area Income']
X2 = ['dummy', 'Avg. Area Income', 'Avg. Area House Age']
X3 = ['dummy', 'Avg. Area Income', 'Avg. Area House Age', 'Avg. Area
Number of Rooms', 'Avg. Area Number of Bedrooms', 'Area Population']

# Estimate an OLS regression for each set of variables
reg1 = sm.OLS(df['Price'], df[X1], missing='drop').fit()
reg2 = sm.OLS(df['Price'], df[X2], missing='drop').fit()
reg3 = sm.OLS(df['Price'], df[X3], missing='drop').fit()
```

We can now display all the summary tables into a single neatly formated table (in R, you might be using stargazer for this step instead).

In [78]:
```
metrics_dict={'R-squared' : lambda x: f"{x.rsquared:.2f}",
           'No. observations' : lambda x: f"{int(x.nobs):d}"}

results_table = summary_col(results=[reg1,reg2,reg3],
                            float_format='%0.2f',
                            stars = True,
                            model_names=['Model 1',
                                         'Model 3',
                                         'Model 4'],
                            info_dict=metrics_dict,
                            regressor_order=['dummy',
                                             'Avg. Area Income',
                                             'Avg. Area House Age',
                                             'Avg. Area Number of Ro
oms',
                                             'Avg. Area Number of Be
drooms',
                                             'Area Population'])

results_table.add_title('Table 2 - OLS Regressions')

print(results_table)
```

```
                   Table 2 - OLS Regressions
=================================================================
                               Model 1     Model 3     Model 4
-----------------------------------------------------------------
dummy                         -221.58*** -1189.78*** -2637.30***
                              (25.00)     (27.53)     (17.16)
Avg. Area Income              21.20***    21.23***    21.58***
                              (0.36)      (0.29)      (0.13)
Avg. Area House Age                       161.64***   165.64***
                                          (3.13)      (1.44)
Avg. Area Number of Rooms                             120.66***
                                                      (1.61)
Avg. Area Number of Bedrooms                          1.65
                                                      (1.31)
Area Population                                       0.02***
                                                      (0.00)
R-squared                     0.41        0.62        0.92
No. observations              5000        5000        5000
=================================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

Of course, these models are likely suffered from endogeneity. The reasons could be:

- There are still omitted variable bias, such as if the house located near a beach.
- The number of rooms and number of bedrooms are correlated (bedrooms are considered as rooms)
- Price and income influence each other. Higher-income earners choose to live in the higher-priced neighborhoods and higher price could attract higher income earners. (this is not a statement but a silly plausible theory)
- The data collection process is biased, only high price housing sellers answer the questionnaire.
 ...and many many more reasons...

To deal with this endogeneity issues, we can use two-stage least squares (2SLS) regression.

This method requires replacing the endogenous variable with a variable called *'instrument_income'* that is:

- correlated with Avg. Area Income
- not correlated with the error term or not directly affect the price (dependent variable)
(let assume that *'Avg. Area Income'* is the only main effect we are interested in here)

This variable 'instrument_income' as I named it, is an instrument. Let assume that such variable exists:

In [155]:
```
rng = np.random.RandomState(1)
df['random_noise'] = rng.normal(1,10,5000) #this step is important,
you should not directly put this in the fuction below
df['instrument_income']   =  0.5*df['Avg. Area Income'] - 5 + df['ra
ndom_noise']
df.head()
#this instrument is correlated with Avg. Area Income (1st condition
satisfied)
#since this is a made up instrument, it should not directly affect t
he housing price (2nd condition statisfied)
```

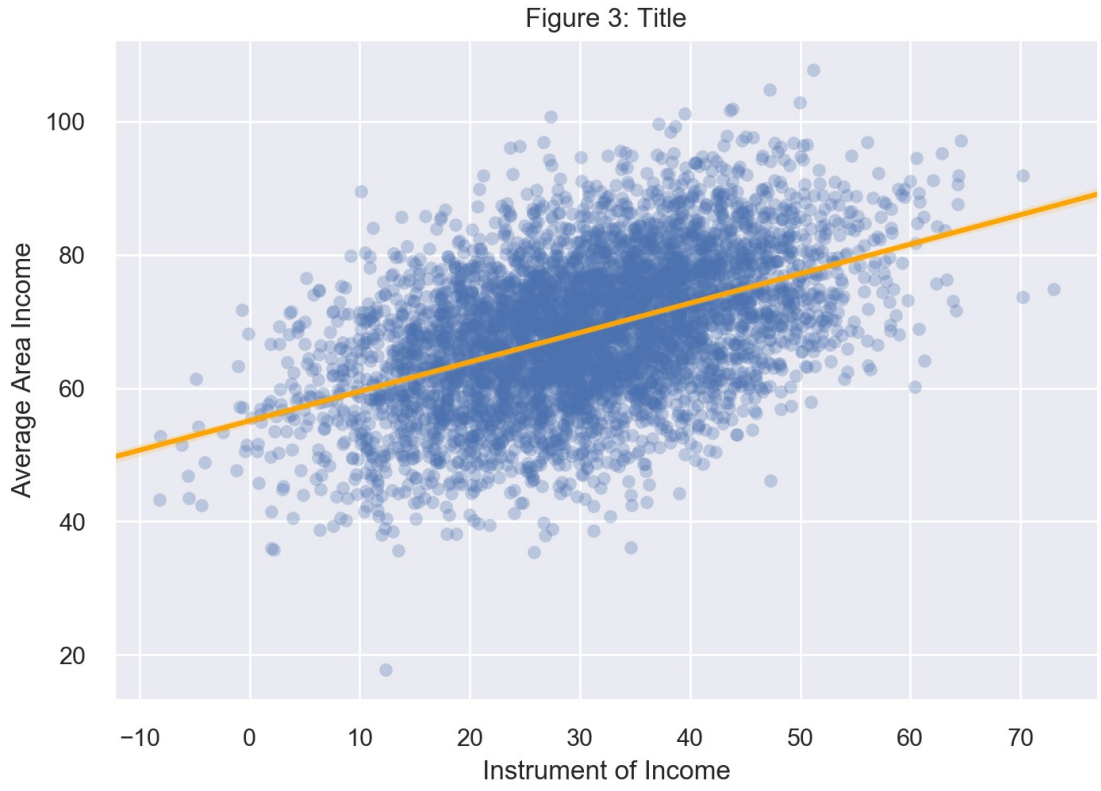
Out[155]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Addı
0	79.545459	5.682861	7.009188	4.09	23086.800503	1059.033558	208 Michael Ferry 674\nLaurabury 37
1	79.248642	6.002900	6.730821	3.09	40173.072174	1505.890915	188 Johnson Vi Suite 079\nL Kathleen, (
2	61.287067	5.865890	8.512727	5.13	36882.159400	1058.987988	9127 Eliza Stravenue\nDanielt WI 064
3	63.345240	7.188236	5.586729	3.26	34310.242831	1260.616807	USS Barnett\nFPC 44
4	59.982197	5.040555	7.839388	4.23	26354.109472	630.943489	USNS Raymond\nI AE 0

First stage

Let us now create a first stage regression using this made-up variable

In [156]:
```
sns.set()
X = df['instrument_income']
y = df['Avg. Area Income']
fig, ax = plt.subplots(dpi=200)
# Fit a linear trend line
sns.regplot(x= X, y= y, ax= ax, data=df, scatter_kws={'alpha':0.3},
line_kws={'color':'orange'})

ax.set_xlabel('Instrument of Income')
ax.set_ylabel('Average Area Income')
ax.set_title('Figure 3: Title');
```



In [157]:
```
# Similar to the case above with OLS, we need to create a dummy for
the intercept
# You can of course use 1 dummy since it's a constant dummy,
# but to make it clear here I use another identical column of dummy
df['dummy2'] = 1

# Fit the first stage regression and print summary
results_fs = sm.OLS(df['Avg. Area Income'],
                    df[['dummy2', 'instrument_income']],
                    missing='drop').fit()
print(results_fs.summary())
```

```
                            OLS Regression Results
==============================================================================
============
Dep. Variable:       Avg. Area Income   R-squared:
0.221
Model:                            OLS   Adj. R-squared:
0.221
Method:                 Least Squares   F-statistic:
1420.
Date:                Thu, 16 Jan 2020   Prob (F-statistic):
9.23e-274
Time:                        22:48:01   Log-Likelihood:
-18301.
No. Observations:                5000   AIC:
3.661e+04
Df Residuals:                    4998   BIC:
3.662e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
==================
                         coef    std err          t      P>|t|
[0.025      0.975]
------------------------------------------------------------------------------
------------------
dummy2                55.1100      0.381    144.466      0.000
54.362      55.858
instrument_income      0.4412      0.012     37.684      0.000
0.418       0.464
==============================================================================
============
Omnibus:                        1.099   Durbin-Watson:
2.069
Prob(Omnibus):                  0.577   Jarque-Bera (JB):
1.047
Skew:                          -0.029   Prob(JB):
0.592
Kurtosis:                       3.040   Cond. No.
93.5
==============================================================================
============

Warnings:
[1] Standard Errors assume that the covariance matrix of the error
s is correctly specified.
```

Second stage

We need to retrieve the predicted values of 'Avg. Area Income' using .predict().

We then replace the endogenous variable 'Avg. Area Income' with the predicted values $\widehat{Avg.AreaIncome}$ in the original linear model. (I admit here I should have changed all the variable names, it is not recommended to use white space in variable name, you can change column name by using: df.rename(columns = {'old_name_col1':'new_name_col1'})

Our second stage regression is thus: price = β_0 + $\beta^* \widehat{Avg.AreaIncome}$ + μ

In [158]:
```
df['predicted_ava_income'] = results_fs.predict()

results_ss = sm.OLS(df['Price'],
                    df[['dummy2', 'predicted_ava_income']]).fit()
print(results_ss.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Price   R-squared:
0.094
Model:                            OLS   Adj. R-squared:
0.094
Method:                 Least Squares   F-statistic:
521.6
Date:                Thu, 16 Jan 2020   Prob (F-statistic):
6.75e-110
Time:                        22:59:10   Log-Likelihood:
-36180.
No. Observations:                5000   AIC:
7.236e+04
Df Residuals:                    4998   BIC:
7.238e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                             coef    std err          t      P>|t|
[0.025      0.975]
------------------------------------------------------------------------------
dummy2                   -252.9088     65.194     -3.879      0.000
-380.718    -125.099
predicted_ava_income       21.6523      0.948     22.839      0.000
19.794      23.511
==============================================================================
Omnibus:                        0.312   Durbin-Watson:
2.002
Prob(Omnibus):                  0.855   Jarque-Bera (JB):
0.353
Skew:                          -0.010   Prob(JB):
0.838
Kurtosis:                       2.965   Cond. No.
944.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the error
s is correctly specified.
```

The second-stage regression results give us an unbiased and consistent estimate of the effect of Avg. Area Income on housing price.

The result suggests a slightly stronger positive relationship than what the OLS results indicated. Of course, this is only a made up example.

Note that while our parameter estimates are correct, computing 2SLS ‘by hand’ (in stages with OLS) is not recommended. Because the second stage will yield the “wrong” residuals (being computed from the instruments rather than the original variables), which implies that all statistics computed from those residuals will be incorrect (the estimated standard errors of the parameters, etc.)

We can correctly estimate a 2SLS regression in one step using the linearmodels package, an extension of statsmodels

Note that when using IV2SLS, the exogenous and instrument variables are split up in the function arguments (whereas before the instrument included exogenous variables)

In [162]:
```
iv_reg = IV2SLS(dependent=df['Price'],
            exog=df['dummy2'],
            endog=df['Avg. Area Income'],
            instruments=df['instrument_income']).fit(cov_type='unadj
usted')

print(iv_reg.summary)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
============
Dep. Variable:                  Price   R-squared:
0.4091
Estimator:                     IV-2SLS   Adj. R-squared:
0.4090
No. Observations:                5000   F-statistic:
799.58
Date:                Thu, Jan 16 2020   P-value (F-stat)
0.0000
Time:                        23:56:37   Distribution:
chi2(1)
Cov. Estimator:            unadjusted

                                Parameter Estimates
==============================================================================
==================
                   Parameter  Std. Err.     T-stat    P-value    Lo
wer CI    Upper CI
------------------------------------------------------------------------------
------------------
dummy2              -252.91     52.656    -4.8030     0.0000     -
356.11     -149.71
Avg. Area Income     21.652     0.7657     28.277     0.0000
20.151      23.153
==============================================================================
==================

Endogenous: Avg. Area Income
Instruments: instrument_income
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

statsmodels should be enough different types of models for your econometric applications. What you should get from this simple linear regression example is the process to get the summary table you want. If you want a different model (says probit or logit) you can look it up in https://www.statsmodels.org/ (https://www.statsmodels.org/) and replace it in step 1 *(*) specifying the model of choice*. Python and statsmodels makes it easy for you to implement many statistical models, you can check them here: https://www.statsmodels.org/dev/regression.html (https://www.statsmodels.org/dev/regression.html).

A very short primer on sklearn (a linear regression case)

In [163]:
```
from sklearn.linear_model import LinearRegression
lm = LinearRegression(fit_intercept=True) #this is object-oriented s
tyle
y = df['Price']
x = df['Avg. Area Income']
```

In [164]:
```
lm.fit(x[:, np.newaxis], y)
```

Out[164]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, nor
malize=False)
```

In [165]:
```
lm.coef_[0]
```

Out[165]:
```
21.19548317193168
```

In [167]:
```
lm.intercept_
```

Out[167]:
```
-221.57947820591812
```

There exists no regression summary report in sklearn (like in R or statsmodels in Python). If you want some metrics you have to get it one by one like the above. The reason is that sklearn focuses on predictive analysis, while in economics we are focusing on the "traditional" statistical approach (explanatory). The evaluation criteria in the predictive analysis are different than explanatory analysis.