PYTHON FOR DATA ANALYSIS

Content from Jose Portilla's Udemy course *Learning Python for Data Analysis and Visualization*
https://www.udemy.com/learning-python-for-data-analysis-and-visualization/

Notes by Michael Brothers, available on http://github.com/mikebrothers/data-science/

Table of Contents

LEARNING PYTHON FOR DATA ANALYSIS & VISUALIZATION Udemy course by Jose Portilla (notes by Michael Brothers)

What's What:
Numpy – fundamental package for scientific computing, working with arrays
Pandas – create high-performance data structures, Series, Data Frames. incl built-in visualization, file reading tools
Matplotlib – data visualization package
Seaborn Libraries – heatmap plots et al
Beautiful Soup – a web-scraping tool
SciKit-Learn – machine learning library

Skills:
Importing data from a variety of formats: JSON, HTML, text, csv, Excel
Data Visualization – using Matplotlib and the Seaborn libraries
Portfolio – set up a portfolio of data projects on GitHub
Machine Learning – using SciKit Learn

Resources:
stock market analysis (access Yahoo finance using pandas datareader)
FDIC list of failed banks (pull data from html)
Kaggle Titanic data set
political election data set
http://www.data.gov (home of the US Government's open data)
http://AWS.amazon.com/public-data-sets/ (Amazon web services public data sets)
http://www.google.com/publicdata/directory
create personal accounts on GitHub and Kaggle

Appendix Materials:
Statistics – includes using SciPy to create distributions & solve statistics problems
SQL with Python – includes using SQLAlchemy to fully integrate SQL with Python to run SQL queries from a Python environment. Also performing basic SQL commands with Python and pandas.
Web Scraping with Python – using Python web requests and the Beautiful-Soup library to scrape the web for data

For Further Reading:
Numpy: http://docs.scipy.org/doc/numpy/reference/
Numpy Universal Functions (ufuncs): http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs
Numpy supplemental materials: http://cs231n.github.io/python-numpy-tutorial/

Philosophy:
What's the difference between a Series, a DataFrame and an Array? (answers by Jose Portilla)
A **NumPy Array** is the basic data structure holding the data itself and allowing you to store and get elements from it.
A **Series** is built on top of an array, allowing you to label the data and index it formally, as well as do other pandas related Series operations.
A **DataFrame** is built on top of Series, and is essentially many series put together with different column names but sharing the same index.
Also, a 1-d numpy array is **not a list**. A list is a built-in data structure in regular Python, a numpy array is an object type only available once you've set up numpy. It is able to perform operations much faster than a list due to built-in optimizations.
Arrays are NumPy data types while Series and DataFrame are Pandas data types. They have different available methods and attributes.

NUMPY

```
import numpy as np
```
do this for every new Jupyter notebook

Creating Arrays

```
my_list1 = [1, 2, 3, 4]
my_array1 = np.array(my_list1)
my_array1
array([1, 2, 3, 4])
```
creates a 1-dimensional array from a list

```
my_list2 = [11, 22, 33, 44]
my_lists = [my_list1, my_list2]
my_array2 = np.array(my_lists)
my_array2
array([[ 1,  2,  3,  4],
       [11, 22, 33, 44]])
```
creates a multi-dimensional array from a list of lists

```
array_2d = (([1,2,3], [4,5,6]))
```
creating from scratch requires two sets of parentheses!

```
my_array2.shape
(2L, 4L)
```
describes the size & shape of the array (rows, columns)

```
my_array2.dtype
dtype('int32')
```
describes the data type of the array

Special Case Arrays

```
np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

```
np.ones((4,4))
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

```
np.empty(5)
np.empty((3,4))
```
resemble zeros arrays

```
np.eye(5)
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.]])
```
called the "identity array"

`dtype('float64')` for the above arrays

```
np.arange([start,] stop[, step])
np.arange(5,10,2)
array([5, 7, 9])
```
uses a range

Using Arrays and Scalars

```
from __future__ import division
arr1 = np.array([[1,2,3], [8,9,10]])
arr1
array([[ 1,  2,  3],
       [ 8,  9, 10]])
```
if running Python v2

note the double parentheses/brackets

<u>Adding arrays:</u>

```
arr1+arr1
array([[ 2,  4,  6],
       [16, 18, 20]])
```

<u>Multiplying arrays:</u>

```
arr1*arr1
array([[  1,   4,   9],
       [ 64,  81, 100]])
```

<u>Subtracting arrays:</u>

```
arr1-arr1
array([[0, 0, 0],
       [0, 0, 0]])
```

<u>Dividing arrays:</u> (Float return)

```
arr1/arr1
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Arithmetic operations with scalars on arrays:

```
1 / arr1
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.125     ,  0.11111111,  0.1       ]])

arr1**3
array([[   1,    8,   27],
       [ 512,  729, 1000]])
```

Indexing Arrays

Arrays are sequenced. They are modified in place by slice operations.

```
arr = np.arange(11)
arr
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

slice_of_arr = arr[0:6]
slice_of_arr
array([0, 1, 2, 3, 4, 5])

slice_of_arr[:]=99      change the slice
slice_of_arr
array([99, 99, 99, 99, 99, 99])

arr
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Note that the changes *also* occur in our original array.

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

```
arr_copy = arr.copy()  To get a copy, you need to be explicit
arr_copy
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Indexing a 2D Array

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

format follows `arr_2d[row][col]` or `arr_2d[row,col]`

```
arr_2d[1]                grab a row
array([20, 25, 30])

arr_2d[1][0] or arr_2d[1,0]     grab an individual element
20
```

Slicing a 2D Array

```
arr_2d[:2,1:]            grab a 2x2 slice from top right corner
array([[10, 15],
       [25, 30]])
```

Fancy Indexing

```
arr
array([[  0.,   10.,   20.,   30.,   40.],
       [  1.,   11.,   21.,   31.,   41.],
       [  2.,   12.,   22.,   32.,   42.]])
```

```
arr[[2,1]]
array([[  2.,   12.,   22.,   32.,   42.],
       [  1.,   11.,   21.,   31.,   41.]])
```

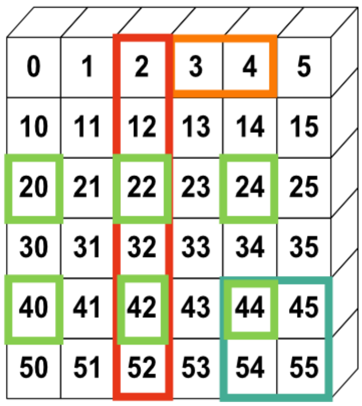
fancy indexing allows a selection of rows in any order using embedded brackets (note that `arr[2,1]` returns 12.0)

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

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

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

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



Source: http://www.scipy-lectures.org/_images/numpy_indexing.png

Array Transposition

```
arr = np.arange(24).reshape((4,6))
```

create an array

```
arr
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]])
```

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

transpose the array (this does NOT change the array in place)

```
np.dot(arr.T,arr)
array([[504, 540, 576, 612, 648, 684],
       [540, 580, 620, 660, 700, 740],
       [576, 620, 664, 708, 752, 796],
       [612, 660, 708, 756, 804, 852],
       [648, 700, 752, 804, 856, 908],
       [684, 740, 796, 852, 908, 964]])
```

take the **dot product** of these two arrays

```
504=(0*0)+(6*6)+(12*12)+(18*18)
540=(0*1)+(6*7)+(12*13)+(18*19)
```

See https://www.mathsisfun.com/algebra/matrix-multiplying.html for a simple explanation of dot products!

You can also transpose a 3D matrix:

```
arr3d = np.arange(18).reshape((3,3,2))
```

arr3d	arr3d.transpose((1,0,2))

array([[[0, 1],
        [2, 3],
        [4, 5]],
 	
       [[6, 7],
        [8, 9],
        [10, 11]],
 	
       [[12, 13],
        [14, 15],
        [16, 17]]])
	array([[[0, 1],
	        [6, 7],
	        [12, 13]],
	
	       [[2, 3],
	        [8, 9],
	        [14, 15]],
	
	       [[4, 5],
	        [10, 11],
	        [16, 17]]])

If you need to get more specific use **swapaxes**:

```
arr = np.array([[1,2,3]])
arr
array([[1, 2, 3]])
arr.swapaxes(0,1)
array([[1],
       [2],
       [3]])
```

Universal Array Functions

```
arr = np.arange(6)
arr
array([0, 1, 2, 3, 4, 5])
np.sqrt(arr)                          square-root function
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ])

np.exp(arr)                           exponential (e^)
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
```

Binary Functions (require two arrays):

```
np.add(A,B)                 returns sum of matching values of two arrays
np.maximum(A,B)             returns maximum between matching values of two arrays
```

Random number generator:

```
np.random.randn(10)         random array (normal distribution)
array([-0.10313268,  1.05811992, -1.98543659, -0.43591721,  0.03393424,
       -1.15738081, -0.35316064,  1.12707714, -0.09061522,  0.28226307])
```

For full and extensive list of all universal functions

```
website = "http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs"
import webbrowser
webbrowser.open(website)      conveniently opens site from within Jupyter notebook!
```

Array Processing

```
import numpy as np
import matplotlib.pyplot as plt
```
import the pyplot libraries from matplotlib which let us visualize the grids & meshes we'll be making

```
%matplotlib inline
```
this lets us see these visualizations in Jupyter notebooks

Using matplotlib.pyplot for visualization

```
points = np.arange(-5,5,0.01)
```
creates a 1-d array with 1000 data points

```
dx,dy=np.meshgrid(points,points)
```
creates a grid (returns coordinate matrices from the vectors we give it)

```
dx
```
these are our rows:

```
array([[-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       ...,
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])
```

```
dy
```
these are our columns: (note that values increase downward)

```
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

```
z = (np.sin(dx) + np.sin(dy))
```
this is just an evaluating function

```
z
array([[  1.91784855e+00,   1.92063718e+00,   1.92332964e+00, ...,
         -8.07710558e-03,  -5.48108704e-03,  -2.78862876e-03],
       [  1.92063718e+00,   1.92342581e+00,   1.92611827e+00, ...,
         -5.28847682e-03,  -2.69245827e-03,  -5.85087534e-14],
       [  1.92332964e+00,   1.92611827e+00,   1.92881072e+00, ...,
         -2.59601854e-03,  -5.63993297e-14,   2.69245827e-03],
       ...,
       [ -8.07710558e-03,  -5.28847682e-03,  -2.59601854e-03, ...,
         -1.93400276e+00,  -1.93140674e+00,  -1.92871428e+00],
       [ -5.48108704e-03,  -2.69245827e-03,  -5.63993297e-14, ...,
         -1.93140674e+00,  -1.92881072e+00,  -1.92611827e+00],
       [ -2.78862876e-03,  -5.85087534e-14,   2.69245827e-03, ...,
         -1.92871428e+00,  -1.92611827e+00,  -1.92342581e+00]])
```

```
plt.imshow(z);
```
plot the array (semicolon avoids extra Out line)

red/blue colored plot for evaluating function, y-axis from 1000-0, x-axis from 0-1000

```
plt.colorbar();
```
needs to be in same cell as plt.imshow(z)

adds vertical colorbar to right of plot (red 2.0 to blue -2.)

```
plt.title("Plot for sin(x)+sin(y)");
```
adds title above plot SEE PLOT IN SEPARATE FILE: PythonDataVisualizations.pdf

Using numpy.where

```
A = np.array([1,2,3,4])
B = np.array([100,200,300,400])
condition = np.array([True,True,False,False])
```
a Boolean array

The slow way: *Using a list comprehension*

```
answer1 = [(A_val if cond else B_val) for A_val,B_val,cond in zip(A,B,condition)]
answer1
[1, 2, 300, 400]
```
Problems include speed issues and multi-dimensional array issues

The numpy.where way:

```
answer2 = np.where(condition,A,B)
```
follows (test, if true, if false)

```
answer2
array([  1,   2, 300, 400])
```

```
Using numpy.where for 2D manipulation:
from numpy.random import randn
arr = randn(5,5)
np.where(arr < 0,0,arr)
```
Where array is less than zero, make that value zero, otherwise leave as is

```
array([[ 0.45983701,  0.        ,  1.56891548,  0.        ,  1.61030401],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.96909611,  0.        ,  0.        ,  0.69907836,  1.41859086],
       [ 0.        ,  1.42554561,  1.30200218,  1.77784525,  0.8120543 ],
       [ 1.39031869,  0.14319058,  0.11438954,  0.        ,  0.        ]])
```

More statistical tools:

```
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
arr.sum() returns 45
arr.sum(0) returns array([12,15,18])
```
sums along vertical axes

```
arr.mean() returns 5.0
```
Note there are no "median" or "mode" functions

```
arr.var() returns 6.666666666666667
```
variance

```
arr.std() returns 2.5819888974716112
```
standard deviation

Any and all for processing Boolean arrays:

```
bool_arr = np.array([True,False,True])
bool_arr.any() returns True
bool_arr.all() returns False
```

Sort, Unique and In1d:

```
arr = randn(5,5)
arr.sort()
```
sorts each row individually, in place

```
np.apply_along_axis(sorted, 0, arr)
```
sorts each item *horizontally*

```
countries = np.array(['France', 'Germany', 'USA', 'Russia', 'USA', 'Mexico'])
np.unique(countries)
array(['France', 'Germany', 'Mexico', 'Russia', 'USA'],
      dtype='|S7')

np.in1d(['France','USA','Sweden'],countries)
array([ True,  True, False], dtype=bool)
```

Array Input and Output

```
import numpy as np
```

Insert an element into an array (see http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.insert.html)

```
a = np.array([[1, 1], [2, 2], [3, 3]])
a
array([[1, 1],
       [2, 2],
       [3, 3]])
np.insert(a, 1, 5)
```
inserts a 5 before index 1 and *flattens* the array (but not in-place!)
```
array([1, 5, 1, 2, 2, 3, 3])
np.insert(a, 1, 5, axis=1)
```
inserts a 5 before index 1 along the vertical axis (but not in-place!)
```
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
```

Saving an array to a binary (.npy) file

```
arr = np.arange(5)
np.save('my_array',arr)          saves the array on disk in binary format (file extension .npy)
arr = np.arange(10)              here we create a different array with the same name
np.load('my_array.npy')          here we load the first array we created
array([0, 1, 2, 3, 4])
```

Saving multiple arrays into a zip (.npz) file

```
np.savez('two_arrays.npz',x=arr,y=arr)        saves 2 copies of arr to one file
```

Loading multiple arrays:

```
archive_array = np.load('two_arrays.npz')     Note: .load works for binary and zip
archive_array['x']                            calls the first array from the file
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Saving and loading text files

```
arr = np.array([[1,2,3],[4,5,6]])
np.savetxt('my_test_text.txt',arr,delimiter=',')

arr = np.loadtxt('my_test_text.txt',delimiter = ',')
arr
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

PANDAS

```
import numpy as np
import pandas as pd
from pandas import Series,DataFrame
```
this saves us from typing 'pd.Series' and 'pd.DataFrame' each time

WORKING WITH SERIES

Creating a Series (an array of data values and their index)

```
obj = Series([3,6,9,12])
obj
0      3
1      6
2      9
3     12
dtype: int64
```

`obj.values` shows the values

```
array([ 3,   6,   9, 12], dtype=int64)
```

`obj.index` shows the index (See section on Index Objects for passing an index to a new object)

```
Int64Index([0, 1, 2, 3], dtype='int64')
```

Creating a Series with a named index

```
coins = Series([.01,.05,.10,.25],index=['penny','nickel','dime','quarter'])
coins
penny      0.01
nickel     0.05
dime       0.10
quarter    0.25
dtype: float64
```

`coins['dime']` returns `0.10` *(actually it returns 0.10000000000000001)*

```
coins[coins>.07]
dime       0.10
quarter    0.25
dtype: float64
```

`'penny' in coins` returns `True` (although 0.25 in coins returns False)

Converting a Series to a Python dictionary

```
coin_dict = coins.to_dict()
coin_dict
{'dime': 0.10000000000000001,
 'nickel': 0.050000000000000003,
 'penny': 0.01,
 'quarter': 0.25}
```

gotta find out how to fix this…

`coins2 = Series(coin_dict)` converts it back, *but not in the same order as the original!*

Passing an index with the dictionary can reload a Series in order:

```
coinlabels = ['penny','nickel','dime','quarter','SBAnthony']
coins3 = Series(coin_dict,index=coinlabels)
```

converts it back *in index order*

note that 'SBAnthony' shows 'NaN' as its value

Use isnull and notnull to find missing data

`pd.isnull(coins3['SBAnthony'])` returns `True`

`pd.notnull(coins3['penny'])` returns `True`

Adding two Series together

`series1 + series2`   adds items by index, including null-value items

Labeling Series Indexes

`coins3.index.name = 'Coins'` puts a label above the index list *(.values does not have a name method)*

Checking for Unique Values and their Counts

```
ser1 = Series(list('abacab'))
```

`ser1.unique()` returns `array(['a', 'b', 'c'], dtype=object)`

`ser1.value_counts()` returns   see the DataFrames section on value_counts for more info

```
a    3
b    2
c    1
dtype: int64
```

Rank and Sort

Sort by Index Name using .sort_index:

```
ser1 = Series(range(3),index=['C','A','B'])
```

`ser1.sort_index()` returns ser1, but in index order (A:1,B:2,C:0)

Note: this does NOT sort `ser1` in place.

For that use either `ser1 = ser1.sort_index()` or `ser1.sort_index(inplace=True)`

Sort by Value using .sort_values:

`ser1.sort_values()` returns ser1, but in value order (C:0,A:1,B:2)

*Note: **.order** works, but throws a "FutureWarning: order is deprecated, use sort_values(…)" As above,*
use either `ser1 = ser1.sort_index()` or `ser1.sort_index(inplace=True)` to sort in place

Rank

`ser1.rank()` returns an integer rank from `1` to `len(ser1)` for each index (low to high)

NOTE: in the case of ties, .rank returns floats (1, 2.5, 2.5, 4)

WORKING WITH DATAFRAMES

For more info: http://pandas.pydata.org/pandas-docs/stable/dsintro.html#dataframe

Creating a DataFrame

```
import numpy as np
import pandas as pd
from pandas import Series,DataFrame
dframe = DataFrame(np.arange(12).reshape(4,3))
```

dframe is constructed by casting a 4-row by 3-col numpy array as a pandas DataFrame, pre-filled with values 0-11. Here the index defaults to [0,1,2,3], the columns to [0,1,2]

Constructing a DataFrame from a Dictionary:

```
data = {'City':['SF','LA','NYC'],'Population':[837000,3880000,8400000]}
city_frame = DataFrame(data)
```

Creates a DataFrame with columns labeled City and Population, indexes of [0,1,2]

Adding a Series to an existing DataFrame:

```
colors = Series(["Blue","Red"],index=[4,1])
dframe['Color']=colors
```

dframe now has a Color column with Blue matched to index 4, Red to 1, and NaN after everything else.

Reading a DataFrame from a webpage (using edit/copy):

Grab NFL Win-Loss data from Wikipedia:

```
import webbrowser
website = 'http://en.wikipedia.org/wiki/NFL_win-loss_records'
webbrowser.open(website)
```

copy the first five rows (edit/copy)

```
nfl_frame = pd.read_clipboard(engine='python', sep='\t+')
```

NOTE: Without the sep argument this technique is hit-or-miss, depending on how the table is copied. Alternatives include copying the table to Excel, saving as .csv, and reading the .csv file instead.

```
nfl_frame
```

	Rank	Team	Won	Lost	Tied	Pct.	First NFL Season	Total Games	Divison
0	1	Chicago Bears	741	555	42	0.57	1920	1338	NFC North
1	2	Dallas Cowboys	480	364	6	0.568	1960	850	NFC East
2	3	Green Bay Packers	720	547	37	0.566	1921	1304	NFC North
3	4	Miami Dolphins	429	335	4	0.561	1966	768	AFC East
4	5	San Francisco 49ers	520	436	14	.553 [a]	1950	1019	NFC West

Note that pandas automatically adds an index in the left-most column. Data as of 1/19/16.

Grab column names:

```
nfl_frame.columns
Index([u'Rank', u'Team', u'Won', u'Lost', u'Tied', u'Pct.',
       u'First NFL Season', u'Total Games', u'Divison'],
      dtype='object')
```

Grab a specific column – 1 word name:

```
nfl_frame.Team
```

Grab a specific column – multiword names:

```
nfl_frame['First Season']
```

Display specific data columns:

```
DataFrame(nfl_frame,columns=['Team','First Season','Total Games'])
```

This returns a *new* DataFrame extracted from nfl_frame

NOTE: if you ask for a column that doesn't exist in the original, you get a column filled with null values (NaN)

Display a specific number of rows:

`nfl_frame.head()` retrieves the first 5 rows

`nfl_frame.head(3)` retrieves the first 3 rows

`nfl_frame.tail()` retrieves the last 5 rows

Grab a record by its index:

`nfl_frame.ix[3]` returns an object with column names & values (the .ix method stands for "index")

Rename index and columns (dict method):

```
dframe.rename(index={0:'a',1:'b',2:'c',3:'d'}, columns={0:'col1',1:'col2},
    inplace=True)
```

I used "inplace=True" instead of "dframe = dframe.rename()"

Rename a specific column:

```
nfl_frame.rename(columns = {'First NFL Season':'First Season'}, inplace=True)
```

<u>Index Objects</u>

Set a Series index to be its own object:

```
coin_index = coins.index
coin_index
Index([u'penny', u'nickel', u'dime', u'quarter'], dtype='object')
```

`coin_index[2]` returns `'dime'`

Note: Indexes are immutable (`coin_index[2]='fred'` is not valid code)

Reindexing

```
ser1 = Series([1,2,3,4],index=['A','B','C','D'])
ser2 = ser1.reindex(['A','B','C','D','E','F'])
```

Creates a new Series, with null values for 'E' and 'F'

NOTE: this <u>also</u> converted the Series from dtype *int64* to *float64*. ser2['C'] returns 3.0

```
ser2.reindex(['A','B','C','D','E','F','G'],fill_value=0)
```

Adds a new index 'G' with a value of 0. Indexes 'E' and 'F' are both still null values.

```
ser2.reindex(['B','A','C','D','E','F','G'])
```

Changes the order of index:value pairs (it doesn't reassign the index) `B:2` is now ahead of `A:1`

```
ser2.reindex(['C','D','E','F'])
```

Removes A:1, B:2 and G:0 from Series ser2.

However: `ser2.reindex(['A','B','C','D','E','F','G'])`

brings back A:1 and B:2 (because ser2 is based on ser1) but *not* G:0. It assigns a null value to G.

Interpolating values between indices:

```
ser3 = Series(['USA','Mexico','Canada'],index=[0,5,10])
```

`ser3.reindex(range(15),method='ffill')` uses a "forward fill" method

ser3 now has 15 members. Index 0-4 = 'USA', 5-9 = 'Mexico' and 10-14='Canada'

Reindexing onto a DataFrame:

```
from numpy.random import randn
dframe = DataFrame(randn(25).reshape((5,5)),index=['A','B','D','E','F'],
   columns=['col1','col2','col3','col4','col5'])
dframe2 = dframe.reindex(['A','B','C','D','E','F'])
```

Inserts a new row 'C' between A and B filled with null values

Reindexing DataFrame columns:

```
dframe2.reindex(columns=['col1','col2','col3','col4','col5','col6']
```

Inserts a new column 'col6' at the end filled with null values (you have to call "`columns`" specifically)

Reindex quickly using .ix:

```
dframe.ix[[rows],[columns]]
```

you could say "newrows=['A',...'F']" and "newcols=['col1',...'col6']" and then dframe.ix[newrows,newcols]

Drop Entry – Rows:

```
ser1 = Series(np.arange(3),index=['a','b','c'])
ser1.drop('b')
```

Displays the Series without row 'b' (although this row still belongs to the series)

Similarly, `dframe1.drop('index1')` drops a row from a DataFrame

Drop Entry – Columns:

`dframe1.drop('col4',axis=1)` axis=0 rows/ axis=1 columns, or `axis=rows` / `axis=columns`

Selecting Entries in a Series:

```
ser1 = Series(np.arange(3),index=['A','B','C'])
ser1 = 2*ser1
```

to avoid confusion in the future

```
ser1
A     0
B     2
C     4
dtype: int32
```

You can grab an entry by index name: `ser1['B']` returns `2`

or by index value: `ser1[1]` returns `2`

or by a range of values: `ser1[0:2]` returns rows `A:0` and `B:2`

or by a *list* of index names: `ser1[['A','B']]` returns rows `A:0` and `B:2`

You can grab entries by logic: `ser1[ser1>3]` returns row `C:4`

You can *change* values using logic: `ser1[ser1>3] = 10 changes C`

Selecting Entries in a DataFrame:

```
dframe = DataFrame(np.arange(25).reshape((5,5)),
   index=['NYC','LA','SF','DC','Chi'],columns=['A','B','C','D','E'])
```

You can grab entries by column name: `dframe['B']` returns all rows with column `B` values

You can grab multiple columns with a *list* of names: `dframe[['B','E']]`

You can grab specific rows using Boolean: `dframe[dframe['C']>8]`

You can grab a specific cell by column and row: `dframe['B']['LA']`

To show a Boolean DataFrame: `dframe>10`

Returns the full DataFrame with `True/False` in each cell as appropriate

You can grab a row using .ix: `dframe.ix['LA']` returns row LA as a Series with column names as its index

NOTE: `dframe.ix[1]` also works to grab the 2nd row. `dframe.ix['LA']['B']` grabs a single cell.

Data Alignment

```
ser1 = Series([0,1,2],index=['A','B','C'])
ser2 = Series([3,4,5,6],index=list('ABCD'))
```
a nice little shortcut

```
ser1                      ser2
A     0                   A     3
B     1                   B     4
C     2                   C     5
dtype: int64              D     6
                          dtype: int64
```

So what happens when we add these together?

```
ser1 + ser2
A      3
B      5
C      7
D    NaN
dtype: float64
```

Because `ser1` didn't have a value for D, it replaced it with a null.

The same behavior occurs with DataFrames (null values are assigned for any unmatched field)

Use .add to assign fill values:

`ser1.add(ser2,fill_value=0)` this adds 0 to whatever hasn't matched

NOTE: `ser2.add(ser1,fill_value=0)` returns the same thing!

When using .add/fill_value with dataframes, null values are assigned when there are no prior values in a cell (at the intersection where new rows from one DataFrame meet new columns from another)

Operations Between a Series and a DataFrame

```
dframe1 = DataFrame(np.arange(9).reshape(3,3),columns=list('ADC'),
   index=['NYC','SF','LA'])
```

`ser1 = dframe1.ix[0]` so `ser1` takes the 'NYC' row and values

`dframe1 - ser1` returns the `dframe1` DataFrame, but now all the 'NYC' values = 0

A DataFrame column is itself a Series, so Series methods apply:
To count the unique values in a DataFrame column:

`dframe['col1'].value_counts()` returns the count from highest to lowest

`dframe['col1'].value_counts(ascending=True)` returns the count from lowest to highest

`dframe['col1'].value_counts(sort=False)` returns the count in index order

`dframe['col1'].value_counts(dropna=False)` includes a count of null values

For more info, incl *normalize* & *bin* parameters:
http://pandas.pydata.org/pandas-docs/version/0.17.1/generated/pandas.Series.value_counts.html

To retrieve rows that contain a particular value:

`dframe[dframe.col1=='value']` **or**

`dframe[dframe['column 1']=='value']`

Summary Statistics on DataFrames

```
arr = np.array([[1,2,np.nan],[np.nan,3,4]])  inserts null values
dframe1 = DataFrame(arr,index=['A','B'],columns = ['One','Two','Three'])
dframe1
```

	One	Two	Three
A	1	2	NaN
B	NaN	3	4

```
dframe1.sum()
One        1
Two        5
Three      4
dtype: float64
```

```
dframe1.sum(axis=1)
A     3
B     7
dtype: float64
```

```
dframe1.min()
One        1
Two        2
Three      4
dtype: float64
```

```
dframe1.idxmin()
One        A
Two        A
Three      B
dtype: object
```

.idxmin returns the index of the lowest value

.max and .idxmax work as expected

```
dframe1.cumsum()
```

redisplays the DataFrame with accumulation sums

	One	Two	Three
A	1	2	NaN
B	NaN	**5**	4

```
dframe1.describe()
```

provides useful summary statistics

	One	Two	Three
count	1	2.000000	1
mean	1	2.500000	4
std	NaN	0.707107	NaN
min	1	2.000000	4
25%	1	2.250000	4
50%	1	2.500000	4
75%	1	2.750000	4
max	1	3.000000	4

For more information about Covariance and Correlation
Check out these great videos! Video credit: Brandon Foltz.

```
from IPython.display import YouTubeVideo
YouTubeVideo('xGbpuFNR1ME')  #Covariance (26:22)
YouTubeVideo('4EXNedimDMs')  #Correlation (27:05)
```

Correlation and Covariance

```
import pandas_datareader.data as pdr
import datetime
```
pandas can get info off the web!

OLD: `import pandas.io.data as pdweb` legacy code still works, but throws a FutureWarning

Get the closing prices:

OLD:
```
prices = pdweb.get_data_yahoo(['CVX','XOM','BP'],
                              start=datetime.datetime(2010, 1, 1),
                              end=datetime.datetime(2013, 1, 1))['Adj Close']
```
NEW:
```
prices = pdr.DataReader(['CVX','XOM','BP'],'yahoo',
                              start=datetime.datetime(2010,1,1),
                              end=datetime.datetime(2013,1,1))['Adj Close']
```
Show preview:

```
prices.head()
```
returns the first 5 rows

	BP	CVX	XOM
Date			
2010-01-04	46.97	66.17	60.26
2010-01-05	47.30	66.63	60.50
2010-01-06	47.55	66.64	61.02
2010-01-07	47.53	66.39	60.83
2010-01-08	47.64	66.51	60.59

Get the volume trades:

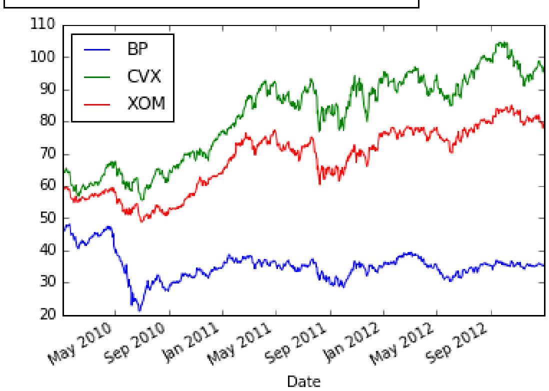
```
volume = pdr.DataReader(['CVX','XOM','BP'],'yahoo',
                              start=datetime.datetime(2010, 1, 1),
                              end=datetime.datetime(2013, 1, 1))['Volume']
volume.head()
```

	BP	CVX	XOM
Date			
2010-01-04	3956100	10173800	27809100
2010-01-05	4109600	10593700	30174700
2010-01-06	6227900	11014600	35044700
2010-01-07	4431300	9626900	27192100
2010-01-08	3786100	5624300	24891800

```
rets = prices.pct_change()
corr = rets.corr
```
calculates the return using the `.pct_change` DataFrame method

gets the correlation of the stocks

```
%matplotlib inline
prices.plot();
```
calls the plot method on the `prices` DataFrame

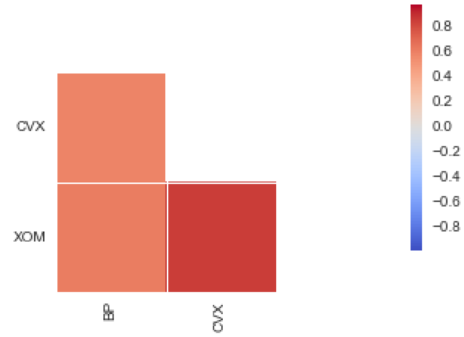


Plot the Correlation using Seaborn:

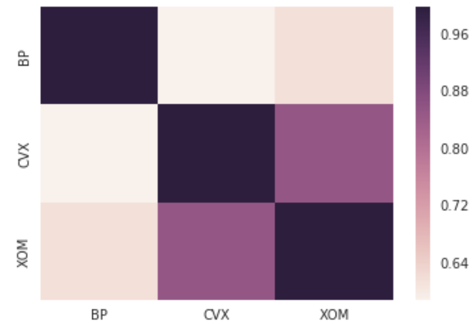
```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

import the seaborn libraries
import pyplot
triggers immediate matplotlib output

OLD: `sns.corrplot(rets,annot=False,diag_names=False)`
Returns a UserWarning: the 'corrplot' function has been deprecated in favor of 'heatmap'

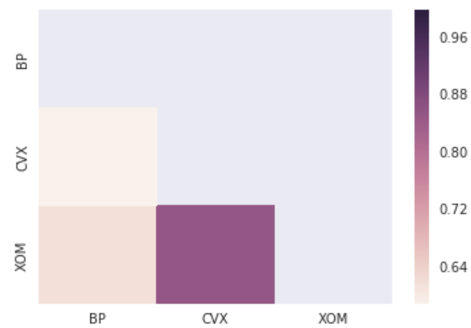


NEW: `sns.heatmap(rets.corr())`
As expected, pretty strong correlations with each other!



Note: to mask half the plot and only show one diagonal:

```
mask = np.zeros_like(rets.corr(), dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
sns.heatmap(rets.corr(),mask=mask)
```



For further info visit
http://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.heatmap.html

For more info:
http://dataskeptic.com/epnotes/ep79_covariance-and-correlation.php

MISSING DATA

Finding, Dropping missing data in a Series:

```
data = Series(['one','two', np.nan, 'four'])
```

```
data                   data.isnull()          data.dropna()
0      one             0 False                0      one
1      two             1 False                1      two
2      NaN             2  True                3     four
3     four             3 False                dtype: object
dtype: object          dtype: bool
```

Finding, Dropping missing data in a DataFrame (Be Careful!):

```
dframe = DataFrame([[1,2,3],[np.nan,5,6],[7,np.nan,9],[np.nan,np.nan,np.nan]])
dframe.dropna(self, axis=0, how='any', thresh=None, subset=None, inplace=False)
```

`dframe.dropna()` will drop entire rows that contain at least one null value!

`dframe.dropna(how='all')` will drop only rows missing *all* data

`dframe.dropna(axis=1)` will drop entire columns that contain at least one null value

`dframe.dropna(thresh=2)` will drop rows that don't have *at least* 2 data points

Note that while `inplace=False`, none of these methods change dframe in place.

Filling missing data points:

`dframe.fillna(1)` fills any missing data point with a 1

`dframe2.fillna({0:5,1:6,2:7,3:8})` will fill 5 in column 0, 6 in column 1, etc.

INDEX HIERARCHY

```
ser = Series(np.random.randn(6),index=[[1,1,1,2,2,2],['a','b','c','a','b','c']])
```

Display the series:

```
ser
1  a   -1.337299
   b   -0.690616
   c    1.792962
2  a    0.457808
   b    0.891199
   c   -1.366387
dtype: float64
```

View the index:

```
ser.index
MultiIndex(levels=[[1, 2], [u'a', u'b', u'c']],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

Select specific subsets:

```
ser[1]
a   -1.337299
b   -0.690616
c    1.792962
dtype: float64
```

Select an internal index level:

```
ser[:,'a']
1   -1.337299
2    0.457808
dtype: float64
```

Now the cool part:

Create a DataFrame from a multilevel Series:

`dframe = ser.unstack()` returns a 2D frame, rows = (1,2), cols = (a,b,c)

Create a multilevel Series from a DataFrame:

```
dframe.unstack()
```

Multilevel Indexing on a DataFrame:

```
dframe2 = DataFrame(np.arange(16).reshape(4,4),
                    index=[['a','a','b','b'],[1,2,1,2]],
                    columns=[['NY','NY','LA','SF'],['cold','hot','hot','cold']])
dframe2
```

		NY		LA	SF
		cold	hot	hot	cold
a	1	0	1	2	3
	2	4	5	6	7
b	1	8	9	10	11
	2	12	13	14	15

Adding names to row & column indices:

```
dframe2.index.names = ['Index1','Index2']
dframe2.columns.names = ['Cities','Temp']

dframe2.swaplevel('Cities','Temp',axis=1)
dframe2.sortlevel(1)
```

swaps column levels (**Temp** is now above **Cities**)

rows become a**1**,b**1**,a**2**,b**2**

Operations on index levels:

```
dframe2.sum(level='Temp',axis=1)
```

	Temp	cold	hot
Index1	**Index2**		
a	**1**	3	3
	2	11	11
b	**1**	19	19
	2	27	27

Renaming columns and indices:

```
dframe.rename(index={0:'A',1:'B'}, inplace=True)
```

READING & WRITING FILES

Setting path names:

Set commonly used directories as raw data strings in the code:

```
path = r'C:\Users\Mike\Documents\Finance\'
file1 = pd.read_csv(path+'file.csv')
```

Comma Separated Value (csv) Files:

`dframe = pd.read_csv('lect25.csv')` previously saved in same directory as lecture notebook

`dframe = pd.read_table('lect25.csv',sep=',')` can also read as table with comma delimiter

`pd.read_csv('lect25.csv',header=None)` assigns an integer column index

`pd.read_csv('lect25.csv',header=None,nrows=2)` takes only the first two rows

`dframe.to_csv('mydataout.csv')` writes a DataFrame to a .csv file

`dframe.to_csv(sys.stdout,sep='_')` lets you see the output directly without saving it

`dframe.to_csv(sys.stdout,columns=[0,1,2])` lets you send a specific set of columns

This is the pandas reader. For info on Python's csv reader/writer go to https://docs.python.org/2/library/csv.html

JSON (JavaScript Object Notation) Files:

```
import json
data = json.loads(json_obj)
```
where "json_obj" is a typical JSON

```
data
{u'clothes': None,
 u'diet': [{u'food': u'grass', u'fur': u'Brown', u'zoo_animal': u'Gazelle'}],
 u'food': [u'Meat', u'Veggies', u'Honey'],
 u'fur': u'Golden',
 u'zoo_animal': u'Lion'}
json.dumps(data)
```
converts back to JSON

```
'{"food": ["Meat", "Veggies", "Honey"], "zoo_animal": "Lion", "fur": "Golden", "diet": [{"food": "grass", "zoo_animal": "Gazelle", "fur": "Brown"}], "clothes": null}'
```

Once you have the JSON, you can choose what info to load into a DataFrame

HTML Files: Note: requires **beautiful_soup**, **html5lib** and **lxml** be installed

```
import pandas as pd
url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'
dframe_list = pd.read_html(url)
```
creates a *list* of DataFrame objects

```
dframe = dframe_list[0]
```
note: in this example, there is no `dframe_list[1]`

```
dframe
```

DataFrame columns: Bank Name, City, ST, CERT, Acquiring Institution, Closing Date, Updated Date, Loss Share Type, Agreement Terminated, Termination Date

NOTE: FOR NFL DATA, THE FOLLOWING CODE WAS NECESSARY:

```
website = 'http://en.wikipedia.org/wiki/NFL_win-loss_records'
nfl_frame_list = pd.read_html(website,match='Rank',header=0)
nfl_frame = nfl_frame_list[0]
```

Note: I was able to pass the argument 'Rank' without 'match=', but this may be library/parsing dependent.

For more info: http://pandas.pydata.org/pandas-docs/stable/io.html#io-read-html

http://pandas.pydata.org/pandas-docs/stable/gotchas.html#html-gotchas

Excel Files: Note: requires **xlrd** and **openpyxl** be installed

Open an excel file as an object:

```
xlsfile = pd.ExcelFile('Lec_28_test.xlsx')
```
file previously saved in notebook directory

Note: this wraps the original file into a special "ExcelFile" class object, which can then be passed to `.read_excel` either sheet by sheet or all at once (performance benefit of reading original file only once)

Parse the first sheet of the excel file and set it as a DataFrame:

OLD: `dframe = xlsfile.parse('Sheet1')`

NEW: `dframe = pd.read_excel(xlsfile, 'Sheet1')` (xlsfile, 0) also works

```
dframe
```

Displays a 3x5 grid, Columns named "This is a test", "Unnamed: 1" and "Unnamed: 2". Rows indexed 0-4.

Note: Unnamed columns are assigned index positions! The tenth column would be "Unnamed: 9"

Passing sheets with the ExcelFile class as a context manager:

```
with pd.ExcelFile('path_to_file.xls') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

For more info: http://pandas.pydata.org/pandas-docs/version/0.17.1/io.html#io-excel-reader

PANDAS CONCATENATE

numpy's concatenate lets you join arrays ("matrices" in the lecture): if arr1 is a 3x4 array,

`np.concatenate([arr1,arr1],axis=1)` creates a horizontal, 3x8 array

`np.concatenate([arr1,arr1],axis=0)` creates a vertical, 6x4 array (default)

in pandas, to concatenate two series:

`pd.concat([ser1,ser2])` creates one long vertical series

If you concatenate two series along axis 1:

`pd.concat([ser1,ser2], axis=1)` the result is a DataFrame! ser1's values fall in column 0, ser2 in column 1

NOTE: if the two series being concatenated share a common index value, then

- the index value will be repeated in a vertical concatenation (axis = 0)
- the index value will appear once, and have values in both columns (axis=1)

You can add a hierarchical index using "keys":

`concat1 = pd.concat([df1, df2, df3], keys= ['x', 'y', 'z'])`

Concatenates DataFrames df1, df2 and df3 one above the other, adds an index hierarchy (x for df1, etc)

`concat1['x']` will retrieve only those records belonging to 'x'

From : http://pandas.pydata.org/pandas-docs/stable/merging.html:

```
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False)
```

`objs`: list or dict of Series, DataFrame, or Panel objects. If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below)

`axis`: {0, 1, ...}, default 0. The axis to concatenate along

`join`: {‘inner’, ‘outer’}, default ‘outer’. How to handle indexes on other axis(es). Outer for union and inner for intersection

`join_axes`: list of Index objects. Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic

`keys`: sequence, default None. Construct hierarchical index using the passed keys as the outermost level If multiple levels passed, should contain tuples.

`levels` : list of sequences, default None. If keys passed, specific levels to use for the resulting MultiIndex. Otherwise they will be inferred from the keys

`names`: list, default None. Names for the levels in the resulting hierarchical index

`verify_integrity`: boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation

`ignore_index` : boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information.

It is worth noting however, that concat (and therefore append) makes a full copy of the data, and that constantly reusing this function can create a signifcant performance hit. If you need to use the operation over several datasets, use a list comprehension.

MERGING DATA

See: http://pandas.pydata.org/pandas-docs/stable/merging.html#database-style-dataframe-joining-merging

Linking rows together by keys

```
dframe1 = DataFrame({'key':['X','Z','Y','Z','X','X'],'data_set_1': np.arange(6)})
dframe2 = DataFrame({'key':['Q','Z','Y','Z'],'data_set_2':[1,5,2,3]})
pd.merge(dframe1,dframe2)
```

	data_set_1	key	data_set_2
0	1	Z	5
1	1	Z	3
2	3	Z	5
3	3	Z	3
4	2	Y	2

Note that `.merge` automatically chooses overlapping columns to merge on (here it's 'key')
Where shared key values appear more than once, `.merge` provides every possible combination

Selecting columns and frames

`pd.merge(dframe1,dframe2,on='key',how='left')` Note: (…how='outer') grabs everything
Returns a frame with all of data_set_1's keys, and null values (NaN) under data_set_2 where it lacked those keys
From the docstring: **how** : {'left', 'right', 'outer', 'inner'}, default 'inner'
* **left**: use only keys from left frame (SQL: left outer join)
* **right**: use only keys from right frame (SQL: right outer join)
* **outer**: use union of keys from both frames (SQL: full outer join)
* **inner**: use intersection of keys from both frames (SQL: inner join)

Merging on multiple keys

```
df_left = DataFrame({'key1': ['SF', 'SF', 'LA'],
                     'key2': ['one', 'two', 'one'],
                     'left_data': [10,20,30]})
df_right = DataFrame({'key1': ['SF', 'SF', 'LA', 'LA'],
                      'key2': ['one', 'one', 'one', 'two'],
                      'right_data': [40,50,60,70]})
pd.merge(df_left, df_right, on=['key1', 'key2'], how='outer')
```

	key1	key2	left_data	right_data
0	SF	one	10	40
1	SF	one	10	50
2	SF	two	20	NaN
3	LA	one	30	60
4	LA	two	NaN	70

Handle duplicate key names with suffixes

If we had merged df_left and df_right on key1 only, there would be two columns named key2.
By default, pandas sets them up as **`key2_x`** for left data, and **`key2_y`** for right data.
We can assign our own suffixes :

```
pd.merge(df_left,df_right,on='key1',suffixes=('_lefty','_righty'))
```

For more info: http://pandas.pydata.org/pandas-docs/dev/generated/pandas.DataFrame.merge.html

Merge on index (not column)

```
df_left = DataFrame({'key': ['X','Y','Z','X','Y'],
                  'data': range(5)})
df_right = DataFrame({'group_data': [10, 20]}, index=['X', 'Y'])
pd.merge(df_left,df_right,left_on='key',right_index=True)
```

	data	key	group_data
0	0	X	10
3	3	X	10
1	1	Y	20
4	4	Y	20

This matched df_right's index values (X,Y) to df_left's "key" data, and retained df_left's index values (0-4).
This works because df_right's index contains unique values (df_left's data would never be duplicated)
From the docstring: "If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on."

Merge on multilevel index
From the docstring: **left_index** : boolean, default False
  Use the index from the left DataFrame as the join key(s).
  If it is a MultiIndex, the number of keys in the other DataFrame
  (either the index or a number of columns) must match the number of levels

Merge key indicator (new in pandas 0.17.0):
`merge(df1, df2, on='col1', how='outer', indicator=True)`
Adds a categorical column `_merge` with values "left_only", "right_only" or "both"

JOIN to join on indexes (row labels)
`df_left.join(df_right)` Column names must be unique. If not:
`df_left.join(df_right, lsuffix='_L')` use `lsuffix` and/or `rsuffix`, not `suffixes`
NOTE: IMO, don't pass an "on=" argument unless coping with hierarchical indices

COMBINING DATAFRAMES
Concatenate, Merge and Join bring two DataFrames together with tools for mapping values from each DataFrame. However, they lack the ability to *choose* between two corresponding values. That is, if dframe1 and dframe2 each have a value for row2, col2, which one wins? Can we choose a value over an empty cell?
The Long Way, using numpy's `where` method:
`Series(np.where(pd.isnull(ser1),ser2,ser1),index=ser1.index)`
Where ser1 is null, take the value from ser2, otherwise use ser1. Apply the index from ser1.
The Shortcut, using pandas' `combine_first` method:
`ser1.combine_first(ser2)`

RESHAPING DATAFRAMES
Stack & Unstack methods

```
dframe1 = DataFrame(np.arange(8).reshape((2, 4)),
                 index=pd.Index(['LA', 'SF'], name='city'),
                 columns=pd.Index(['A', 'B', 'C','D'], name='letter'))
```

`dframe_st = dframe1.stack()` converts to a 3-column Series with col 1,2 as the 2 level index city/letter
`dframe_st = dframe1.unstack()` converts back to a DataFrame
`dframe_st = dframe1.unstack(0)` converts back to a DataFrame but assigns City to columns
`dframe_st = dframe1.unstack('city')` same as above

Note: stack **filters out** NaN by default. To avoid this use `.stack(dropna=False)`

PIVOTING DATAFRAMES

Consider a 12x3 dataframe. Column 1 'date' has 3 values repeating 4 times, column 2 'variable' has 4 values repeating 3 times, and column 3 'value' has random values.

```
dframe_piv = dframe.pivot('date','variable','value')
```

returns a 3x4 dataframe. Here, 'date' becomes a named *index*,
'variable' becomes the *column headings*, and 'value' fills the frame.
If we left the 'value' argument out, it would still fill frame but have 'value' shown as a label above the column headings.

Now consider a 12x7 DataFrame. Feed any three columns to the .pivot method (row, column, filler).
Alternatively, feed only two columns (row, column) and the remaining five will fill the table in turn.
For more info: https://en.wikipedia.org/wiki/Pivot_table
NOTE: the ***.pivot_table*** *method on DataFrames behaves more like groupby. It aggregates values (default=mean).*

DUPLICATES IN DATAFRAMES

```
dframe = DataFrame({'key1': ['A'] * 2 + ['B'] * 3,
                    'key2': [2, 2, 2, 3, 3]})
```

Returns key1/key2 pairs (A:2, A:2, B:2, B:3 and B:3)

```
dframe.duplicated()
```

identifies duplicates. works top-to-bottom, dupes don't need to be adjacent

Returns 0 False, 1 True, 2 False, 3 False, 4 True (dtype: bool)

```
dframe.drop_duplicates()                          drops full-record duplicates
dframe.drop_duplicates(['key1'])                  keeps only the first occurrence of records from 'key1'
dframe.drop_duplicates(['key1'],take_last=True)   keeps the last occurrence
```

MAPPING

Consider a DataFrame with a column called "city" and a Dictionary that matches cities up with states.

```
dframe['state'] = dframe['city'].map(state_map)
```

Creates a new column called 'state' that uses keys from 'city' to grab values from the state_map dictionary.
If a city doesn't exist in the dictionary it assigns a null value.

REPLACE

```
ser1.replace('a','b')               replaces 'a' with 'b' (entire entry?)
ser1.replace([1,2,3],[5,6,7])       replaces 1s, 2s & 3s with 5s, 6s & 7s
```

RENAME INDEX using string operations

```
dframe= DataFrame(np.arange(12).reshape((3, 4)),
                  index=['NY', 'LA', 'SF'],
                  columns=['A', 'B', 'C', 'D'])
dframe.index = dframe.index.map(str.lower)
```

permanently changes the index to lowercase

<u>Change both index & column names while retaining the original:</u>

```
dframe2 = dframe.rename(index=str.title, columns=str.lower)
```

<u>Use dictionaries to change specific names within the index and/or columns:</u> *(note: keys are case sensitive!)*

```
dframe.rename(index={dictionary}, columns={dictionary})
```

add *inplace=True* to change in place

BINNING

Using cut to design a category object

```
years = [1990,1991,1992,2008,2012,2015,1987,1969,2013,2008,1999]
decade_bins = [1960,1970,1980,1990,2000,2010,2020] order matters!!
decade_cat = pd.cut(years,decade_bins)        ...otherwise use sort(decade_bins)
decate_cat
[(1980, 1990], (1990, 2000], (1990, 2000], (2000, 2010], (2010, 2020], ...,
(1980, 1990], (1960, 1970], (2010, 2020], (2000, 2010], (1990, 2000]]
Length: 11
Categories (6, object): [(1960, 1970] < (1970, 1980] < (1980, 1990] < (1990,
2000] < (2000, 2010] < (2010, 2020]]
decade_cat.categories
Index([u'(1960, 1970]', u'(1970, 1980]', u'(1980, 1990]', u'(1990, 2000]',
u'(2000, 2010]', u'(2010, 2020]'], dtype='object')
pd.value_counts(decade_cat)        ranks largest to smallest
(2010, 2020]    3
(1990, 2000]    3
(2000, 2010]    2
(1980, 1990]    2
(1960, 1970]    1
(1970, 1980]    0
dtype: int64
```

In the notation, () means "open" while [] means "closed/inclusive"

NOTE: As it stands, this example bins 1990 with 1985 and requires a sorted "decade_bins" list. To avoid this:

```
decade_cat = pd.cut(years,sorted(decade_bins),right=False)
```

For some reason, if I change 1969 to 1959, the .value_counts method creates a bin of (1958.9, 1987]

Passing data values to the cut:

```
pd.cut(years,2,precision=1)
```

Creates 2 bins, evenly cut between the min/max values of "years". Note that pandas may

```
pd.cut(np.array([.2, 1.4, 2.5, 6.2, 9.7, 2.1]), 3, retbins=True)
([(0.191, 3.367], (0.191, 3.367], (0.191, 3.367], (3.367, 6.533], (6.533, 9.7],
(0.191, 3.367]]
Categories (3, object): [(0.191, 3.367] < (3.367, 6.533] < (6.533, 9.7]],
array([ 0.1905    ,  3.36666667,  6.53333333,  9.7       ])) this comes from retbins=True
```

```
pd.cut(np.array([.2, 1.4, 2.5, 6.2, 9.7, 2.1]), 3, labels=["good","med","bad"])
[good, good, good, med, bad, good]
Categories (3, object): [good < med < bad]
```

Values that lie outside the bins are ignored (the cut array passes a null value)
Floats are converted to integers (by chopping the decimal, not by rounding)
You can't bin in alphabetical order.

OUTLIERS

Consider a 4-column data set with 1000 rows of random numbers:

`np.random.seed(12345)` seed the numpy generator (generates the same set of "random" numbers for each trial)

`dframe = DataFrame(np.random.randn(1000,4))`

Grab a column from the dataset and see which values are greater than 3:

```
col = dframe[0]
col[np.abs(col)>3]
523   -3.428254
900    3.366626
```

in this column, rows 523 and 900 have abs values > 3

`dframe[(np.abs(dframe)>3).any(1)]` would grab rows where any column >3

To cap the data at 3:

`dframe[np.abs(dframe)>3] = np.sign(dframe) *3` multiply the sign by 3 (in place)

PERMUTATIONS

Create an array with a random permutation of 0,1,2,3,4:

`array1 = np.random.permutation(5)`

Note that this produces a permutation *without replacement* (each number appears only once in the array)

```
array1
array([2, 0, 4, 3, 1])
```

(for example)

Shuffle the rows of a DataFrame against this array:

`dframe.take(array1)`

IF ARRAY1 < DFRAME: dframe keeps only those rows represented by array1, and drops the rest.

IF ARRAY1 > DFRAME: throws an IndexError, indices are out of bounds (as opposed to filling a row with null values)

Note: if a row appears more than once in the array, it will appear more than once in the shuffled DataFrame.

Create a permutation with replacement:

`array2 = np.random.randint(0, len(dframe), size=10)`

This will make a 10-member array made up of randomly selected dframe rows.

Rows *can* appear more than once, and not at all.

Note that len(dframe) counts the number of rows, not the number of cells.

GROUPBY ON DATAFRAMES

```
dframe = DataFrame({'k1':['X','X','Y','Y','Z'],
                    'k2':['alpha','beta','alpha','beta','alpha'],
                    'dataset1':np.random.randn(5),
                    'dataset2':np.random.randn(5)})
```

	dataset1	dataset2	k1	k2
0	-0.45067	-1.63403	X	alpha
1	0.268817	0.458236	X	beta
2	0.023818	0.212936	Y	alpha
3	-1.2282	-1.36003	Y	beta
4	0.032472	-1.54512	Z	alpha

Create a SeriesGroupBy object:

`group1 = dframe.groupby('k1')` divides the DataFrame into groups around values in column 'k1'

```
group1
&lt;pandas.core.groupby.SeriesGroupBy object at 0x000000000A6C9898&gt;
```

Note that the GroupBy object is just stored data, not a DataFrame

Operations on a group return a DataFrame:

`dframe.groupby('k1').mean()` returns a DataFrame with index = k1, and mean values for dataset1 and dataset2

NOTE: Since 'k2' did not contain numerical values, it was dropped from the groupby.mean DataFrame

Groupby.mean ignores null values. (the mean of x and null is x)

When we get to statistical analysis, is this a good way to obtain sample means to test for normal distribution?

Group data in one column 'dataset1' by another column 'k1'

```
group1 = dframe['dataset1'].groupby(dframe['k1'])
```

When specifying only one column from dframe, you also have to call "*dframe*['k1']"

Group by multiple column keys:

`dframe.groupby(['k1','k2']).mean()` returns a DataFrame with hierarchical index

Group one column by multiple column keys:

```
dataset2_group = dframe.groupby(['k1','k2'])[['dataset2']]
```

Note: you're calling the 'dataset2' column from the larger data set, which returns a DataFrame, because (dframe['dataset2'].groupby(dframe['k1','k2']) is not valid code.

Assign keys to 'dataset1' and group by them instead:

```
cities = np.array(['NY','LA','LA','NY','NY'])
month = np.array(['JAN','FEB','JAN','FEB','JAN'])
dframe['dataset1'].groupby([cities,month]).mean()
LA   FEB    0.268817
     JAN    0.023818
NY   FEB   -1.228203
     JAN   -0.209097
Name: dataset1, dtype: float64
```

this is the only calculated mean (JAN −0.209097)

Note that the output sorts by city then month *alphabetically*

Because we only grouped one column, groupby.mean returned a Series

Unlike the example above, this passed array indices to groupby, not column names

Other GroupBy methods:

```
dframe.groupby('k1').size()
```
returns the number of occurrences of each not-null member of k1

```
dframe.groupby('k1').count()
```
returns a DataFrame, index=k1, other columns report a count of not-null members that match up to k1 values

```
dframe.groupby('k1').sum()
```
returns a DataFrame, index=k1, other columns report a sum of not-null members that match up to k1 values

```
dframe[['col1','col2']].groupby(dframe['k1']).sum()
```
as above, but for specified column(s)

```
dframe.groupby('k1').max()
dframe.groupby('k1').min()
```

Iterate over groups:

```
for name,group in dframe.groupby('k1'):
    print "This is the %s group" %name
    print group
    print '\n'
This is the X group
   dataset1  dataset2 k1     k2
0 -0.123544  1.924614  X  alpha
1 -1.448666  0.477115  X   beta

This is the Y group
   dataset1  dataset2 k1     k2
2 -1.139759 -1.378362  Y  alpha
3 -0.617664 -0.105714  Y   beta

This is the Z group
   dataset1  dataset2 k1     k2
4 -0.573748  0.409242  Z  alpha
```

This operation supports multiple keys:

```
for (k1,k2),group in dframe.groupby(['k1','k2']):
    print "Key1 = %s Key2 = %s" %(k1,k2)
    print group
    print '\n'
```
(sorts alphabetically by k1 then k2)

Create a dictionary from grouped data pieces:

```
group_dict = dict(list(dframe.groupby('k1')))
```

Here each unique member of k1 becomes a key, and its group DataFrame becomes a value!

For more info: http://pandas.pydata.org/pandas-docs/stable/groupby.html

Apply GroupBy using Dictionaries and Series

First, make a dataframe:

```
animals = DataFrame(np.arange(16).reshape(4, 4),
                    columns=['W', 'X', 'Y', 'Z'],
                    index=['Dog', 'Cat', 'Bird', 'Mouse'])
```

Add some null values:

```
animals.ix[1:2, ['W', 'Y']] = np.nan
animals
```

	W	X	Y	Z
Dog	0	1	2	3
Cat	NaN	5	NaN	7
Bird	8	9	10	11
Mouse	12	13	14	15

Create a dictionary with "behavior" values:

```
behavior_map = {'W': 'good', 'X': 'bad', 'Y': 'good','Z': 'bad'}
```

Group the DataFrame using the dictionary:

```
animal_col = animals.groupby(behavior_map, axis=1)
```

Now you can perform operations on `animals` based on the behavior_map dictionary values!
The same thing can happen using Series.

Aggregation

The `.agg(func)` method lets you pass an aggregate function (like mean, max_minus_min, etc) to a GroupBy object.
You can also pass string methods: `grouped_frame.agg('mean')`
Note: the `.pivot_table` method on DataFrames takes an "`aggfunc=`" argument (default is np.mean)

Refer to the Python Sample Code file for an example using UC Irvine's wine quality dataset on GroupBy aggregates.

Cross Tabulation

This is a special case of the `.pivot_table` method on DataFrames

	Sample	Animal	Intelligence
0	1	Dog	Smart
1	2	Dog	Smart
2	3	Cat	Dumb
3	4	Cat	Dumb
4	5	Dog	Dumb
5	6	Cat	Smart

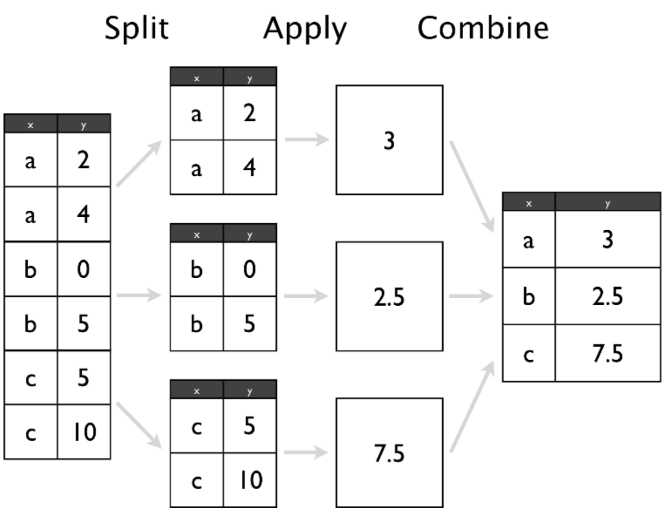
`pd.crosstab(dframe.Animal,dframe.Intelligence,margins=True)` margins=True adds the "All" info

Intelligence	Dumb	Smart	All
Animal			
Cat	2	1	3
Dog	1	2	3
All	3	3	6

Provides a frequency table by default, although crosstab does support aggfunc arguments as well

Split, Apply, Combine

A visual explanation: source = https://github.com/ramnathv/rblocks/issues/8



Split here is accomplished by the groupby command. If the function you're applying requires that members of the group be sorted, sort the dataframe first.

Apply can be a predefined function to be performed on each group in turn.

Combine is whatever gets returned once the apply finishes.

Using the same UC Irvine wine quality dataset as above (Aggregation – refer to the Python Sample Code file):

Build a DataFrame from the downloaded file

```
dframe_wine = pd.read_csv('winequality_red.csv',sep=';')
```

Create a function that assigns a rank to each wine based on alcohol content, with 1 being the highest alcohol content

```
def ranker(df):
    df['alc_content_rank'] = np.arange(len(df)) + 1    index items 0-4 are ranked 1-5
    return df
```

Sort the DataFrame by alcohol in descending order (highest at the top)

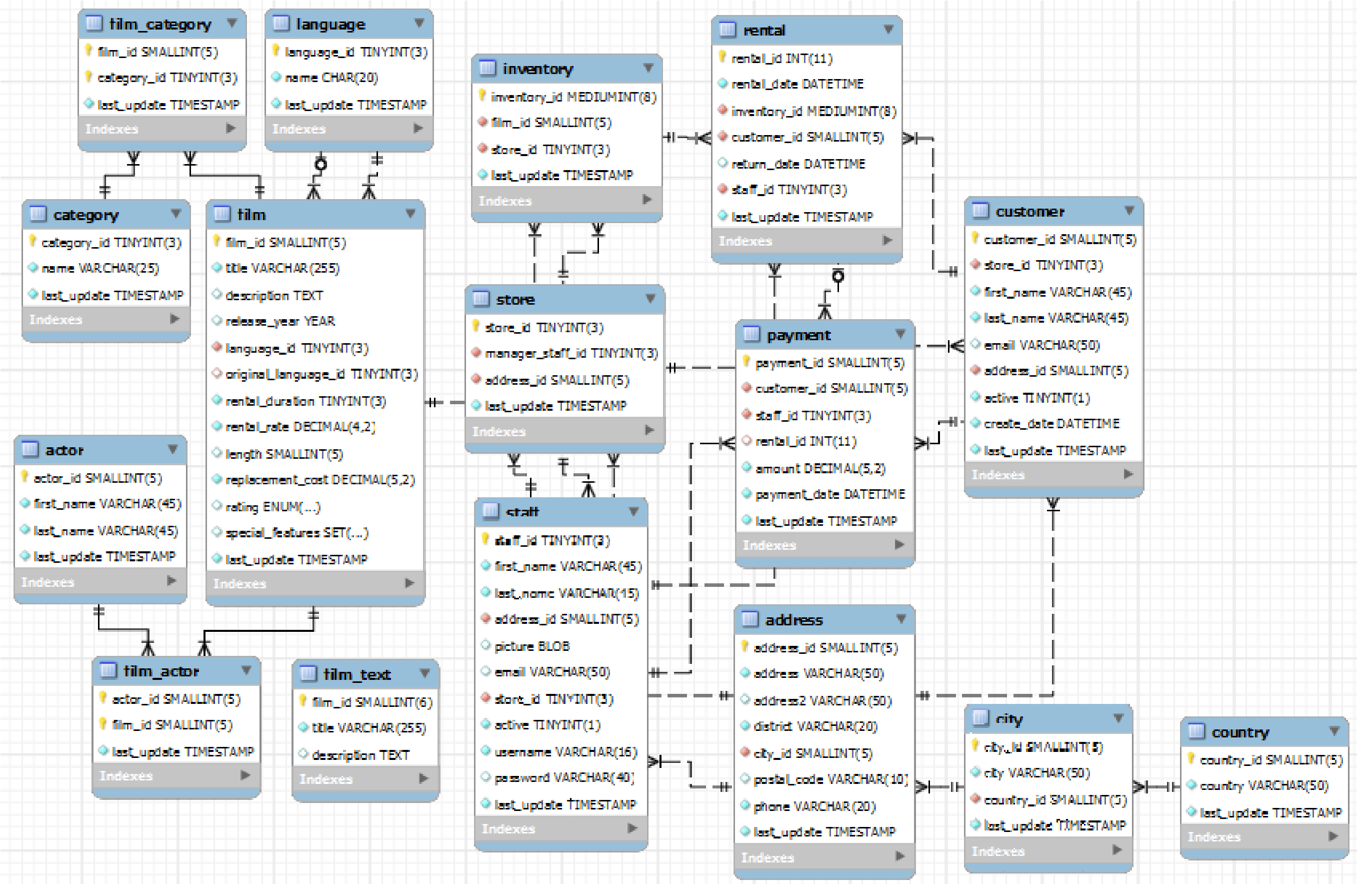
```
dframe_wine.sort_values(by='alcohol', ascending=False, inplace=True)
```

Group by quality and apply the ranking function

```
dframe_wine = dframe_wine.groupby('quality').apply(ranker)

dframe_wine[dframe_wine.alc_content_rank == 1].sort_values(by='quality')
```

SQL with Python
For this lecture we'll focus on using pandas, SQLAlchemy and the the SQLite sql browser for performing SQL queries. (Many other options exist). Also, I downloaded the sakila DVD rental database from here.



First, connect to the SQL database (using Python's built-in SQLite3 module):

```
import sqlite3
import pandas as pd
con = sqlite3.connect("sakila.db")
sql_query = ''' SELECT * FROM customer '''
```

Use pandas to pass the sql query using connection from SQLite3

```
df = pd.read_sql(sql_query, con)
df.head()
```

	customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
0	1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2/14/2006 10:04:36 PM	9/14/2011 6:10:28 PM
1	2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	6	1	2/14/2006 10:04:36 PM	9/14/2011 6:10:28 PM
2	3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7	1	2/14/2006 10:04:36 PM	9/14/2011 6:10:28 PM
3	4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2/14/2006 10:04:36 PM	9/14/2011 6:10:28 PM
4	5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	9	1	2/14/2006 10:04:36 PM	9/14/2011 6:10:28 PM

SQL Statements: Select, Distinct, Where, And & Or

In the statements above, we used SELECT and loaded the entire customer table

To save overhead, we can define a function for passing specific queries:

```
def sql_to_df(sql_query):
    df = pd.read_sql(sql_query, con)
    return df

query = ''' SELECT first_name, last_name
            FROM customer; '''
sql_to_df(query).head()
```

rely on linebreaks & indents for improved readability

also, SELECT, FROM etc. are not case-sensitive

Returns two specific columns

```
query = ''' SELECT DISTINCT country_id
            FROM city'''
```

Returns distinct values from a specific column (not the entire record)

```
query = ''' SELECT *
            FROM customer
            WHERE store_id = 1'''
```

Returns records that fit a specific criteria. Supports Boolean operators (=, <> or !=, <, >, etc.)

```
            WHERE first_name = 'MARY'  '''
```

Text values should be enclosed in single quotes (numerical values are not)

```
query = ''' SELECT *
            FROM film
            WHERE release_year = 2006
            AND rating = 'R' '''
```

Supports conditional statements AND and OR

Aggregate functions include:

- AVG() - Returns the average value.
- COUNT() - Returns the number of rows.
- FIRST() - Returns the first value.
- LAST() - Returns the last value.
- MAX() - Returns the largest value.
- MIN() - Returns the smallest value.
- SUM() - Returns the sum.

The usual syntax is:

```
    SELECT AGG_FUNC(column_name)
    FROM table_name
    WHERE column_name
```

```
query = ''' SELECT COUNT(customer_id)
            FROM customer'''
```

Returns one item, with a count of the number of customers (index = 0)

Note that parentheses are required (they're optional after DISTINCT)

Wildcards are used with the LIKE operator

```
query = ''' SELECT *
            FROM customer
            WHERE first_name LIKE 'M%' ; '''
            WHERE last_name LIKE '_ING' ; '''
```

use % to denote trailing text

use _ to denote leading text

Character Lists are enclosed in brackets

NOTE: Using [charlist] with SQLite is a little different than with other SQL formats, such as MySQL.

In MySQL you would use:

```
    WHERE value LIKE '[charlist]%'
```

In SQLite you use:

```
    WHERE value GLOB '[charlist]*'
```

```
query = ''' SELECT *
            FROM customer
            WHERE first_name GLOB '[AB]*' ; '''
```

Returns records with any combination of A and/or B in the first name

Sorting with ORDER BY

```
query = ''' SELECT *
            FROM customer
            ORDER BY last_name ; '''            sort ascending
            ORDER BY last_name DESC; '''        sort descending
```

Grouping with GROUP BY

The GROUP BY statement is used with the aggregate functions to group the results by one or more columns:

```
    SELECT column_name, aggregate_function(column_name)
    FROM table_name
    WHERE column_name operator value
    GROUP BY column_name;
```

```
query = ''' SELECT store_id , COUNT(customer_id)
            FROM customer
            GROUP BY store_id; '''
```

For more info: https://pymotw.com/2/sqlite3/

Web Scraping with Python

Practical considerations:

1.) You should check a site's terms and conditions before you scrape them.
2.) Space out your requests so you don't overload the site's server, doing this could get you blocked.
3.) Scrapers break after time - web pages change their layout all the time, you'll more than likely have to rewrite your code.
4.) Web pages are usually inconsistent, more than likely you'll have to clean up the data after scraping it.
5.) Every web page and situation is different, you'll have to spend time configuring your scraper.

Standard Imports (BeautifulSoup4, lxml, requests)

```
from bs4 import BeautifulSoup
import requests
import pandas as pd
from pandas import Series,DataFrame
```

Set the URL (in this case, legislative reports from the University of California Web Page)

url = 'http://www.ucop.edu/operating-budget/budgets-and-reports/legislative-reports/2013-14-legislative-session.html'

Request content from webpage

```
result = requests.get(url)
c = result.content
```

Set as Beautiful Soup Object

```
soup = BeautifulSoup(c)
```

Use BeautifulSoup to search for the table we want to grab:

Use "inspect" in your web browser to find the particular keywords that correspond to the section you want

```
summary = soup.find("div",{'class':'list-land','id':'content'})
tables = summary.find_all('table')
```

Now we need to use Beautiful Soup to find the table entries.

A 'td' tag defines a standard cell in an HTML table. The 'tr' tag defines a row in an HTML table.
We'll parse through our tables object and try to find each cell using the findALL('td') method.
There are tons of options to use with findALL in beautiful soup. You can read about them here.

```
data = []                              Set up empty data list
rows = tables[0].findAll('tr')         Set rows as first indexed object in tables with a row
for tr in rows:                        grab every HTML cell in every row
    cols = tr.findAll('td')
    for td in cols:                    Check to see if text is in the row
        text = td.find(text=True)
        print text,
        data.append(text)
```

Refer to the jupyter notebook to see output

Now we'll use a for loop to go through the list and grab only the cells with a pdf file in them.
We also need to keep track of the index to set up the date of the report.

```
reports = []                              Set up empty lists
date = []
index = 0
for item in data:                         Go find the pdf cells
    if 'pdf' in item:
        date.append(data[index-1])        Add the date and reports
        reports.append(item.replace(u'\xa0', u' '))      Get rid of \xa0
    index += 1
```

You'll notice a line to take care of '\xa0 ' This is due to a unicode error that occurs if you don't do this. Web pages can be messy and inconsistent and it is very likely you'll have to do some research to take care of problems like these.
Here's the link I used to solve this particular issue: StackOverflow Page

Now all that is left is to organize our data into a pandas DataFrame!

```
date = Series(date)                                   Set up Dates and Reports as Series
reports = Series(reports)
legislative_df = pd.concat([date,reports],axis=1)     Concatenate into a DataFrame
legislative_df.columns = ['Date','Reports']           Set up the columns
legislative_df.head()                                 Show the finished DataFrame (20 reports)
```

	Date	Reports
0	8/1/2013	2013-14 (EDU 92495) Proposed Capital Outlay Pr...
1	9/1/2013	2014-15 (EDU 92495) Proposed Capital Outlay P...
2	11/1/2013	Utilization of Classroom and Teaching Laborato...
3	11/1/2013	Instruction and Research Space Summary & Analy...
4	11/15/2013	Statewide Energy Partnership Program (pdf)

For more info on HTML:
W3School
Codecademy

Available webscraping tools:
https://import.io/
https://www.kimonolabs.com/