Group by

We've seen that even though PANDAS allows us to iterate over every row in a data frame this is generally a slow way to accomplish a given task and it's not very pandorable.
For instance, if we wanted to write some code to iterate over all the states and generate a list of the average census population numbers.
We could do so using a loop in the unique function.
Another option is to use the dataframe *group by* function.
This function takes some column name or names and splits the dataframe up into chunks based on those names; it returns a dataframe group by object.
Which can be iterated upon, and then returns a tuple where the first item is the group condition, and the second item is the data frame reduced by that grouping.
Since it's made up of two values, you can unpack this, and project just the column that you're interested in, to calculate the average.

Here are some examples of both methods in action.
Let's first load the census data, and then exclude the state level summarizations, which had a sum level value of 50.

```
import pandas as pd
import numpy as np
df = pd.read_csv('census.csv')
df = df[df['SUMLEV']==50]
df
```

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOMESTICMIG20
1	50	3	6	1	1	Alabama	Autauga County	54571	54571	54660	...	7.2420
2	50	3	6	1	3	Alabama	Baldwin County	182265	182265	183193	...	14.8329
3	50	3	6	1	5	Alabama	Barbour County	27457	27457	27341	...	-4.7281
4	50	3	6	1	7	Alabama	Bibb County	22915	22919	22861	...	-5.5270
5	50	3	6	1	9	Alabama	Blount County	57322	57322	57373	...	1.8073
...	...	...	...	...	...	...	...	...	...	...	...	
3188	50	4	8	56	37	Wyoming	Sweetwater County	43806	43806	43593	...	1.0726
3189	50	4	8	56	39	Wyoming	Teton County	21294	21294	21297	...	-1.5895
3190	50	4	8	56	41	Wyoming	Uinta County	21118	21118	21102	...	-17.7559

In the first we used the census data.

We get a list of the unique states. Then for each state we reduce the dataframe and calculate the average.

```
%%timeit -n 10
for state in df['STNAME'].unique():
    avg = np.average(df.where(df['STNAME']==state).dropna()['CENSUS2010POP'])
    print('Counties in state ' + state + ' have an average population of ' + str(avg))
```

```
Counties in state North Carolina have an average population of 95354.83
Counties in state North Dakota have an average population of 12690.396226415094
Counties in state Ohio have an average population of 131096.63636363635
Counties in state Oklahoma have an average population of 48718.844155844155
Counties in state Oregon have an average population of 106418.72222222222
Counties in state Pennsylvania have an average population of 189587.74626865672

Counties in state Rhode Island have an average population of 210513.4
Counties in state South Carolina have an average population of 100551.39130434782
Counties in state South Dakota have an average population of 12336.060606060606
Counties in state Tennessee have an average population of 66801.1052631579
Counties in state Texas have an average population of 98998.27165354331
Counties in state Utah have an average population of 95306.37931034483
Counties in state Vermont have an average population of 44695.78571428572
Counties in state Virginia have an average population of 60111.29323308271
Counties in state Washington have an average population of 172424.10256410256
Counties in state West Virginia have an average population of 33690.8
Counties in state Wisconsin have an average population of 78985.91666666667
Counties in state Wyoming have an average population of 24505.478260869564
9.85 s ± 118 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

If we time this we see it takes a while.
I've set the number of loops here the time it should take to ten because I'm live loading.

Here's the same approach with a group by object.
We tell pandas we're interested in group and with a state name and then we calculate the average using just one column and all of the data in that column.
When we time it we see a huge difference in the speed

```
%%timeit -n 10
for group, frame in df.groupby('STNAME'):
    avg = np.average(frame['CENSUS2010POP'])
    print('Counties in state ' + group + ' have an average population of ' + str(avg))
```

```
Counties in state North Carolina have an average population of 95354.83
Counties in state North Dakota have an average population of 12690.396226415094
Counties in state Ohio have an average population of 131096.63636363635
Counties in state Oklahoma have an average population of 48718.844155844155
Counties in state Oregon have an average population of 106418.72222222222
Counties in state Pennsylvania have an average population of 189587.74626865672
Counties in state Rhode Island have an average population of 210513.4
Counties in state South Carolina have an average population of 100551.39130434782
Counties in state South Dakota have an average population of 12336.060606060606
Counties in state Tennessee have an average population of 66801.1052631579
Counties in state Texas have an average population of 98998.27165354331
Counties in state Utah have an average population of 95306.37931034483
Counties in state Vermont have an average population of 44695.78571428572
Counties in state Virginia have an average population of 60111.29323308271
Counties in state Washington have an average population of 172424.10256410256
Counties in state West Virginia have an average population of 33690.8
Counties in state Wisconsin have an average population of 78985.91666666667
Counties in state Wyoming have an average population of 24505.478260869564
103 ms ± 14.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Now, 99% of the time, you'll use group by on one or more columns.
But you can actually provide a function to group by as well and use that to segment your data.
This is a bit of a fabricated example but let's say that you have a big batch job with lots of processing and you want to work on only a third or so of the states at a given time.
We could create some function which returns a number between zero and two based on the first character of the state name.
Then we can tell group by to use this function to split up our dataframe.
It's important to note that in order to do this you need to set the index of the dataframe to be the column that you want to group by first.

Here's an example.
We'll create some new function called fun and if the first letter of the parameter is a capital M we'll return a 0.
If it's a capital Q we'll return a 1 and otherwise we'll return a 2.
Then we'll pass this function to the dataframe, and see that the dataframe is segmented by the calculated group number.

```
df.head()
```

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOMESTICMIG2011
1	50	3	6	1	1	Alabama	Autauga County	54571	54571	54660	...	7.242091
2	50	3	6	1	3	Alabama	Baldwin County	182265	182265	183193	...	14.832960
3	50	3	6	1	5	Alabama	Barbour County	27457	27457	27341	...	-4.728132
4	50	3	6	1	7	Alabama	Bibb County	22915	22919	22861	...	-5.527043
5	50	3	6	1	9	Alabama	Blount County	57322	57322	57373	...	1.807375

5 rows × 100 columns

```
df = df.set_index('STNAME')

def fun(item):
    if item[0]<'M':
        return 0
    if item[0]<'Q':
        return 1
    return 2

for group, frame in df.groupby(fun):
    print('There are ' + str(len(frame)) + ' records in group ' + str(group) + ' for processing.')
```

```
There are 1177 records in group 0 for processing.
There are 1134 records in group 1 for processing.
There are 831 records in group 2 for processing.
```

This kind of technique, which is sort of a light weight hashing, is commonly used to distribute tasks across multiple workers.
Whether they are cores in a processor, nodes in a supercomputer, or disks in a database.

A common work flow with group by as that you split your data, you apply some function, then you combine the results.
This is called split apply combine pattern.
And we've seen the splitting method, but what about apply?
Certainly iterative methods as we've seen can do this, but the groupby object also has a method called *agg* which is short for aggregate.
This method applies a function to the column or columns of data in the group, and returns the results.

With *agg*, you simply pass in a dictionary of the column names that you're interested in, and the function that you want to apply.
For instance to build a summary dataframe for the average populations per state, we could just give agg a dictionary with the Census 2010pop key and the numpy average function.

```
df = pd.read_csv('census.csv')
df = df[df['SUMLEV']==50]
```

```
df.groupby('STNAME').agg({'CENSUS2010POP': np.average})
```

	CENSUS2010POP
STNAME	
Alabama	71339.343284
Alaska	24490.724138
Arizona	426134.466667
Arkansas	38878.906667
California	642309.586207
Colorado	78581.187500
Connecticut	446762.125000
Delaware	299311.333333
District of Columbia	601723.000000
Florida	280616.567164
Georgia	60928.635220
Hawaii	272060.200000

Now, I want to flag a potential issue and using the aggregate method of group by objects.
You see, when you pass in a dictionary it can be used to either to identify the columns to apply a function on or to name an output column if there's multiple functions to be run.

The difference depends on the keys that you pass in from the dictionary and how they're named.

In short, while much of the documentation and examples will talk about a single groupby object, there are really two different objects.
<u>The dataframe groupby and the series groupby.</u>
And these objects behave a little bit differently with aggregate.

For instance, we take our census data and convert it into a series with the state names as the index and only columns as the census 2010 population. And then we can group this by index using the *level* parameter.
Then we call the agg method where the dictionary that has both the numpy average and the numpy sum functions.
PANDAS applies those functions to the series object and, since there's only one column of data It applies both functions to that column and prints out the output.

We can do the same thing with a dataframe instead of a series.
We set the index to be the state name, we group by the index, and we project two columns.
The population estimate in 2010, the population estimate in 2011.
When we call aggregate with two parameters, it builds a nice hierarchical column space and all of our functions are applied.

```
(df.set_index('STNAME').groupby(level=0)['POPESTIMATE2010','POPESTIMATE2011']
    .agg({'avg': np.average, 'sum': np.sum}))
```

	avg		sum	
	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE2010	POPESTIMATE2011
STNAME				
Alabama	71420.313433	71658.328358	4785161	4801108
Alaska	24621.413793	24921.379310	714021	722720
Arizona	427213.866667	431248.800000	6408208	6468732
Arkansas	38965.253333	39180.506667	2922394	2938538
California	643691.017241	650000.586207	37334079	37700034
Colorado	78878.968750	79991.875000	5048254	5119480
Connecticut	447464.625000	448719.875000	3579717	3589759
Delaware	299930.333333	302638.666667	899791	907916
District of Columbia	605126.000000	620472.000000	605126	620472
Florida	281341.641791	285157.208955	18849890	19105533
Georgia	61090.905660	61712.452830	9713454	9812280
Hawaii	272796.000000	275645.400000	1363980	1378227
Idaho	35704.227273	36003.045455	1570986	1584134
Illinois	125894.598039	126096.882353	12841249	12861882
Indiana	70549.891304	70835.271739	6490590	6516845
Iowa	30815.090909	30963.525253	3050694	3065389

Where the confusion can come in is when we change the labels of the dictionary we passed to aggregate, to correspond to the labels in our group dataframe.
In this case, pandas recognizes that they're the same and maps the functions directly to columns instead of creating a hierarchically labeled column.
From my perspective this is very odd behavior, not what I would expect given the labeling change.
So just be aware of this when using the aggregate function.

```
(df.set_index('STNAME').groupby(level=0)['POPESTIMATE2010','POPESTIMATE2011']
    .agg({'POPESTIMATE2010': np.average, 'POPESTIMATE2011': np.sum}))
```

	POPESTIMATE2010	POPESTIMATE2011
STNAME		
Alabama	71420.313433	4801108
Alaska	24621.413793	722720
Arizona	427213.866667	6468732
Arkansas	38965.253333	2938538
California	643691.017241	37700034
Colorado	78878.968750	5119480
Connecticut	447464.625000	3589759
Delaware	299930.333333	907916
District of Columbia	605126.000000	620472
Florida	281341.641791	19105533
Georgia	61090.905660	9812280
Hawaii	272796.000000	1378227
Idaho	35704.227273	1584134

So that's the group by function.
I use the group by function regularly in my work, and it's very handy for segmenting a dataframe, working on small pieces of the data frame, and then creating bigger data frames later.