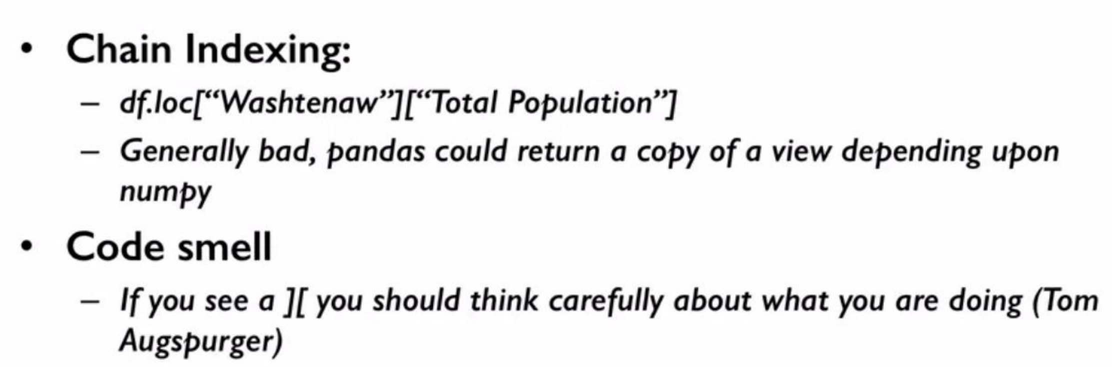
Pandas Idioms

Python programmers will often suggest that there many ways the language can be used to solve a particular problem.
But that some are more appropriate than others.
The best solutions are celebrated as Idiomatic Python and there are lots of great examples of this on stack overflow and websites.

An idiomatic solution is often one which has both high performance and high readability.
This isn't necessarily true.
A sort of sub-language within Python, Pandas has its own set of idioms.
We've alluded to some of these already, such as using vectorization whenever possible, and not using iterative loops if you don't need to.
Several developers and users within the Panda's community have used the term pandorable for these idioms.
I think it's a great term.
So, I wanted to share with you a couple of key features of how you can make your code pandorable.

The first of these is called method chaining.
Now we saw that previously, you could chain pandas calls together when you're querying DataFrames.
For, instance if you wanted to select rows based on index like county name.
Then you wanted to only project certain columns like the total population, you can write a query, like df.loc[“Washtenaw”][“Total Population”]
This is a form of chaining, called chain indexing.
And it's generally a bad practice.
Because it's possible that pandas could be returning a copy or a view of the DataFrame depending upon the underlying NumPy library.
In his descriptions of idiomatic Pandas patterns developer Tom Augspurger described a rule of thumb for this.

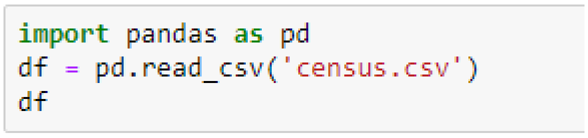


- **Chain Indexing:**
  - ***df.loc[“Washtenaw”][“Total Population”]***
  - ***Generally bad, pandas could return a copy of a view depending upon numpy***
- **Code smell**
  - ***If you see a][you should think carefully about what you are doing (Tom Augspurger)***

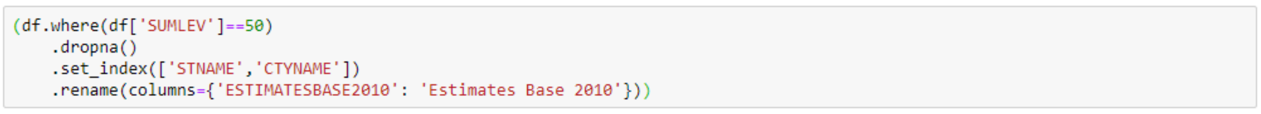
If you see back to back square brackets, then you should think carefully if you want to be doing chain indexing.
I think this is great as a sort of code smell or anti pattern.

Method chaining though, little bit different.
The general idea behind method chaining is that every method on an object returns a reference to that object.

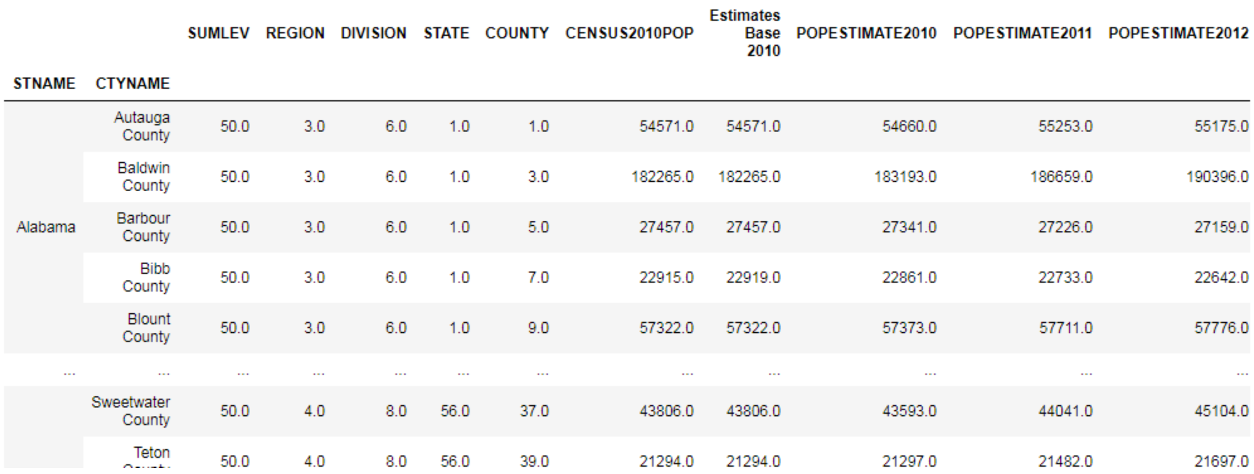
The beauty of this is that you can condense many different operations on a DataFrame, for instance, into one line or at least one statement of code. Here's an example of two pieces of code in pandas using our census data. The first is the pandorable way to write the code with method chaining.



```
import pandas as pd
df = pd.read_csv('census.csv')
df
```



```
(df.where(df['SUMLEV']==50)
    .dropna()
    .set_index(['STNAME','CTYNAME'])
    .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))
```

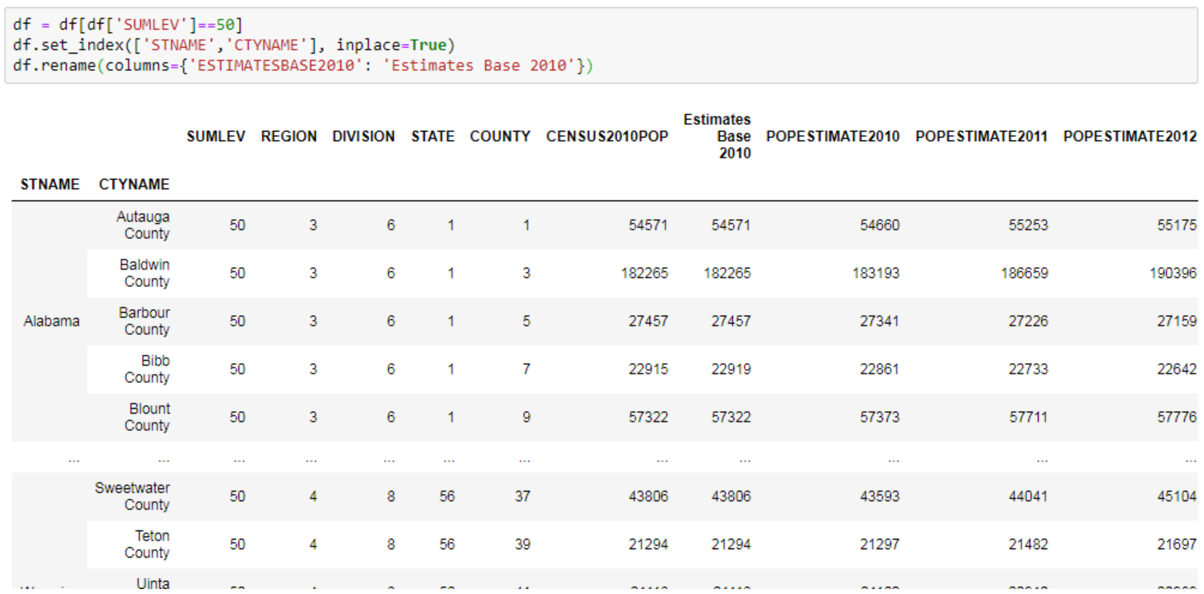


		SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Estimates Base 2010	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE2012
STNAME	CTYNAME										
Alabama	Autauga County	50.0	3.0	6.0	1.0	1.0	54571.0	54571.0	54660.0	55253.0	55175.0
	Baldwin County	50.0	3.0	6.0	1.0	3.0	182265.0	182265.0	183193.0	186659.0	190396.0
	Barbour County	50.0	3.0	6.0	1.0	5.0	27457.0	27457.0	27341.0	27226.0	27159.0
	Bibb County	50.0	3.0	6.0	1.0	7.0	22915.0	22919.0	22861.0	22733.0	22642.0
	Blount County	50.0	3.0	6.0	1.0	9.0	57322.0	57322.0	57373.0	57711.0	57776.0
...	...	...	...	...	...	...	...	...	...	...	...
	Sweetwater County	50.0	4.0	8.0	56.0	37.0	43806.0	43806.0	43593.0	44041.0	45104.0
	Teton	50.0	4.0	8.0	56.0	39.0	21294.0	21294.0	21297.0	21482.0	21697.0

In this code, there's no *inplace* flag being used and you can see that when we first run a where query, then a dropna, then a set_index, and then a rename. You might wonder why the whole statement is enclosed in parentheses and that's just to make the statement more readable.

In Python, if you begin with an open parentheses, you can span a statement over multiple lines and things read a little bit nicer.

The second example is a more traditional way of writing code. There's nothing wrong with this code in the functional sense, you might even be able to understand it better as a new person to the language.



```
df = df[df['SUMLEV']==50]
df.set_index(['STNAME','CTYNAME'], inplace=True)
df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})
```

		SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Estimates Base 2010	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE2012
STNAME	CTYNAME										
Alabama	Autauga County	50	3	6	1	1	54571	54571	54660	55253	55175
	Baldwin County	50	3	6	1	3	182265	182265	183193	186659	190396
	Barbour County	50	3	6	1	5	27457	27457	27341	27226	27159
	Bibb County	50	3	6	1	7	22915	22919	22861	22733	22642
	Blount County	50	3	6	1	9	57322	57322	57373	57711	57776
...	...	...	...	...	...	...	...	...	...	...	...
	Sweetwater County	50	4	8	56	37	43806	43806	43593	44041	45104
	Teton County	50	4	8	56	39	21294	21294	21297	21482	21697
[illegible]	Uinta	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

It's just not as pandorable as the first example.
Now, the key with any good idiom is to understand when it isn't helping you.
In this case, you can actually time both methods and see that the latter method is faster.
So, this is a particular example of a classic time readability trade off.

You'll see lots of examples on stock overflow and in documentation of people using method chaining in their pandas.
And so, I think being able to read and understand the syntax is really worth your time.

Here's another pandas idiom.
Python has a wonderful function called map, which is sort of a basis for functional programming in the language.
When you want to use map in Python, you pass it some function you want called, and some iterable, like a list, that you want the function to be applied to.
The results are that the function is called against each item in the list, and there's a resulting list of all of the evaluations of that function.

Python has a similar function called applymap.
In applymap, you provide some function which should operate on each cell of a DataFrame, and the return set is itself a DataFrame.
Now I think applymap is fine, but I actually rarely use it.
Instead, I find myself often wanting to map across all of the rows in a DataFrame.
And pandas has a function that I use heavily there, called apply.

Let's look at an example.

Let's take our census DataFrame.
In this DataFrame, we have five columns for population estimates. Each column corresponding with one year of estimates.

It's quite reasonable to want to create some new columns for minimum or maximum values, and the ***apply*** function is an easy way to do this.

First, we need to write a function which takes in a particular row of data, finds a minimum and maximum values, and returns a new row of data.
We'll call this function min_max, this is pretty straight forward.
We can create some small slice of a row by projecting the population columns.
Then use the NumPy min and max functions, and create a new series with a label values represent the new values we want to apply.

```
import numpy as np
def min_max(row):
    data = row[['POPESTIMATE2010',
                'POPESTIMATE2011',
                'POPESTIMATE2012',
                'POPESTIMATE2013',
                'POPESTIMATE2014',
                'POPESTIMATE2015']]
    return pd.Series({'min': np.min(data), 'max': np.max(data)})
```

Then we just need to call apply on the DataFrame.
Apply takes the function and the axis on which to operate as parameters.

```
df.apply(min_max, axis=1)
```

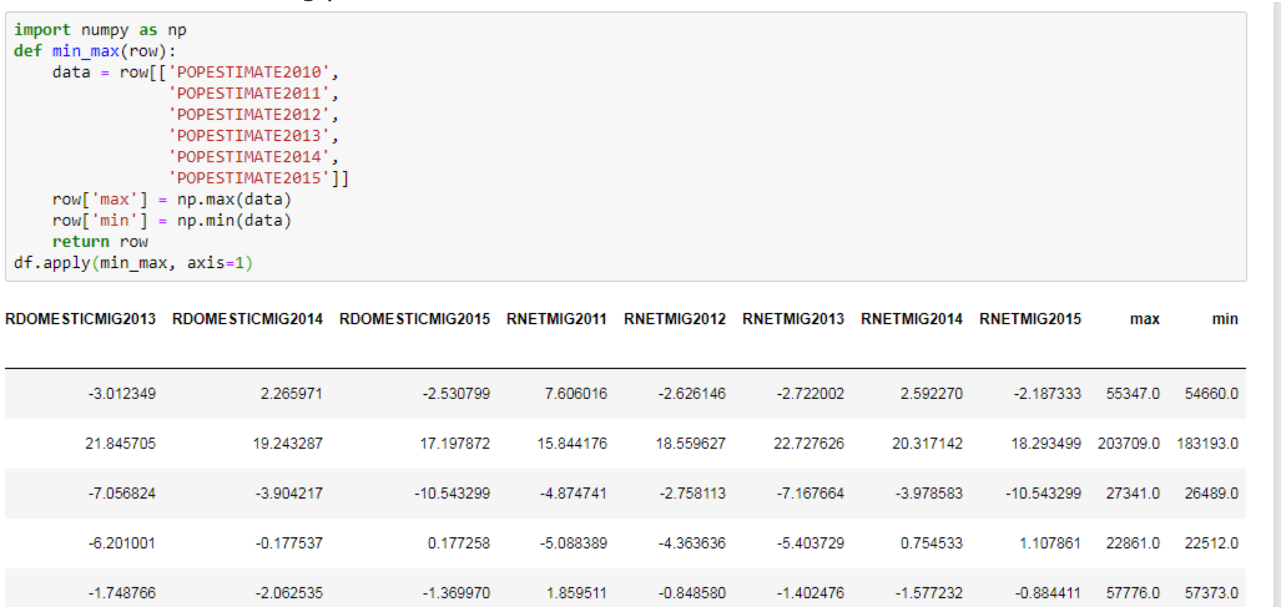
		min	max
STNAME	CTYNAME		
Alabama	Autauga County	54660.0	55347.0
	Baldwin County	183193.0	203709.0
	Barbour County	26489.0	27341.0
	Bibb County	22512.0	22861.0
	Blount County	57373.0	57776.0
...	...	...	...
Wyoming	Sweetwater County	43593.0	45162.0
	Teton County	21297.0	23125.0
	Uinta County	20822.0	21102.0
	Washakie County	8316.0	8545.0
	Weston County	7065.0	7234.0

3142 rows × 2 columns

Now, we have to be a bit careful, we've talked about axis zero being the rows of the DataFrame in the past.
But this parameter is really the parameter of the index to use.

So, to apply across all rows, you pass axis equal to one.
Of course there's no need to limit yourself to returning a new series object.
If you're doing this as part of data cleaning your likely to find yourself wanting to add new data to the existing DataFrame.
In that case you just take the row values and add in new columns indicating the max and minimum scores.
This is a regular part of my workflow when bringing in data and building summary or descriptive statistics. And is often used heavily with the merging of DataFrames.

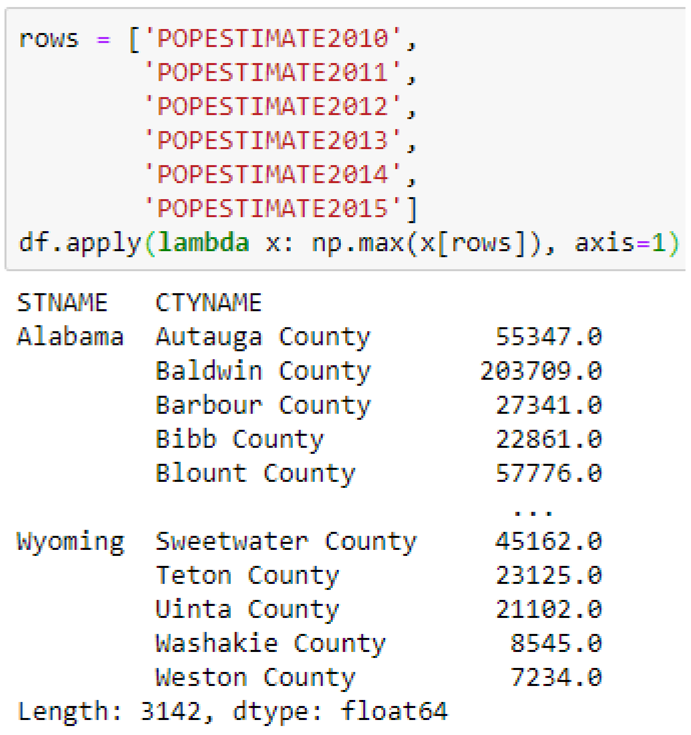
Okay, this is all great, and *apply* is an extremely important tool in your toolkit.
But this lecture wasn't really supposed to be about the new features of the API, but about making pandorable code.



```python
import numpy as np
def min_max(row):
    data = row[['POPESTIMATE2010',
                'POPESTIMATE2011',
                'POPESTIMATE2012',
                'POPESTIMATE2013',
                'POPESTIMATE2014',
                'POPESTIMATE2015']]
    row['max'] = np.max(data)
    row['min'] = np.min(data)
    return row
df.apply(min_max, axis=1)
```

RDOMESTICMIG2013	RDOMESTICMIG2014	RDOMESTICMIG2015	RNETMIG2011	RNETMIG2012	RNETMIG2013	RNETMIG2014	RNETMIG2015	max	min
-3.012349	2.265971	-2.530799	7.606016	-2.626146	-2.722002	2.592270	-2.187333	55347.0	54660.0
21.845705	19.243287	17.197872	15.844176	18.559627	22.727626	20.317142	18.293499	203709.0	183193.0
-7.056824	-3.904217	-10.543299	-4.874741	-2.758113	-7.167664	-3.978583	-10.543299	27341.0	26489.0
-6.201001	-0.177537	0.177258	-5.088389	-4.363636	-5.403729	0.754533	1.107861	22861.0	22512.0
-1.748766	-2.062535	-1.369970	1.859511	-0.848580	-1.402476	-1.577232	-0.884411	57776.0	57373.0

The reason I introduced *apply* here is because you rarely see it used with large function definitions, like we did.
Instead, you typically see it used with lambdas.
Now, you'll recall from the first week of this course that knowing lambda isn't a requirement of the course.
But to get the most of the discussions you'll see online, you're going to need to know how at least read lambdas.
Here's a one line example of how you might calculate the max of the columns using the apply function.
You can imagine how you might chain several apply calls with lambdas together to create a readable yet succinct data manipulation script.



```python
rows = ['POPESTIMATE2010',
        'POPESTIMATE2011',
        'POPESTIMATE2012',
        'POPESTIMATE2013',
        'POPESTIMATE2014',
        'POPESTIMATE2015']
df.apply(lambda x: np.max(x[rows]), axis=1)
```

```
STNAME   CTYNAME
Alabama  Autauga County        55347.0
         Baldwin County       203709.0
         Barbour County        27341.0
         Bibb County           22861.0
         Blount County         57776.0
                                ...
Wyoming  Sweetwater County     45162.0
         Teton County          23125.0
         Uinta County          21102.0
         Washakie County        8545.0
         Weston County          7234.0
Length: 3142, dtype: float64
```

So there are a couple of pandas idioms.
But I think there's many more, and I haven't talked about them here.
So here's an unofficial assignment for you.
Go look at some of the top ranked questions on pandas on Stack Overflow, and look at how some of the more experienced authors, answer those questions.
Do you see any interesting patterns?