03-02: Being Pandorable (Pandas Idiomatic Syntax)

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 StackOverflow and other websites.

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.

In [1]:

```
# Let's start by bringing in our data processing libraries
import pandas as pd
import numpy as np
# And we'll bring in some timing functionality too, from the timeit module
import timeit

# And lets look at some census data from the US
df = pd.read_csv('datasets/census.csv')
df.head()
```

Out[1]:

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	E
0	40	3	6	1	0	Alabama	Alabama	4779736	
1	50	3	6	1	1	Alabama	Autauga County	54571	
2	50	3	6	1	3	Alabama	Baldwin County	182265	
3	50	3	6	1	5	Alabama	Barbour County	27457	
4	50	3	6	1	7	Alabama	Bibb County	22915	

5 rows × 100 columns

Time Readability TradeOff

Time Readability Tradeoff: Pandorable Approach

In [2]:

```
# The first of the pandas idioms I would like to talk about is called method cha
ining. 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 the pandorable way to write code with method chaining. In this code I'm
going to pull out the state
# and city names as a multiple index, and I'm going to do so only for data which
has a summary level of 50,
# which in this dataset is county-level data. I'll rename a column too, just to
 make it a bit more readable.
(df.where(df['SUMLEV']==50) # boolean mask
    .dropna() #to the result, drop nan values
    .set_index(['STNAME','CTYNAME']) # set new primary index 'stname' and a seco
ndary index "city name"
    .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))  # rename col
umns.
```

Out[2]:

STNAME	CTYNAME	SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Es
Alabama	Autauga County	50.0	3.0	6.0	1.0	1.0	54571.0	
	Baldwin County	50.0	3.0	6.0	1.0	3.0	182265.0	1
	Barbour County	50.0	3.0	6.0	1.0	5.0	27457.0	
	Bibb County	50.0	3.0	6.0	1.0	7.0	22915.0	
	Blount County	50.0	3.0	6.0	1.0	9.0	57322.0	
	Bullock County	50.0	3.0	6.0	1.0	11.0	10914.0	
	Butler County	50.0	3.0	6.0	1.0	13.0	20947.0	
	Calhoun County	50.0	3.0	6.0	1.0	15.0	118572.0	1
	Chambers County	50.0	3.0	6.0	1.0	17.0	34215.0	
	Cherokee County	50.0	3.0	6.0	1.0	19.0	25989.0	
	Chilton County	50.0	3.0	6.0	1.0	21.0	43643.0	
	Choctaw County	50.0	3.0	6.0	1.0	23.0	13859.0	
	Clarke County	50.0	3.0	6.0	1.0	25.0	25833.0	
	Clay County	50.0	3.0	6.0	1.0	27.0	13932.0	
	Cleburne County	50.0	3.0	6.0	1.0	29.0	14972.0	
	Coffee County	50.0	3.0	6.0	1.0	31.0	49948.0	
	Colbert County	50.0	3.0	6.0	1.0	33.0	54428.0	
	Conecuh County	50.0	3.0	6.0	1.0	35.0	13228.0	
	Coosa County	50.0	3.0	6.0	1.0	37.0	11539.0	
	Covington County	50.0	3.0	6.0	1.0	39.0	37765.0	
	Crenshaw County	50.0	3.0	6.0	1.0	41.0	13906.0	

STNAME	CTYNAME	SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Es
	Cullman County	50.0	3.0	6.0	1.0	43.0	80406.0	
	Dale County	50.0	3.0	6.0	1.0	45.0	50251.0	
	Dallas County	50.0	3.0	6.0	1.0	47.0	43820.0	
	DeKalb County	50.0	3.0	6.0	1.0	49.0	71109.0	
	Elmore County	50.0	3.0	6.0	1.0	51.0	79303.0	
	Escambia County	50.0	3.0	6.0	1.0	53.0	38319.0	
	Etowah County	50.0	3.0	6.0	1.0	55.0	104430.0	1
	Fayette County	50.0	3.0	6.0	1.0	57.0	17241.0	
	Franklin County	50.0	3.0	6.0	1.0	59.0	31704.0	
...	...	...	...	...	...	...	...	
Wisconsin	Washburn County	50.0	2.0	3.0	55.0	129.0	15911.0	
	Washington County	50.0	2.0	3.0	55.0	131.0	131887.0	1
	Waukesha County	50.0	2.0	3.0	55.0	133.0	389891.0	3
	Waupaca County	50.0	2.0	3.0	55.0	135.0	52410.0	
	Waushara County	50.0	2.0	3.0	55.0	137.0	24496.0	
	Winnebago County	50.0	2.0	3.0	55.0	139.0	166994.0	1
	Wood County	50.0	2.0	3.0	55.0	141.0	74749.0	
Wyoming	Albany County	50.0	4.0	8.0	56.0	1.0	36299.0	
	Big Horn County	50.0	4.0	8.0	56.0	3.0	11668.0	
	Campbell County	50.0	4.0	8.0	56.0	5.0	46133.0	
	Carbon County	50.0	4.0	8.0	56.0	7.0	15885.0	
	Converse County	50.0	4.0	8.0	56.0	9.0	13833.0	
	Crook County	50.0	4.0	8.0	56.0	11.0	7083.0	

STNAME	CTYNAME	SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Es
	Fremont County	50.0	4.0	8.0	56.0	13.0	40123.0	
	Goshen County	50.0	4.0	8.0	56.0	15.0	13249.0	
	Hot Springs County	50.0	4.0	8.0	56.0	17.0	4812.0	
	Johnson County	50.0	4.0	8.0	56.0	19.0	8569.0	
	Laramie County	50.0	4.0	8.0	56.0	21.0	91738.0	
	Lincoln County	50.0	4.0	8.0	56.0	23.0	18106.0	
	Natrona County	50.0	4.0	8.0	56.0	25.0	75450.0	
	Niobrara County	50.0	4.0	8.0	56.0	27.0	2484.0	
	Park County	50.0	4.0	8.0	56.0	29.0	28205.0	
	Platte County	50.0	4.0	8.0	56.0	31.0	8667.0	
	Sheridan County	50.0	4.0	8.0	56.0	33.0	29116.0	
	Sublette County	50.0	4.0	8.0	56.0	35.0	10247.0	
	Sweetwater County	50.0	4.0	8.0	56.0	37.0	43806.0	
	Teton County	50.0	4.0	8.0	56.0	39.0	21294.0	
	Uinta County	50.0	4.0	8.0	56.0	41.0	21118.0	
	Washakie County	50.0	4.0	8.0	56.0	43.0	8533.0	
	Weston County	50.0	4.0	8.0	56.0	45.0	7208.0	

3142 rows × 98 columns

Lets walk through this.

- First, we use the `where()` function on the dataframe and pass in a boolean mask which is only true for those rows where the SUMLEV is equal to 50.
- This indicates in our source data that the data is summarized at the county level. With the result of the where() function evaluated, we drop missing values. ***Remember that .where() doesn't drop missing values by default.***
- Then we set an index on the result of that___. In this case I've set it to the state name followed by the county name.
- Finally. I rename a column to make it more readable.

Pandorable: Note that instead of writing this all on one line, as I could have done, I began the statement with a parenthesis, which tells python I'm going to span the statement over multiple lines for readability.

Time Readability Tradeoff: Traditional Approach

In [3]:

```
# Here's a more traditional, non-pandorable way, of writing this. There's nothin
g wrong with this code in the
# functional sense, you might even be able to understand it better as a new pers
on to the language. It's just
# not as pandorable as the first example.

# First create a new dataframe from the original
df = df[df['SUMLEV']==50] # I'll use the overloaded indexing operator [] which d
rops nans
# Update the dataframe to have a new index, we use inplace=True to do this in pl
ace
df.set_index(['STNAME','CTYNAME'], inplace=True)
# Set the column names
df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})
```

Out[3]:

STNAME	CTYNAME	SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Es
Alabama	Autauga County	50	3	6	1	1	54571	
	Baldwin County	50	3	6	1	3	182265	
	Barbour County	50	3	6	1	5	27457	
	Bibb County	50	3	6	1	7	22915	
	Blount County	50	3	6	1	9	57322	
	Bullock County	50	3	6	1	11	10914	
	Butler County	50	3	6	1	13	20947	
	Calhoun County	50	3	6	1	15	118572	
	Chambers County	50	3	6	1	17	34215	
	Cherokee County	50	3	6	1	19	25989	
	Chilton County	50	3	6	1	21	43643	
	Choctaw County	50	3	6	1	23	13859	
	Clarke County	50	3	6	1	25	25833	
	Clay County	50	3	6	1	27	13932	
	Cleburne County	50	3	6	1	29	14972	
	Coffee County	50	3	6	1	31	49948	
	Colbert County	50	3	6	1	33	54428	
	Conecuh County	50	3	6	1	35	13228	
	Coosa County	50	3	6	1	37	11539	
	Covington County	50	3	6	1	39	37765	
	Crenshaw County	50	3	6	1	41	13906	

STNAME	CTYNAME	SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Es
	Cullman County	50	3	6	1	43	80406	
	Dale County	50	3	6	1	45	50251	
	Dallas County	50	3	6	1	47	43820	
	DeKalb County	50	3	6	1	49	71109	
	Elmore County	50	3	6	1	51	79303	
	Escambia County	50	3	6	1	53	38319	
	Etowah County	50	3	6	1	55	104430	
	Fayette County	50	3	6	1	57	17241	
	Franklin County	50	3	6	1	59	31704	
...	...	...	...	...	...	...	...	
Wisconsin	Washburn County	50	2	3	55	129	15911	
	Washington County	50	2	3	55	131	131887	
	Waukesha County	50	2	3	55	133	389891	
	Waupaca County	50	2	3	55	135	52410	
	Waushara County	50	2	3	55	137	24496	
	Winnebago County	50	2	3	55	139	166994	
	Wood County	50	2	3	55	141	74749	
Wyoming	Albany County	50	4	8	56	1	36299	
	Big Horn County	50	4	8	56	3	11668	
	Campbell County	50	4	8	56	5	46133	
	Carbon County	50	4	8	56	7	15885	
	Converse County	50	4	8	56	9	13833	
	Crook County	50	4	8	56	11	7083	

STNAME	CTYNAME	SUMLEV	REGION	DIVISION	STATE	COUNTY	CENSUS2010POP	Es
	Fremont County	50	4	8	56	13	40123	
	Goshen County	50	4	8	56	15	13249	
	Hot Springs County	50	4	8	56	17	4812	
	Johnson County	50	4	8	56	19	8569	
	Laramie County	50	4	8	56	21	91738	
	Lincoln County	50	4	8	56	23	18106	
	Natrona County	50	4	8	56	25	75450	
	Niobrara County	50	4	8	56	27	2484	
	Park County	50	4	8	56	29	28205	
	Platte County	50	4	8	56	31	8667	
	Sheridan County	50	4	8	56	33	29116	
	Sublette County	50	4	8	56	35	10247	
	Sweetwater County	50	4	8	56	37	43806	
	Teton County	50	4	8	56	39	21294	
	Uinta County	50	4	8	56	41	21118	
	Washakie County	50	4	8	56	43	8533	
	Weston County	50	4	8	56	45	7208	

3142 rows × 98 columns

Time Readability Tradeoff: Time Analysis

In [4]:

```
# Now, the key with any good idiom is to understand when it isn't helping you. I
n this case, you can actually
# time both methods and see which one runs faster

# We can put the approach into a function and pass the function into the timeit
 function to count the time the
# parameter number allows us to choose how many times we want to run the functio
n. Here we will just set it to
# 10

# Lets write a wrapper for our first function
def first_approach():
    global df
    # And we'll just paste our code right here
    return (df.where(df['SUMLEV']==50)
              .dropna()
              .set_index(['STNAME','CTYNAME'])
              .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))

# Read in our dataset anew
df = pd.read_csv('datasets/census.csv')

# And now lets run it
timeit.timeit(first_approach, number=10)
```

Out[4]:

```
0.48141226917505264
```

In [5]:

```
# Now let's test the second approach. As you may notice, we use our global varia
ble df in the function.
# However, changing a global variable inside a function will modify the variable
even in a global scope and we
# do not want that to happen in this case. Therefore, for selecting summary leve
ls of 50 only, I create a new
# dataframe for those records

# Let's run this for once and see how fast it is
def second_approach():
    global df
    new_df = df[df['SUMLEV']==50]
    new_df.set_index(['STNAME','CTYNAME'], inplace=True)
    return new_df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})

# Read in our dataset anew
df = pd.read_csv('datasets/census.csv')

# And now lets run it
timeit.timeit(second_approach, number=10)
```

Out[5]:

```
0.05346250161528587
```

As you can see, the second approach is much faster! So, this is a particular example of a classic _**time readability trade off**__.

You'll see lots of examples on stack 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. But keep in mind that following what appears to be stylistic idioms might have performance issues that you need to consider as well.

Pandas: `apply()`

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.

Pandas 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.

In [6]:

```
# Let's take a look at our census DataFrame. In this DataFrame, we have five col
umns for population estimates,
# with each column corresponding with one year of estimates. It's quite reasonab
le to want to create some new
# columns for minimum or maximum values, and the apply function is an easy way t
o do this.

# First, we need to write a function which takes in a particular row of data, fi
nds a minimum and maximum
# values, and returns a new row of data nd returns a new row of data.  We'll cal
l this function min_max, this
# is pretty straight forward. We can create some small slice of a row by project
ing 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.

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. 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**, ***which is applying on all columns***, you pass axis equal to 'columns'.

In [7]:

```
df.apply(min_max, axis='columns').head()
```

Out[7]:

	min	max
0	4785161	4858979
1	54660	55347
2	183193	203709
3	26489	27341
4	22512	22861

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.

Pandas `apply()` : Creating More Columns In Database

In [8]:

```
# Here's an example where we have a revised version of the function min_max Inst
ead of returning a separate
# series to display the min and max we add two new columns in the original dataf
rame to store min and max

def min_max(row):
    data = row[['POPESTIMATE2010',
                'POPESTIMATE2011',
                'POPESTIMATE2012',
                'POPESTIMATE2013',
                'POPESTIMATE2014',
                'POPESTIMATE2015']]
    # Create a new entry for max
    row['max'] = np.max(data)
    # Create a new entry for min
    row['min'] = np.min(data)
    return row
# Now just apply the function across the dataframe
df.apply(min_max, axis='columns')
```

Out[8]:

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010PO
0	40	3	6	1	0	Alabama	Alabama	477973
1	50	3	6	1	1	Alabama	Autauga County	5457
2	50	3	6	1	3	Alabama	Baldwin County	18226
3	50	3	6	1	5	Alabama	Barbour County	2745
4	50	3	6	1	7	Alabama	Bibb County	2291
5	50	3	6	1	9	Alabama	Blount County	5732
6	50	3	6	1	11	Alabama	Bullock County	1091
7	50	3	6	1	13	Alabama	Butler County	2094
8	50	3	6	1	15	Alabama	Calhoun County	11857
9	50	3	6	1	17	Alabama	Chambers County	3421
10	50	3	6	1	19	Alabama	Cherokee County	2598
11	50	3	6	1	21	Alabama	Chilton County	4364
12	50	3	6	1	23	Alabama	Choctaw County	1385
13	50	3	6	1	25	Alabama	Clarke County	2583
14	50	3	6	1	27	Alabama	Clay County	1393
15	50	3	6	1	29	Alabama	Cleburne County	1497
16	50	3	6	1	31	Alabama	Coffee County	4994
17	50	3	6	1	33	Alabama	Colbert County	5442
18	50	3	6	1	35	Alabama	Conecuh County	1322
19	50	3	6	1	37	Alabama	Coosa County	1153
20	50	3	6	1	39	Alabama	Covington County	3776
21	50	3	6	1	41	Alabama	Crenshaw County	1390
22	50	3	6	1	43	Alabama	Cullman County	8040
23	50	3	6	1	45	Alabama	Dale County	5025

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010PO
24	50	3	6	1	47	Alabama	Dallas County	4382
25	50	3	6	1	49	Alabama	DeKalb County	7110
26	50	3	6	1	51	Alabama	Elmore County	7930
27	50	3	6	1	53	Alabama	Escambia County	3831
28	50	3	6	1	55	Alabama	Etowah County	10443
29	50	3	6	1	57	Alabama	Fayette County	1724
...	...	...	...	...	...	...	...	.
3163	50	2	3	55	131	Wisconsin	Washington County	13188
3164	50	2	3	55	133	Wisconsin	Waukesha County	38989
3165	50	2	3	55	135	Wisconsin	Waupaca County	5241
3166	50	2	3	55	137	Wisconsin	Waushara County	2449
3167	50	2	3	55	139	Wisconsin	Winnebago County	16699
3168	50	2	3	55	141	Wisconsin	Wood County	7474
3169	40	4	8	56	0	Wyoming	Wyoming	56362
3170	50	4	8	56	1	Wyoming	Albany County	3629
3171	50	4	8	56	3	Wyoming	Big Horn County	1166
3172	50	4	8	56	5	Wyoming	Campbell County	4613
3173	50	4	8	56	7	Wyoming	Carbon County	1588
3174	50	4	8	56	9	Wyoming	Converse County	1383
3175	50	4	8	56	11	Wyoming	Crook County	708
3176	50	4	8	56	13	Wyoming	Fremont County	4012
3177	50	4	8	56	15	Wyoming	Goshen County	1324
3178	50	4	8	56	17	Wyoming	Hot Springs County	481
3179	50	4	8	56	19	Wyoming	Johnson County	856
3180	50	4	8	56	21	Wyoming	Laramie County	9173
3181	50	4	8	56	23	Wyoming	Lincoln County	1810

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010PO
3182	50	4	8	56	25	Wyoming	Natrona County	7545
3183	50	4	8	56	27	Wyoming	Niobrara County	248
3184	50	4	8	56	29	Wyoming	Park County	2820
3185	50	4	8	56	31	Wyoming	Platte County	866
3186	50	4	8	56	33	Wyoming	Sheridan County	2911
3187	50	4	8	56	35	Wyoming	Sublette County	1024
3188	50	4	8	56	37	Wyoming	Sweetwater County	4380
3189	50	4	8	56	39	Wyoming	Teton County	2129
3190	50	4	8	56	41	Wyoming	Uinta County	2111
3191	50	4	8	56	43	Wyoming	Washakie County	853
3192	50	4	8	56	45	Wyoming	Weston County	720

3193 rows × 102 columns

Pandas `apply()` : A More Pandorable Approach (using vectorisation)

In [9]:

```
# Apply is an extremely important tool in your toolkit. 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. To get the most of the discussions you'll see online, you're going to
need to know how to at least
# read lambdas.

# Here's You can imagine how you might chain several apply calls with lambdas to
gether to create a readable
# yet succinct data manipulation script. One line example of how you might calcu
late the max of the columns
# using the apply function.
rows = ['POPESTIMATE2010', 'POPESTIMATE2011', 'POPESTIMATE2012', 'POPESTIMATE201
3','POPESTIMATE2014',
        'POPESTIMATE2015']
# Now we'll just apply this across the dataframe with a lambda
df.apply(lambda x: np.max(x[rows]), axis=1).head()
```

Out[9]:

```
0    4858979
1      55347
2     203709
3      27341
4      22861
dtype: int64
```

In [ ]:

```
# If you don't remember lambdas just pause the video for a moment and look up th
e syntax. A lambda is just an
# unnamed function in python, in this case it takes a single parameter, x, and r
eturns a single value, in this
# case the maximum over all columns associated with row x.
```

In [10]:

```
# The beauty of the apply function is that it allows flexibility in doing whatev
er manipulation that you
# desire, as the function you pass into apply can be any customized however you
 want. Let's say we want to
# divide the states into four categories: Northeast, Midwest, South, and West We
can write a customized
# function that returns the region based on the state the state regions informat
ion is obtained from Wikipedia

def get_state_region(x):
    northeast = ['Connecticut', 'Maine', 'Massachusetts', 'New Hampshire',
                 'Rhode Island','Vermont','New York','New Jersey','Pennsylvania'
]
    midwest = ['Illinois','Indiana','Michigan','Ohio','Wisconsin','Iowa',
               'Kansas','Minnesota','Missouri','Nebraska','North Dakota',
               'South Dakota']
    south = ['Delaware','Florida','Georgia','Maryland','North Carolina',
             'South Carolina','Virginia','District of Columbia','West Virginia',
             'Alabama','Kentucky','Mississippi','Tennessee','Arkansas',
             'Louisiana','Oklahoma','Texas']
    west = ['Arizona','Colorado','Idaho','Montana','Nevada','New Mexico','Utah',
            'Wyoming','Alaska','California','Hawaii','Oregon','Washington']

    if x in northeast:
        return "Northeast"
    elif x in midwest:
        return "Midwest"
    elif x in south:
        return "South"
    else:
        return "West"
```

In [11]:

```
# Now we have the customized function, let's say we want to create a new column
 called Region, which shows the
# state's region, we can use the customized function and the apply function to d
o so. The customized function
# is supposed to work on the state name column STNAME. So we will set the apply
 function on the state name
# column and pass the customized function into the apply function
df['state_region'] = df['STNAME'].apply(lambda x: get_state_region(x))
```

In [12]:

```
# Now let's see the results
df[['STNAME','state_region']].head()
```

Out[12]:

	STNAME	state_region
0	Alabama	South
1	Alabama	South
2	Alabama	South
3	Alabama	South
4	Alabama	South

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? Feel free to share them with myself and others in the class.