PandasIdioms_ed

March 1, 2021

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.

[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()
```

[1]:
```
   SUMLEV  REGION  DIVISION  STATE  COUNTY   STNAME         CTYNAME  \
0      40       3         6      1       0  Alabama         Alabama
1      50       3         6      1       1  Alabama  Autauga County
2      50       3         6      1       3  Alabama  Baldwin County
3      50       3         6      1       5  Alabama  Barbour County
4      50       3         6      1       7  Alabama     Bibb County

   CENSUS2010POP  ESTIMATESBASE2010  POPESTIMATE2010  ...  RDOMESTICMIG2011  \
0        4779736            4780127          4785161  ...          0.002295
1          54571              54571            54660  ...          7.242091
2         182265             182265           183193  ...         14.832960
3          27457              27457            27341  ...         -4.728132
4          22915              22919            22861  ...         -5.527043

   RDOMESTICMIG2012  RDOMESTICMIG2013  RDOMESTICMIG2014  RDOMESTICMIG2015  \
0         -0.193196          0.381066          0.582002         -0.467369
1         -2.915927         -3.012349          2.265971         -2.530799
2         17.647293         21.845705         19.243287         17.197872
```

```
3        -2.500690        -7.056824        -3.904217        -10.543299
4        -5.068871        -6.201001        -0.177537          0.177258

   RNETMIG2011  RNETMIG2012  RNETMIG2013  RNETMIG2014  RNETMIG2015
0     1.030015     0.826644     1.383282     1.724718     0.712594
1     7.606016    -2.626146    -2.722002     2.592270    -2.187333
2    15.844176    18.559627    22.727626    20.317142    18.293499
3    -4.874741    -2.758113    -7.167664    -3.978583   -10.543299
4    -5.088389    -4.363636    -5.403729     0.754533     1.107861

[5 rows x 100 columns]
```

[2]:
```
# The first of the pandas idioms I would like to talk about is called method␣
 ↪chaining. 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)
    .dropna()
    .set_index(['STNAME','CTYNAME'])
    .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))
```

[2]:
```
                           SUMLEV  REGION  DIVISION  STATE  COUNTY  \
STNAME  CTYNAME
Alabama Autauga County       50.0     3.0       6.0    1.0     1.0
        Baldwin County       50.0     3.0       6.0    1.0     3.0
        Barbour County       50.0     3.0       6.0    1.0     5.0
        Bibb County          50.0     3.0       6.0    1.0     7.0
        Blount County        50.0     3.0       6.0    1.0     9.0
...                           ...     ...       ...    ...     ...
Wyoming Sweetwater County    50.0     4.0       8.0   56.0    37.0
        Teton County         50.0     4.0       8.0   56.0    39.0
        Uinta County         50.0     4.0       8.0   56.0    41.0
        Washakie County      50.0     4.0       8.0   56.0    43.0
        Weston County        50.0     4.0       8.0   56.0    45.0

                           CENSUS2010POP  Estimates Base 2010  \
STNAME  CTYNAME
```

```
Alabama Autauga County          54571.0             54571.0
        Baldwin County         182265.0            182265.0
        Barbour County          27457.0             27457.0
        Bibb County             22915.0             22919.0
        Blount County           57322.0             57322.0
...                                 ...                 ...
Wyoming Sweetwater County       43806.0             43806.0
        Teton County            21294.0             21294.0
        Uinta County            21118.0             21118.0
        Washakie County          8533.0              8533.0
        Weston County            7208.0              7208.0

                           POPESTIMATE2010  POPESTIMATE2011  POPESTIMATE2012  \
STNAME  CTYNAME
Alabama Autauga County             54660.0          55253.0          55175.0
        Baldwin County            183193.0         186659.0         190396.0
        Barbour County             27341.0          27226.0          27159.0
        Bibb County                22861.0          22733.0          22642.0
        Blount County              57373.0          57711.0          57776.0
...                                    ...              ...              ...
Wyoming Sweetwater County          43593.0          44041.0          45104.0
        Teton County               21297.0          21482.0          21697.0
        Uinta County               21102.0          20912.0          20989.0
        Washakie County             8545.0           8469.0           8443.0
        Weston County               7181.0           7114.0           7065.0

                           ...  RDOMESTICMIG2011  RDOMESTICMIG2012  \
STNAME  CTYNAME            ...
Alabama Autauga County     ...          7.242091         -2.915927
        Baldwin County     ...         14.832960         17.647293
        Barbour County     ...         -4.728132         -2.500690
        Bibb County        ...         -5.527043         -5.068871
        Blount County      ...          1.807375         -1.177622
...                        ...               ...               ...
Wyoming Sweetwater County  ...          1.072643         16.243199
        Teton County       ...         -1.589565          0.972695
        Uinta County       ...        -17.755986         -4.916350
        Washakie County    ...        -11.637475         -0.827815
        Weston County      ...        -11.752361         -8.040059

                           RDOMESTICMIG2013  RDOMESTICMIG2014  \
STNAME  CTYNAME
Alabama Autauga County            -3.012349          2.265971
        Baldwin County            21.845705         19.243287
        Barbour County            -7.056824         -3.904217
        Bibb County               -6.201001         -0.177537
        Blount County             -1.748766         -2.062535
```

```
...                                            ...          ...
Wyoming Sweetwater County          -5.339774   -14.252889
        Teton County               19.525929    14.143021
        Uinta County               -6.902954   -14.215862
        Washakie County            -2.013502   -17.781491
        Weston County              12.372583     1.533635

                         RDOMESTICMIG2015  RNETMIG2011  RNETMIG2012  \
STNAME  CTYNAME
Alabama Autauga County          -2.530799     7.606016    -2.626146
        Baldwin County          17.197872    15.844176    18.559627
        Barbour County         -10.543299    -4.874741    -2.758113
        Bibb County              0.177258    -5.088389    -4.363636
        Blount County           -1.369970     1.859511    -0.848580
...                                   ...          ...          ...
Wyoming Sweetwater County      -14.248864     1.255221    16.243199
        Teton County            -0.564849     0.654527     2.408578
        Uinta County           -12.127022   -18.136812    -5.536861
        Washakie County          1.682288   -11.990126    -1.182592
        Weston County            6.935294   -12.032179    -8.040059

                         RNETMIG2013  RNETMIG2014  RNETMIG2015
STNAME  CTYNAME
Alabama Autauga County     -2.722002     2.592270    -2.187333
        Baldwin County     22.727626    20.317142    18.293499
        Barbour County     -7.167664    -3.978583   -10.543299
        Bibb County        -5.403729     0.754533     1.107861
        Blount County      -1.402476    -1.577232    -0.884411
...                              ...          ...          ...
Wyoming Sweetwater County  -5.295460   -14.075283   -14.070195
        Teton County       21.160658    16.308671     1.520747
        Uinta County       -7.521840   -14.740608   -12.606351
        Washakie County    -2.250385   -18.020168     1.441961
        Weston County      12.372583     1.533635     6.935294

[3142 rows x 98 columns]
```

[3]:
```
# 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. 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.
```

[4]:
```
# Here's a more traditional, non-pandorable way, of writing this. 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. 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␣
↪drops nans
# Update the dataframe to have a new index, we use inplace=True to do this in␣
↪place
df.set_index(['STNAME','CTYNAME'], inplace=True)
# Set the column names
df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})
```

[4]:
```
                           SUMLEV  REGION  DIVISION  STATE  COUNTY  \
STNAME  CTYNAME
Alabama Autauga County         50       3         6      1       1
        Baldwin County         50       3         6      1       3
        Barbour County         50       3         6      1       5
        Bibb County            50       3         6      1       7
        Blount County          50       3         6      1       9
...                           ...     ...       ...    ...     ...
Wyoming Sweetwater County      50       4         8     56      37
        Teton County           50       4         8     56      39
        Uinta County           50       4         8     56      41
        Washakie County        50       4         8     56      43
        Weston County          50       4         8     56      45

                           CENSUS2010POP  Estimates Base 2010  \
STNAME  CTYNAME
Alabama Autauga County             54571                54571
        Baldwin County            182265               182265
        Barbour County             27457                27457
        Bibb County                22915                22919
        Blount County              57322                57322
...                                  ...                  ...
Wyoming Sweetwater County          43806                43806
        Teton County               21294                21294
        Uinta County               21118                21118
        Washakie County             8533                 8533
        Weston County            7208                7208

                           POPESTIMATE2010  POPESTIMATE2011  POPESTIMATE2012  \
STNAME  CTYNAME
Alabama Autauga County               54660            55253            55175
        Baldwin County              183193           186659           190396
        Barbour County               27341            27226            27159
        Bibb County                  22861            22733            22642
        Blount County                57373            57711            57776
...                                    ...              ...              ...
Wyoming Sweetwater County            43593            44041            45104
        Teton County                 21297            21482            21697
        Uinta County                 21102            20912            20989
        Washakie County               8545             8469             8443
        Weston County                 7181             7114             7065

                           ...  RDOMESTICMIG2011  RDOMESTICMIG2012  \
STNAME  CTYNAME            ...
Alabama Autauga County     ...          7.242091         -2.915927
        Baldwin County     ...         14.832960         17.647293
        Barbour County     ...         -4.728132         -2.500690
        Bibb County        ...         -5.527043         -5.068871
        Blount County      ...          1.807375         -1.177622
...                        ...               ...               ...
Wyoming Sweetwater County  ...          1.072643         16.243199
        Teton County       ...         -1.589565          0.972695
        Uinta County       ...        -17.755986         -4.916350
        Washakie County    ...        -11.637475         -0.827815
        Weston County      ...        -11.752361         -8.040059

                           RDOMESTICMIG2013  RDOMESTICMIG2014  \
STNAME  CTYNAME
Alabama Autauga County            -3.012349          2.265971
        Baldwin County            21.845705         19.243287
        Barbour County            -7.056824         -3.904217
        Bibb County               -6.201001         -0.177537
        Blount County             -1.748766         -2.062535
...                                     ...               ...
Wyoming Sweetwater County         -5.339774        -14.252889
        Teton County              19.525929         14.143021
        Uinta County              -6.902954        -14.215862
        Washakie County           -2.013502        -17.781491
        Weston County             12.372583          1.533635

                           RDOMESTICMIG2015  RNETMIG2011  RNETMIG2012  \
STNAME  CTYNAME
Alabama Autauga County            -2.530799     7.606016    -2.626146
```

```
        Baldwin County           17.197872    15.844176    18.559627
        Barbour County          -10.543299    -4.874741    -2.758113
        Bibb County               0.177258    -5.088389    -4.363636
        Blount County            -1.369970     1.859511    -0.848580
...                                    ...          ...          ...
Wyoming Sweetwater County       -14.248864     1.255221    16.243199
        Teton County             -0.564849     0.654527     2.408578
        Uinta County            -12.127022   -18.136812    -5.536861
        Washakie County           1.682288   -11.990126    -1.182592
        Weston County             6.935294   -12.032179    -8.040059

                           RNETMIG2013  RNETMIG2014  RNETMIG2015
STNAME  CTYNAME
Alabama Autauga County       -2.722002     2.592270    -2.187333
        Baldwin County       22.727626    20.317142    18.293499
        Barbour County       -7.167664    -3.978583   -10.543299
        Bibb County          -5.403729     0.754533     1.107861
        Blount County        -1.402476    -1.577232    -0.884411
...                                ...          ...          ...
Wyoming Sweetwater County    -5.295460   -14.075283   -14.070195
        Teton County         21.160658    16.308671     1.520747
        Uinta County         -7.521840   -14.740608   -12.606351
        Washakie County      -2.250385   -18.020168     1.441961
        Weston County        12.372583     1.533635     6.935294

[3142 rows x 98 columns]
```

[5]:
```
# 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 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␣
↪function. 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)
```

[5]: 1.1084833510685712

[6]:
```
# Now let's test the second approach. As you may notice, we use our global␣
 ↪variable 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␣
 ↪levels 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)
```

[6]: 0.10386669298168272

[7]:
```
# 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.
```

[8]:
```
# 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.
```

[9]:
```
# Let's take a look at our census DataFrame. In this DataFrame, we have five␣
↪columns for population estimates,
# with 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 nd 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.

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

[10]:
```
# 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'.
df.apply(min_max, axis='columns').head()
```

[10]:
```
       min      max
0  4785161  4858979
1    54660    55347
2   183193   203709
3    26489    27341
4    22512    22861
```

[11]:
```
# 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.
```

[12]:
```
# Here's an example where we have a revised version of the function min_max␣
 ↪Instead of returning a separate
# series to display the min and max we add two new columns in the original␣
 ↪dataframe 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')
```

[12]:
```
      SUMLEV  REGION  DIVISION  STATE  COUNTY   STNAME            CTYNAME  \
0         40       3         6      1       0  Alabama            Alabama
1         50       3         6      1       1  Alabama     Autauga County
2         50       3         6      1       3  Alabama     Baldwin County
3         50       3         6      1       5  Alabama     Barbour County
4         50       3         6      1       7  Alabama        Bibb County
...      ...     ...       ...    ...     ...      ...                ...
3188      50       4         8     56      37  Wyoming  Sweetwater County
```

```
3189      50      4        8     56      39  Wyoming       Teton County
3190      50      4        8     56      41  Wyoming       Uinta County
3191      50      4        8     56      43  Wyoming    Washakie County
3192      50      4        8     56      45  Wyoming      Weston County

      CENSUS2010POP  ESTIMATESBASE2010  POPESTIMATE2010  ...  \
0           4779736            4780127          4785161  ...
1             54571              54571            54660  ...
2            182265             182265           183193  ...
3             27457              27457            27341  ...
4             22915              22919            22861  ...
...             ...                ...              ...  ...
3188          43806              43806            43593  ...
3189          21294              21294            21297  ...
3190          21118              21118            21102  ...
3191           8533               8533             8545  ...
3192           7208               7208             7181  ...

      RDOMESTICMIG2013  RDOMESTICMIG2014  RDOMESTICMIG2015  RNETMIG2011  \
0             0.381066          0.582002         -0.467369     1.030015
1            -3.012349          2.265971         -2.530799     7.606016
2            21.845705         19.243287         17.197872    15.844176
3            -7.056824         -3.904217        -10.543299    -4.874741
4            -6.201001         -0.177537          0.177258    -5.088389
...                ...               ...               ...          ...
3188         -5.339774        -14.252889        -14.248864     1.255221
3189         19.525929         14.143021         -0.564849     0.654527
3190         -6.902954        -14.215862        -12.127022   -18.136812
3191         -2.013502        -17.781491          1.682288   -11.990126
3192         12.372583          1.533635          6.935294   -12.032179

      RNETMIG2012  RNETMIG2013  RNETMIG2014  RNETMIG2015      max      min
0        0.826644     1.383282     1.724718     0.712594  4858979  4785161
1       -2.626146    -2.722002     2.592270    -2.187333    55347    54660
2       18.559627    22.727626    20.317142    18.293499   203709   183193
3       -2.758113    -7.167664    -3.978583   -10.543299    27341    26489
4       -4.363636    -5.403729     0.754533     1.107861    22861    22512
...           ...          ...          ...          ...      ...      ...
3188    16.243199    -5.295460   -14.075283   -14.070195    45162    43593
3189     2.408578    21.160658    16.308671     1.520747    23125    21297
3190    -5.536861    -7.521840   -14.740608   -12.606351    21102    20822
3191    -1.182592    -2.250385   -18.020168     1.441961     8545     8316
3192    -8.040059    12.372583     1.533635     6.935294     7234     7065

[3193 rows x 102 columns]
```

[13]:
```
# 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␣
↪together to create a readable
# yet succinct data manipulation script. One line example of how you might␣
↪calculate the max of the columns
# using the apply function.
rows = ['POPESTIMATE2010', 'POPESTIMATE2011', 'POPESTIMATE2012',␣
↪'POPESTIMATE2013','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()
```

[13]:
```
0    4858979
1      55347
2     203709
3      27341
4      22861
dtype: int64
```

[14]:
```
# If you don't remember lambdas just pause the video for a moment and look up␣
↪the syntax. A lambda is just an
# unnamed function in python, in this case it takes a single parameter, x, and␣
↪returns a single value, in this
# case the maximum over all columns associated with row x.
```

[15]:
```
# The beauty of the apply function is that it allows flexibility in doing␣
↪whatever 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␣
↪information 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"
```

[16]:
```
# 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␣
 ↪do 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))
```

[17]:
```
# Now let's see the results
df[['STNAME','state_region']].head()
```

[17]:
```
    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.