DataFrameManipulation_ed

November 7, 2020

Now that you know the basics of what makes up a pandas dataframe, lets look at how we might actually clean some messy data. Now, there are many different approaches you can take to clean data, so this lecture is just one example of how you might tackle a problem.

[2]:
```
import pandas as pd
dfs=pd.read_html("https://en.wikipedia.org/wiki/
 ↪College_admissions_in_the_United_States")
len(dfs)
```

[2]:
```
14
```

[15]:
```
dfs[10]
```

[15]:

	University	St	Ap-plied#	Over-allrate	Earlyrate	Regu-larrate
0	Amherst	MA	5511	12%	NaN	10%
1	Babson	MA	5511	29%	53%	21%
2	Barnard	NY	5440	21%	45%	18%
3	SUNY-Bing.	NY	28174	42%	56%	36%
4	Boston Coll	MA	34000	29%	40%	26%
5	Boston U	MA	43979	45%	47%	44%
6	Bowdoin	ME	6716	16%	25%	14%
7	Brown	RI	28742	10%	19%	NaN
8	Caltech[237]	CA	5225	17%	19%	NaN
9	Carleton	MN	5850	26%	41%	26%
10	Carnegie-Mel	PA	17300	27%	26%	27%
11	Claremont Mc	CA	5056	12%	29%	10%
12	Colby	ME	5241	29%	50%	29%
13	Colgate	NY	7795	29%	51%	27%
14	Columbia	NY	31851	7%	20%	NaN
15	Cooper Union	NY	3556	6%	9%	6%
16	Cornell	NY	37812	16%	33%	NaN
17	Dartmouth	NH	23110	9%	26%	8%
18	Dickinson	PA	5843	40%	53%	28%
19	Duke	NC	31600	12%	25%	11%
20	Elon	NC	10195	51%	86%	29%
21	Emory	GA	17502	26%	38%	25%
22	G Washington	DC	21759	33%	38%	NaN
23	Grinnell	IA	4554	30%	58%	28%
24	Hamilton	NY	5107	27%	44%	NaN

25	Hanover	IN	3546	62%	NaN	62%
26	Harvard[238]	MA	34302	6%	18%	4%
27	Harvey Mudd	CA	3591	17%	20%	17%
28	Johns Hopkins	MD	20496	18%	38%	16%
29	Juilliard	NY	2319	7%	NaN	7%
..	...	..	...	...	...	...
51	Texas A&M	TX	31478	59%	NaN	59%
52	Trinity	CT	7716	33%	NaN	NaN
53	UC Berkeley	CA	61702	21%	NaN	NaN
54	UC Davis	CA	49416	46%	NaN	NaN
55	UC Irvine	CA	54532	36%	NaN	NaN
56	UCLA	CA	72657	21%	NaN	NaN
57	UC Merced	CA	13148	75%	NaN	NaN
58	UC Riverside	CA	29888	61%	NaN	NaN
59	UC San Diego	CA	60838	38%	NaN	NaN
60	UC Santa Bar	CA	54831	42%	NaN	NaN
61	UC Santa Cr.	CA	32954	60%	NaN	NaN
62	U Chicago	IL	25277	13%	18%	NaN
63	U Delaware	DE	26534	53%	NaN	53%
64	U Florida	FL	29220	41%	NaN	41%
65	UNC Chapel H	NC	28491	27%	38%	16%
66	U. Penn.	PA	31217	12%	25%	10%
67	U Puget Sou.	WA	6772	53%	88%	53%
68	U Rochester[243]	NY	17428	36%	NaN	NaN
69	USC	CA	46030	18%	NaN	18%
70	U. Virginia	VA	27200	29%	29%	23%
71	U. Wisc-Mad.	WI	29008	54%	54%	NaN
72	Vanderbilt	TN	28335	13%	25%	12%
73	Vassar	NY	7908	22%	43%	21%
74	Wake Forest	NC	11366	32%	43%	31%
75	Wash & Lee	VA	5970	18%	40%	15%
76	Washington U	MO	28826	15%	31%	17%
77	Wesleyan	CT	10503	20%	45%	18%
78	William & M.	VA	13651	31%	48%	29%
79	Williams	MA	7067	17%	NaN	NaN
80	Yale	CT	28974	7%	16%	5%

[81 rows x 6 columns]

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.

[ ]:
```
import pandas as pd
import numpy as np
import timeit

df = pd.read_csv('census.csv')
df.head()
```

[ ]:
```
# The first of these 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 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. In
# this code, there's no in place 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.
(df.where(df['SUMLEV']==50)
    .dropna()
    .set_index(['STNAME','CTYNAME'])
    .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))
```

[ ]:
```
# 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.
# It's just not as pandorable as the first example.

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

[ ]:
```
# 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 1

def first_approach():
    global df
    return (df.where(df['SUMLEV']==50)
              .dropna()
```

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

timeit.timeit(first_approach, number=1)
```

[ ]:
```
# Now let's test the second approach. As we 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'})
timeit.timeit(second_approach, number=1)
```

[ ]:
```
# 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 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 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)})
```

[ ]:
```
# 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 one.
df.apply(min_max, axis=1)
```

[ ]:
```
# 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.

# Here 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']]
    row['max'] = np.max(data)
    row['min'] = np.min(data)
```

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

[ ]:
```
# 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']
df.apply(lambda x: np.max(x[rows]), axis=1)
```

[ ]:
```
# The beauty of the apply function is that it allows flexibility in doing
# whatever manipulation that you desire, and the function you pass into apply
# can be any customized function that you write. 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"
```

[ ]:
```
# 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))
```

[ ]:
```
# Now let's see the results
df[['STNAME','state_region']]
```

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? Chime in on the course discussion forums and let's build some pandorable documents together.