Photo by chuttersnap on Unsplash

Apply and Lambda usage in pandas

Learn these to master Pandas

Rahul Agarwal Follow
Jul 1, 2019 · 6 min read ★

Pandas is a wonderful tool to have at your disposal.

I have been working with Pandas for years and it never ceases to amaze me with its new functionalities, shortcuts and multiple ways of doing a particular thing.

But I have realized that sticking to some of the conventions I have learned has served me well over the years.

`apply` and `lambda` are some of the best things I have learned to use with pandas.

I use `apply` and `lambda` anytime I get stuck while building a complex logic for a new column or filter.

And that happens a lot when the business comes to you with custom requests.

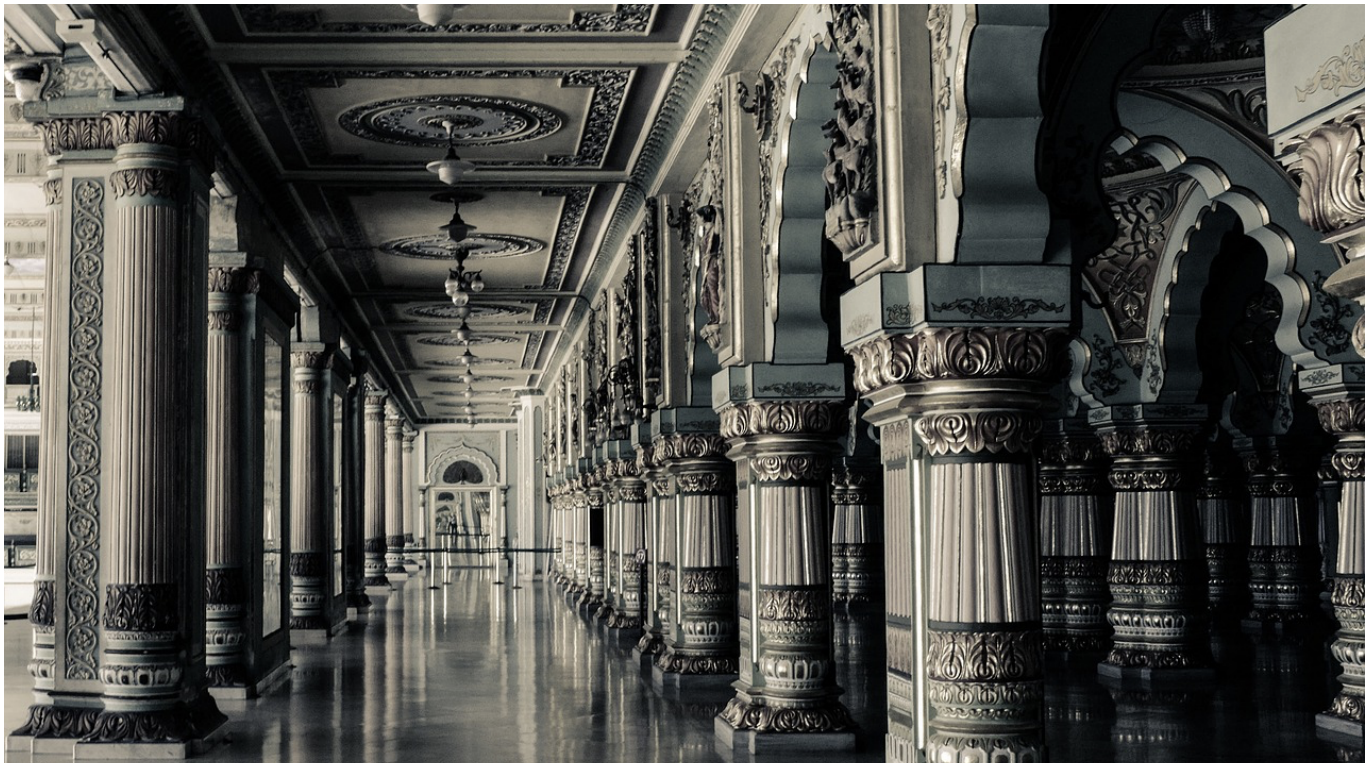
This post is about demonstrating the power of `apply` and `lambda` to you.

. . .

I will be using a data set of 1,000 popular movies on IMDB in the last 10 years. You can also follow along in the Kaggle Kernel.

. . .

Creating a Column



Complex columns

You can create a new column in many ways.

If you want a column that is a sum or difference of columns, you can pretty much use simple basic arithmetic. Here I get the average rating based on IMDB and Normalized Metascore.

```
df['AvgRating'] = (df['Rating'] + df['Metascore']/10)/2
```

But sometimes we may need to build complex logic around the creation of new columns.

To give you a convoluted example, let's say that we want to build a custom movie score based on a variety of factors.

Say, If the movie is of the thriller genre, I want to add 1 to the IMDB rating subject to the condition that IMDB rating remains less than or equal to 10. And If a movie is a comedy I want to subtract 1 from the rating.

How do we do that?

Whenever I get a hold of such complex problems, I use `apply/lambda`. Let me first show you how I will do this.

```
def custom_rating(genre,rating):
    if 'Thriller' in genre:
        return min(10,rating+1)
    elif 'Comedy' in genre:
        return max(0,rating-1)
    else:
        return rating

df['CustomRating'] = df.apply(lambda x:
custom_rating(x['Genre'],x['Rating']),axis=1)
```

The general structure is:

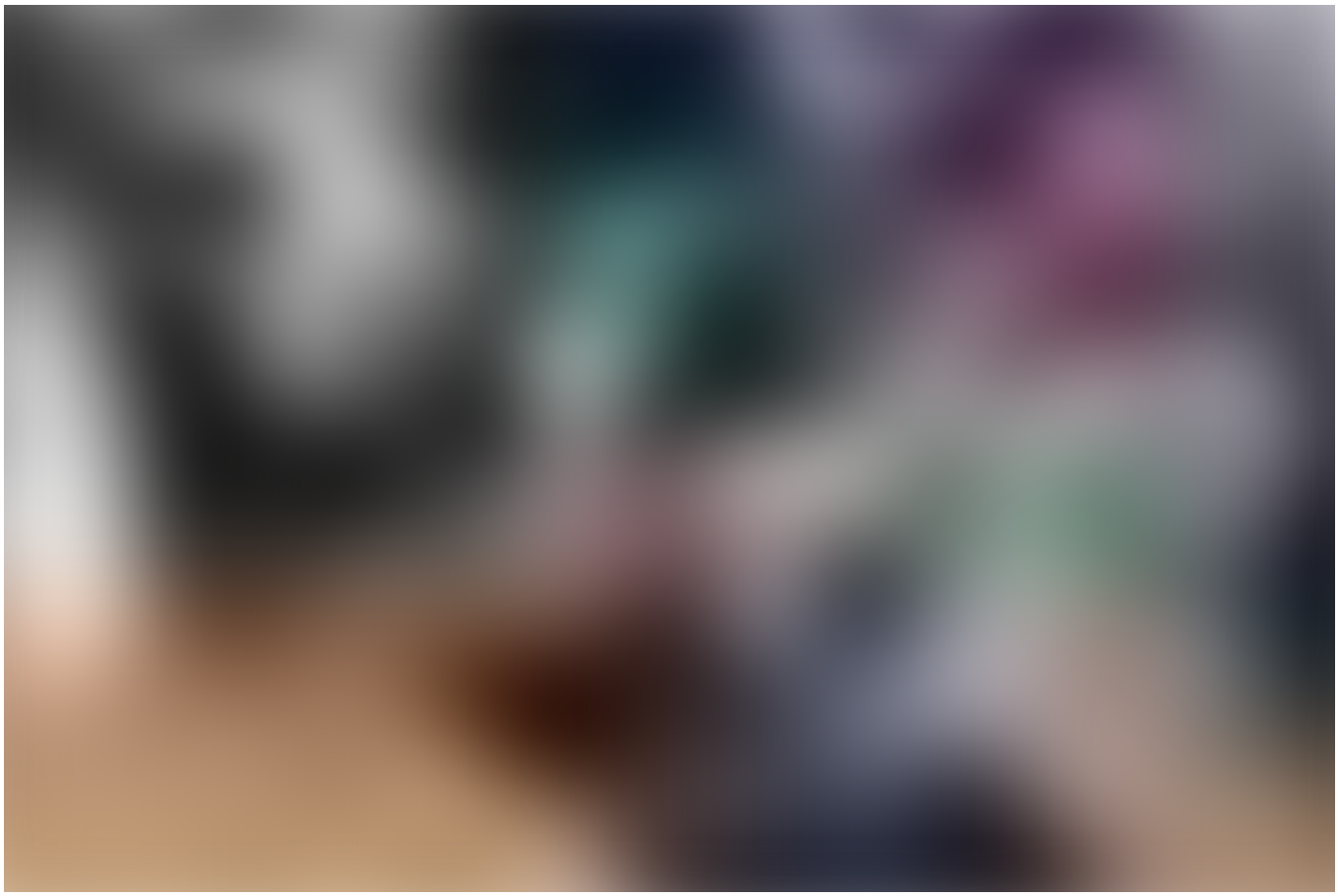
- You define a function that will take the column values you want to play with to come up with your logic. Here the only two columns we end up using are genre and rating.
- You use an apply function with lambda along the row with axis=1. The general syntax is:

```
df.apply(lambda x: func(x['col1'],x['col2']),axis=1)
```

You should be able to create pretty much any logic using apply/lambda since you just have to worry about the custom function.

. . .

Filtering a dataframe



Pandas make filtering and subsetting dataframes pretty easy. You can filter and subset dataframes using normal operators and `&,|,~` operators.

```
# Single condition: dataframe with all movies rated greater than 8

df_gt_8 = df[df['Rating']>8]

# Multiple conditions: AND - dataframe with all movies rated greater
than 8 and having more than 100000 votes

And_df = df[(df['Rating']>8) & (df['Votes']>100000)]

# Multiple conditions: OR - dataframe with all movies rated greater
than 8 or having a metascore more than 90

Or_df = df[(df['Rating']>8) | (df['Metascore']>80)]

# Multiple conditions: NOT - dataframe with all emovies rated greater
than 8 or having a metascore more than 90 have to be excluded

Not_df = df[~((df['Rating']>8) | (df['Metascore']>80))]
```

Pretty simple stuff.

But sometimes we may need to do complex filtering operations.

And sometimes we need to do some operations which we won't be able to do using just the above format.

For instance: Let us say ***we want to filter those rows where the number of words in the movie title is greater than or equal to than 4.***

How would you do it?

Trying the below will give you an error. Apparently, you cannot do anything as simple as split with a series.

```
new_df = df[len(df['Title'].split(" "))>=4]
----------------------------------------
AttributeError: 'Series' object has no attribute 'split'
```

One way is to first create a column which contains no of words in the title using apply and then filter on that column.

```
#create a new column
df['num_words_title'] = df.apply(lambda x : len(x['Title'].split("
")),axis=1)

#simple filter on new column
new_df = df[df['num_words_title']>=4]
```

And that is a perfectly fine way as long as you don't have to create a lot of columns. But, I prefer this:

```
new_df = df[df.apply(lambda x : len(x['Title'].split("
"))>=4,axis=1)]
```

What I did here is that ***my apply function returns a boolean which can be used to filter.***

Now once you understand that you just have to create a column of booleans to filter, you can use any function/logic in your `apply` statement to get however complex a logic you want to build.

Let us see another example. I will try to do something a little complex to just show the structure.

We want to find movies for which the revenue is less than the average revenue for that particular year?

```
year_revenue_dict =
df.groupby(['Year']).agg({'Rev_M':np.mean}).to_dict()['Rev_M']
```

```
def bool_provider(revenue, year):
    return revenue<year_revenue_dict[year]

new_df = df[df.apply(lambda x :
bool_provider(x['Rev_M'],x['Year']),axis=1)]
```

We have a function here which we can use to write any logic. That provides a lot of power for advanced filtering as long as we can play with simple variables.

. . .

Change Column Types

I even use apply to change the column types since I don't want to remember the syntax for changing column type and also since it lets me do much more complex things.

The normal syntax to change column type is `astype` in Pandas. So if I had a column named price in my data in an `str` format. I could do this:

```
df['Price'] = newDf['Price'].astype('int')
```

But sometimes it won't work as expected.

You might get the error: `ValueError: invalid literal for long() with base 10: '13,000'.` That is you cannot cast a string with `","` to an int. To do that we first have to get rid of the comma.

After facing this problem time and again, I have stopped using `astype` altogether now and just use apply to change column types.

```
df['Price'] = df.apply(lambda x: int(x['Price'].replace(',',
'')),axis=1)
```

. . .

And lastly there is `progress_apply`

`progress_apply` is a single function that comes with `tqdm` package.

And this has saved me a lot of time.

Sometimes when you have got a lot of rows in your data, or you end up writing a pretty complex apply function, you will see that apply might take a lot of time.

I have seen apply taking hours when working with Spacy. In such cases, you might like to see the progress bar with apply.

You can use `tqdm` for that.

After the initial imports at the top of your notebook, just replace `apply` with `progress_apply` and everything remains the same.

```
from tqdm import tqdm, tqdm_notebook
tqdm_notebook().pandas()

df.progress_apply(lambda x:
custom_rating_function(x['Genre'],x['Rating']),axis=1)
```

And you get progress bars.

. . .

Conclusion

`apply` and `lambda` functionality lets you take care of a lot of complex things while manipulating data.

I feel that I don't have to worry about a lot of stuff while using Pandas since I can use `apply` well.

In this post, I tried to explain how it works. And there might be other ways to do whatever I have done above.

But I like to stick with `apply` / `lambda` in place of `map` / `applymap` because I find it more readable and well suited to my workflow.

. . .

If you want to learn more about Python 3, I would like to call out an excellent course on Learn **Intermediate level Python** from the University of Michigan. Do check it out.

. . .

I am going to be writing more of such posts in the future too. Let me know what you think about the series. Follow me up at **Medium** or Subscribe to my **blog** to be informed about them. As always, I welcome feedback and constructive criticism and can be reached on Twitter @mlwhiz.

)

Data Science | Machine Learning | Programming | Towards Data Science | Python