Open in app

Lera Tsayukova

16 Followers About Follow

You can now subscribe to get stories delivered directly to your inbox.

Got it

Pandas Tricks:

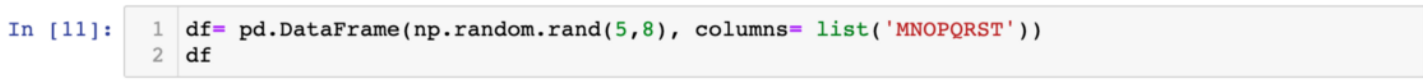
Lera Tsayukova Jun 3 · 4 min read

Pandering to the Data Science Student Masses

There are a **variety of ways to do things in pandas** — translation: This is both a *blessing* and a *curse*. The novice user of Pandas (ie. me) can easily get confused. So while there exists probably more than one way to do any of the following these are the methods I found the least intimidating.

Create a Random DataFrame (Quickly!):

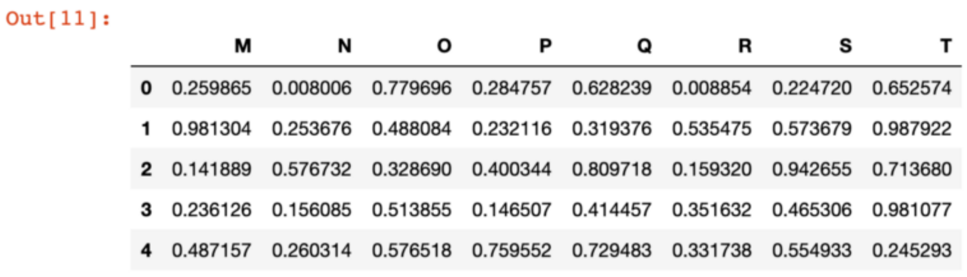
- Say you want to give an example or how to do something in Pandas, but you don't want to bother scrolling through endless folders to find an existing dataset to work with. Solution: Create one with Pandas!
- Using the — pd.random.rand() — you can quickly generate an example DataFrame:



```
In [11]:  df= pd.DataFrame(np.random.rand(5,8), columns= list('MNOPQRST'))
          df
```

Optionally pass in a string list to columns=

Pandas by default will generate a DataFrame of numbers for both the column and row names, but you can pass it a list of strings as shown up in order to make it easier to work with. The input generates:

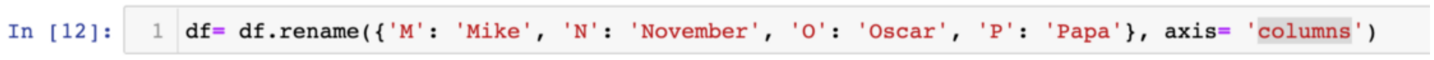


Out[11]:

	M	N	O	P	Q	R	S	T
0	0.259865	0.008006	0.779696	0.284757	0.628239	0.008854	0.224720	0.652574
1	0.981304	0.253676	0.488084	0.232116	0.319376	0.535475	0.573679	0.987922
2	0.141889	0.576732	0.328690	0.400344	0.809718	0.159320	0.942655	0.713680
3	0.236126	0.156085	0.513855	0.146507	0.414457	0.351632	0.465306	0.981077
4	0.487157	0.260314	0.576518	0.759552	0.729483	0.331738	0.554933	0.245293

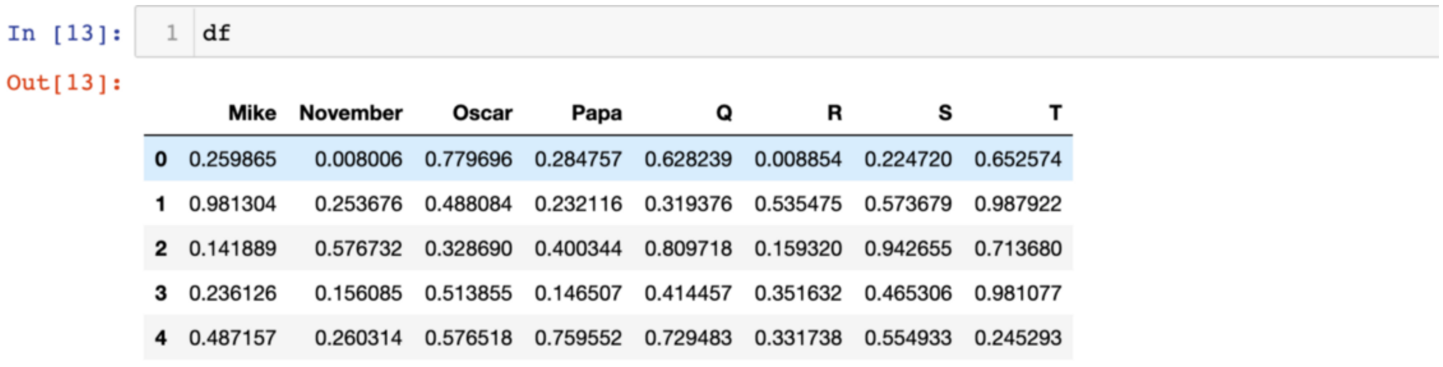
Let's say you decided you hate the column names, whether they are of your choosing or the columns of another dataset whose names make as much sense as Carol Baskin's alibi….

You can quickly rename any Pandas DataFrame columns by inputting df.rename() and passing in a dictionary. Don't forget to specify which axis you are working on!



```
In [12]:  df= df.rename({'M': 'Mike', 'N': 'November', 'O': 'Oscar', 'P': 'Papa'}, axis= 'columns')
```

And just like that we have renamed (almost) all our columns:

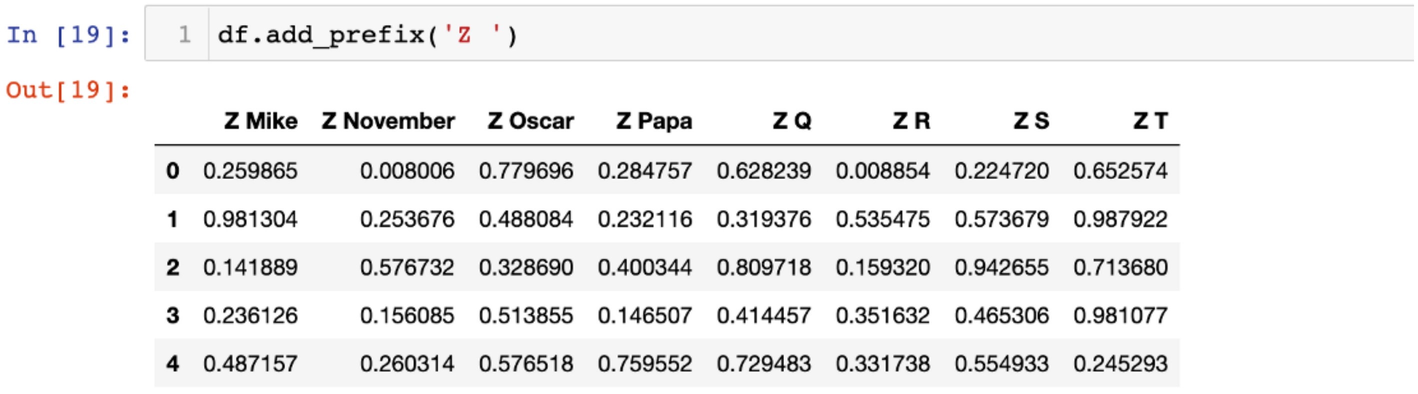


```
In [13]: df
Out[13]:
```

	Mike	November	Oscar	Papa	Q	R	S	T
0	0.259865	0.008006	0.779696	0.284757	0.628239	0.008854	0.224720	0.652574
1	0.981304	0.253676	0.488084	0.232116	0.319376	0.535475	0.573679	0.987922
2	0.141889	0.576732	0.328690	0.400344	0.809718	0.159320	0.942655	0.713680
3	0.236126	0.156085	0.513855	0.146507	0.414457	0.351632	0.465306	0.981077
4	0.487157	0.260314	0.576518	0.759552	0.729483	0.331738	0.554933	0.245293

Okay fine, after we renamed HALF our columns:

We can also add a prefix, or get rid of a prefix if need be:



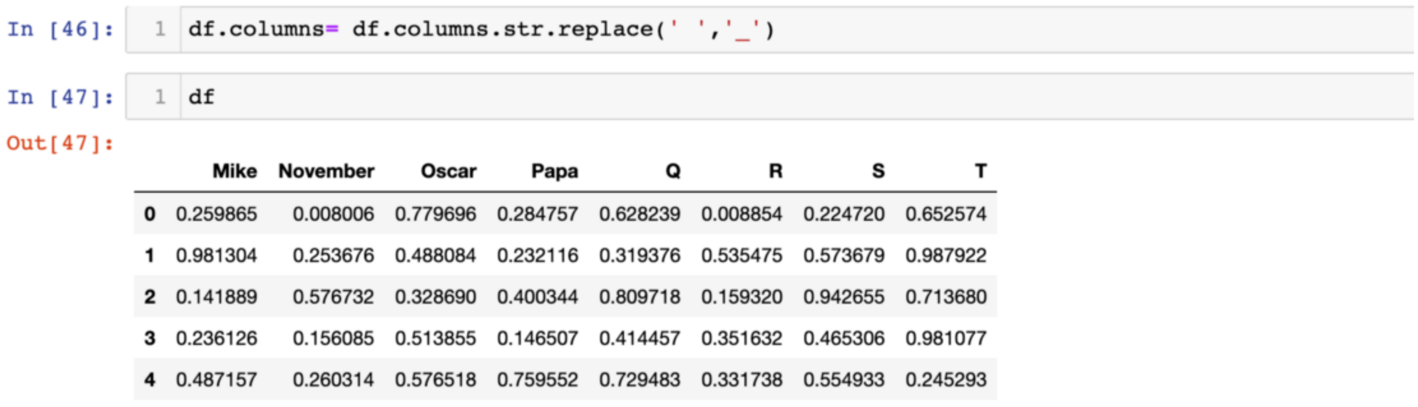
```
In [19]: df.add_prefix('Z ')
Out[19]:
```

	Z Mike	Z November	Z Oscar	Z Papa	Z Q	Z R	Z S	Z T
0	0.259865	0.008006	0.779696	0.284757	0.628239	0.008854	0.224720	0.652574
1	0.981304	0.253676	0.488084	0.232116	0.319376	0.535475	0.573679	0.987922
2	0.141889	0.576732	0.328690	0.400344	0.809718	0.159320	0.942655	0.713680
3	0.236126	0.156085	0.513855	0.146507	0.414457	0.351632	0.465306	0.981077
4	0.487157	0.260314	0.576518	0.759552	0.729483	0.331738	0.554933	0.245293

Adding prefix to all of our columns

Ahhh this is terrible. CHANGE IT BACK!!

Not to worry: this mistake can easily be remedied by using df.columns.str.replace() method as shown below:



```
In [46]: df.columns= df.columns.str.replace(' ','_')
In [47]: df
Out[47]:
```

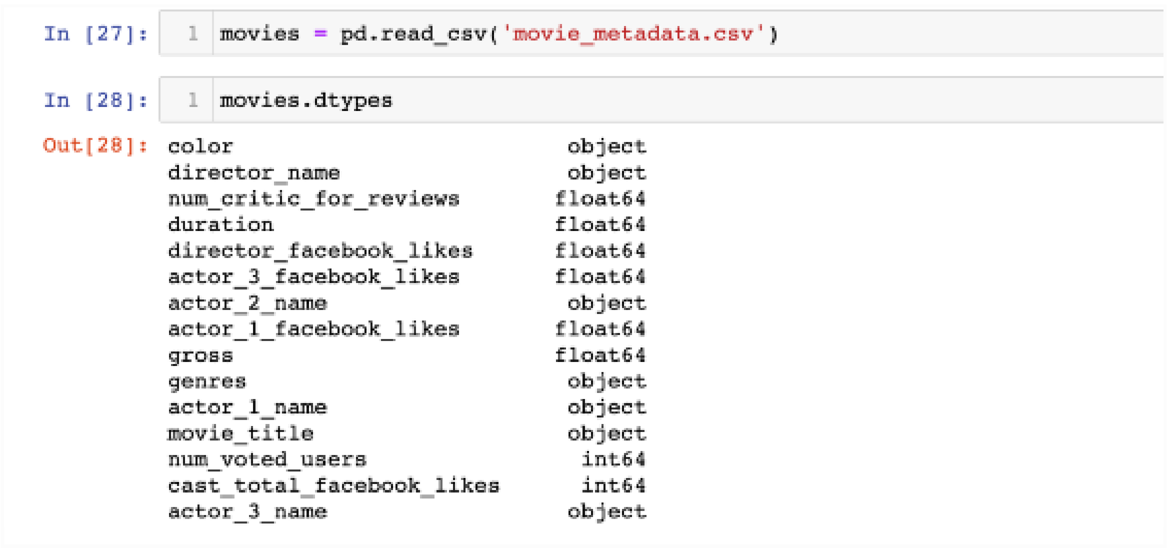
	Mike	November	Oscar	Papa	Q	R	S	T
0	0.259865	0.008006	0.779696	0.284757	0.628239	0.008854	0.224720	0.652574
1	0.981304	0.253676	0.488084	0.232116	0.319376	0.535475	0.573679	0.987922
2	0.141889	0.576732	0.328690	0.400344	0.809718	0.159320	0.942655	0.713680
3	0.236126	0.156085	0.513855	0.146507	0.414457	0.351632	0.465306	0.981077
4	0.487157	0.260314	0.576518	0.759552	0.729483	0.331738	0.554933	0.245293

Using the df.columns.str.replace() feature

2). D-Types:

Next, we examine D-Types and how to interact with them in order to get more out of our data exploration. For this we will be using our movies dataset:

In any dataset, we usually have a mixture of datatypes or D-types which sometimes makes it increasingly frustrating to work with and infuriating to discover after you have already error-red out. Believe me I share your rage…As such here are some quick and dirty examples of how to filter out d-types:

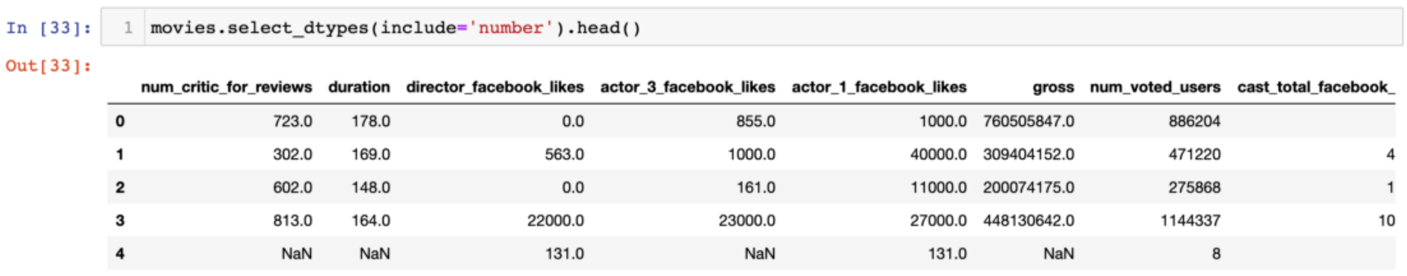


```
In [27]:  movies = pd.read_csv('movie_metadata.csv')

In [28]:  movies.dtypes
Out[28]: color                        object
         director_name                object
         num_critic_for_reviews      float64
         duration                    float64
         director_facebook_likes     float64
         actor_3_facebook_likes      float64
         actor_2_name                 object
         actor_1_facebook_likes      float64
         gross                       float64
         genres                       object
         actor_1_name                 object
         movie_title                  object
         num_voted_users               int64
         cast_total_facebook_likes     int64
         actor_3_name                 object
```

Here in our movie dataset we can see a variety of objects, integers, and floats.

Let’s say you ONLY wanted to look at the numerical values in this dataset. To do this simply employ the following: df.select_dtypes(include=) as shown below:

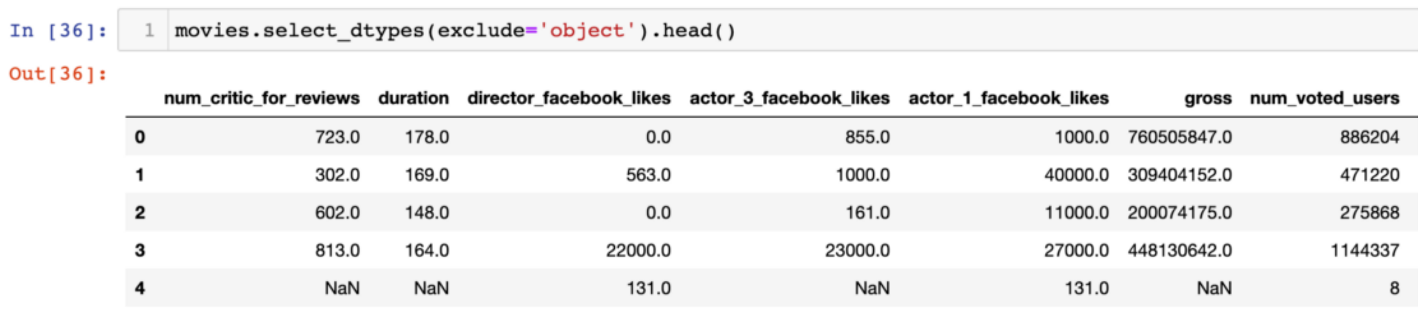


```
In [33]:  movies.select_dtypes(include='number').head()
Out[33]:
```

	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_1_facebook_likes	gross	num_voted_users	cast_total_facebook_
0	723.0	178.0	0.0	855.0	1000.0	760505847.0	886204	
1	302.0	169.0	563.0	1000.0	40000.0	309404152.0	471220	4
2	602.0	148.0	0.0	161.0	11000.0	200074175.0	275868	1
3	813.0	164.0	22000.0	23000.0	27000.0	448130642.0	1144337	10
4	NaN	NaN	131.0	NaN	131.0	NaN	8	

Using df.select_dytpes(include="") feature

Luckily, this feature also works in reverse! Let's use the **'exclude'** feature to **only include numbers:**



```
In [36]: movies.select_dtypes(exclude='object').head()
Out[36]:
```

	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_1_facebook_likes	gross	num_voted_users
0	723.0	178.0	0.0	855.0	1000.0	760505847.0	886204
1	302.0	169.0	563.0	1000.0	40000.0	309404152.0	471220
2	602.0	148.0	0.0	161.0	11000.0	200074175.0	275868
3	813.0	164.0	22000.0	23000.0	27000.0	448130642.0	1144337
4	NaN	NaN	131.0	NaN	131.0	NaN	8

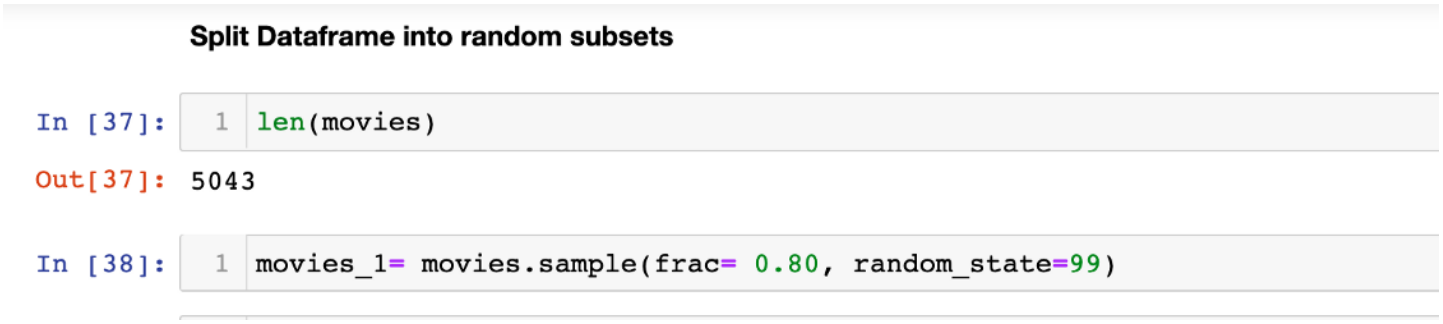
Using select_dtypes(exclude=) feature

3) Subsetting DataFrames

Lastly, lets look at Pandas DataFrames themselves, namely how to create subsets of a DataFrame.

If for any reason you want to split up your DataFrame, Pandas has an answer to that. It will allow you to create two subsets of data according to the parameters you set, whether that be 75:25 or 60:40 you get the idea.

Let's go back to our movie dataset. It has 5043 rows. For whatever reason we don't need this many and have decided that we would like ehh roughly 4000 movies instead. In order to subset our 'movies' DataFrame we call the following function: df.sample(frac= , random_state=). In practice it looks much like this:



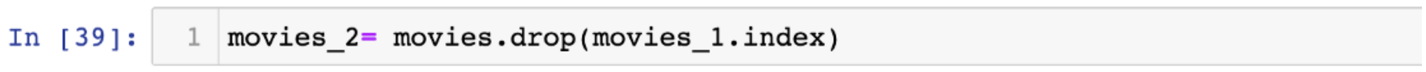
Split Dataframe into random subsets

```
In [37]: len(movies)
Out[37]: 5043

In [38]: movies_1= movies.sample(frac= 0.80, random_state=99)
```

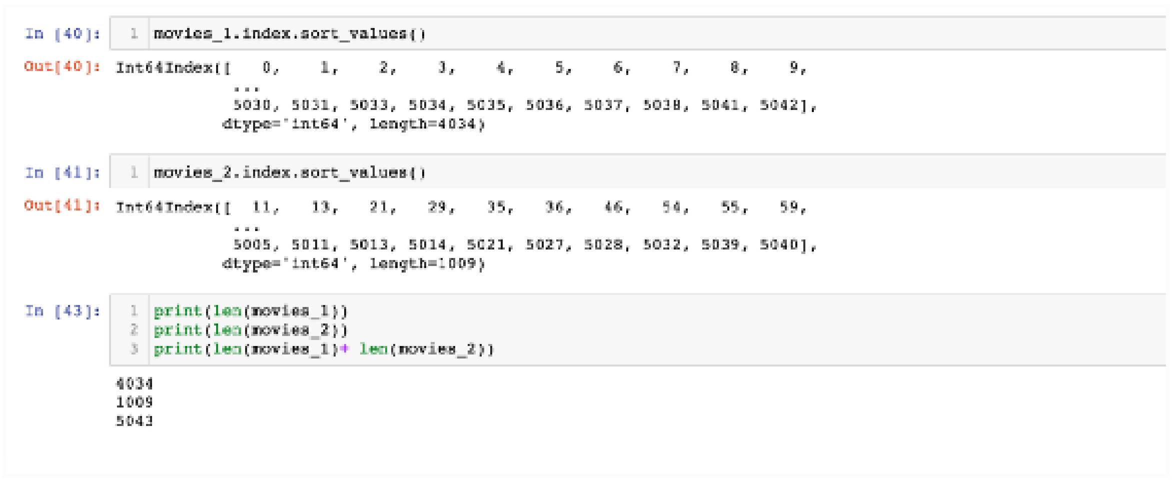
How to subset DataFrame using df.sample()

The 'random_state' simply splits up the data set randomly which the 'frac' takes the percentage of the data set you want to subset. Now we can create a second subset with the remaining 20% of our movies data.



```
In [39]:  movies_2= movies.drop(movies_1.index)
```

Creating a second subset by dropping our first subset from our original DataFrame using df.drop()



```
In [40]:  movies_1.index.sort_values()
Out[40]: Int64Index([   0,    1,    2,    3,    4,    5,    6,    7,    8,    9,
             ...
             5030, 5031, 5033, 5034, 5035, 5036, 5037, 5038, 5041, 5042],
            dtype='int64', length=4034)

In [41]:  movies_2.index.sort_values()
Out[41]: Int64Index([  11,   13,   21,   29,   35,   36,   46,   54,   55,   59,
             ...
             5005, 5011, 5013, 5014, 5021, 5027, 5028, 5032, 5039, 5040],
            dtype='int64', length=1009)

In [43]:  print(len(movies_1))
          print(len(movies_2))
          print(len(movies_1)+ len(movies_2))

         4034
         1009
         5043
```

Here we can see the index numbers of our two movie subsets. And confirm that their concatenation will add up back to the original DataFrame.

Note: This method will only work if you have unique id's for your index.

Hopefully, the above methods have been insightful or at the very least offer some help to the struggling data science students out there. I feel your pain.

Data Science Pandas Dataframe Jupyter Notebook