03-03: Using Function GroupBy( )

Sometimes we want to select data based on groups and understand aggregated data on a group level. We have seen that even though Pandas allows us to iterate over every row in a dataframe, it is generally very slow to do so. Fortunately Pandas has a `groupby()` function to speed up such task. The idea behind the `groupby()` function is that it takes some dataframe, splits it into chunks based on some key values, applies computation on those chunks, then combines the results back together into another dataframe. In pandas this is refered to as the ***split-apply-combine*** pattern.

Splitting using `GroupBy()`

In [1]:

```
# Let's look at an example. First, we'll bring in our pandas and numpy libraries
import pandas as pd
import numpy as np
```

In [2]:

```
# Let's look at some US census data
df = pd.read_csv('datasets/census.csv')
# And exclude state level summarizations, which have sum level value of 40
df = df[df['SUMLEV']==50]
df.head()
```

Out[2]:

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	E
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	50	3	6	1	9	Alabama	Blount County	57322	

5 rows × 100 columns

Getting Average Population: Time Analysis Without Using `GroupBy()`

Let's get a list of the unique states, then we can iterate over all the states and for each state we reduce the data frame and calculate the average. Let's run such task for 3 times and time it.

For this we'll use the cell magic function `%%timeit`

In [3]:

```
%%timeit -n 3

for state in df['STNAME'].unique():
    # We'll just calculate the average using numpy for this particular state
    avg = np.average(df.where(df['STNAME']==state).dropna()['CENSUS2010POP'])
    # And we'll print it to the screen
    #print('Counties in state ' + state +
          #' have an average population of ' + str(avg))
```

```
2.18 s ± 12.9 ms per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

If you scroll down to the bottom of that output you can see it takes a fair bit of time to finish.

Getting Average Population: Time Analysis Using `GroupBy()`

Return Values - groupby() returns a tuple, where the **first** value is the value of the key we were trying to group by (ie. state name), and the **second** value is the *projected* ***dataframe*** that was found for that group.

The groupby() step is the **"SPLIT"** in the split-apply-combine pattern, so this splits data into the groups that we aim to split the data by.

In [4]:

```
%%timeit -n 3
# For this method, we start by telling pandas we're interested in grouping by st
ate name, this is the "split"
for group, frame in df.groupby('STNAME'):

    # Now we include our logic in the "apply" step, which is to calculate an ave
rage of the census2010pop
    avg = np.average(frame['CENSUS2010POP'])
    # And print the results
    #print('Counties in state ' + group +
         # ' have an average population of ' + str(avg))
# And we don't have to worry about the combine step in this case, because all of
our data transformation is
# actually printing out results.
```

```
14.5 ms ± 369 µs per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

In [5]:

```
# Wow, what a huge difference in speed. An improve by roughly by two factors!
```

Creating Groups with `GroupBy(function_reference)` using Multiple Columns

Example 1: Batch Number

In [6]:

```
# Now, 99% of the time, you'll use group by on one or more columns. But you can
 also provide a function to
# group by and use that to segment your data.

# This is a bit of a fabricated example but lets say that you have a big batch j
ob with lots of processing and
# you want to work on only a third or so of the states at a given time. We could
create some function which
# returns a number between zero and two based on the first character of the stat
e name. Then we can tell group
# by to use this function to split up our data frame. It's important to note tha
t in order to do this you need
# to set the index of the data frame to be the column that you want to group by
 first.

# We'll create some new function called set_batch_number and if the first letter
of the parameter is a capital
# M we'll return a 0. If it's a capital Q we'll return a 1 and otherwise we'll r
eturn a 2. Then we'll pass
# this function to the data frame

df = df.set_index('STNAME')

def set_batch_number(item):
    if item[0]<'M':
        return 0
    if item[0]<'Q':
        return 1
    return 2

# The dataframe is supposed to be grouped by according to the batch number And w
e will loop through each batch
# group
for group, frame in df.groupby(set_batch_number):
    print('There are ' + str(len(frame)) + ' records in group ' + str(group) + '
for processing.')
```

```
There are 1177 records in group 0 for processing.
There are 1134 records in group 1 for processing.
There are 831 records in group 2 for processing.
```

Notice that this time I didn't pass in a column name to groupby(). Instead, I set the index of the dataframe to be STNAME, and **if no column identifier is passed groupby() will automatically use the index**.

Example 2: AirBNB Listings

In [7]:

```
# Let's take one more look at an example of how we might group data. In this exa
mple, I want to use a dataset
# of housing from airbnb. In this dataset there are two columns of interest, one
is the cancellation_policy
# and the other is the review_scores_value.
df=pd.read_csv("datasets/listings.csv")
df.head()
```

Out[7]:

	id	listing_url	scrape_id	last_scraped	name
0	12147973	https://www.airbnb.com/rooms/12147973	20160906204935	2016-09-07	Sunny Bungalow in the City
1	3075044	https://www.airbnb.com/rooms/3075044	20160906204935	2016-09-07	Charming room in pet friendly apt
2	6976	https://www.airbnb.com/rooms/6976	20160906204935	2016-09-07	Mexican Folk Art Haven in Boston
3	1436513	https://www.airbnb.com/rooms/1436513	20160906204935	2016-09-07	Spacious Sunny Bedroom Suite in Historic Home
4	7651065	https://www.airbnb.com/rooms/7651065	20160906204935	2016-09-07	Come Home to Boston

5 rows × 95 columns

Using `GroupBy()` using Multiple Columns

Promote them to a multiindexand just call groupby(). When we have a multiindex we need to pass in the levels we are interested in grouping by

In [8]:

```
# So, how would I group by both of these columns? A first approach might be to p
romote them to a multiindex
# and just call groupby()
df=df.set_index(["cancellation_policy","review_scores_value"])

# When we have a multiindex we need to pass in the levels we are interested in g
rouping by
for group, frame in df.groupby(level=(0,1)):
    print(group)
```

```
('flexible', 2.0)
('flexible', 4.0)
('flexible', 5.0)
('flexible', 6.0)
('flexible', 7.0)
('flexible', 8.0)
('flexible', 9.0)
('flexible', 10.0)
('moderate', 2.0)
('moderate', 4.0)
('moderate', 6.0)
('moderate', 7.0)
('moderate', 8.0)
('moderate', 9.0)
('moderate', 10.0)
('strict', 2.0)
('strict', 3.0)
('strict', 4.0)
('strict', 5.0)
('strict', 6.0)
('strict', 7.0)
('strict', 8.0)
('strict', 9.0)
('strict', 10.0)
('super_strict_30', 6.0)
('super_strict_30', 7.0)
('super_strict_30', 8.0)
('super_strict_30', 9.0)
('super_strict_30', 10.0)
```

In [9]:

```
# This seems to work ok. But what if we wanted to group by the cancelation polic
y and review scores, but
# separate out all the 10's from those under ten? In this case, we could use a f
unction to manage the
# groupings

def grouping_fun(item):
    # Check the "review_scores_value" portion of the index. item is in the forma
t of
    # (cancellation_policy,review_scores_value
    if item[1] == 10.0:
        return (item[0],"10.0")
    else:
        return (item[0],"not 10.0")

for group, frame in df.groupby(by=grouping_fun):
    print(group)
```

```
('flexible', '10.0')
('flexible', 'not 10.0')
('moderate', '10.0')
('moderate', 'not 10.0')
('strict', '10.0')
('strict', 'not 10.0')
('super_strict_30', '10.0')
('super_strict_30', 'not 10.0')
```

In [10]:

```
df.head()
```

Out[10]:

		id	listing_url	scra
cancellation_policy	**review_scores_value**			
moderate	**NaN**	12147973	https://www.airbnb.com/rooms/12147973	201
	9.0	3075044	https://www.airbnb.com/rooms/3075044	201
	10.0	6976	https://www.airbnb.com/rooms/6976	201
	10.0	1436513	https://www.airbnb.com/rooms/1436513	201
flexible	**10.0**	7651065	https://www.airbnb.com/rooms/7651065	201

5 rows × 93 columns

Applying

To this point we have applied very simple processing to our data after splitting, really just outputting some print statements to demonstrate how the splitting works. The pandas developers have three broad categories of data processing to happen during the apply step, **Aggregation** of group data, **Transformatio** of group data, and **Filtration** of group data

Applying: Aggregation

The most straight forward apply step is the aggregation of data, and uses the method agg() on the groupby object. Thus far we have only iterated through the groupby object, unpacking it into a label (the group name) and a dataframe. But **with agg we can pass in a dictionary of the columns** we are interested in aggregating along with **the reference** of the function we are looking to apply to aggregate

In [11]:

```
# Let's reset the index for our airbnb data
df=df.reset_index()

# Now lets group by the cancellation policy and find the average review_scores_v
alue by group
#Note that np.average() is a function, np.average is a function reference that i
s passed into agg.
df.groupby("cancellation_policy").agg({"review_scores_value":np.average})
```

Out[11]:

	review_scores_value
cancellation_policy	
flexible	NaN
moderate	NaN
strict	NaN
super_strict_30	NaN

Using `np.nanmean()`

In [12]:

```
# Hrm. That didn't seem to work at all. Just a bunch of not a numbers. The issue
is actually in the function
# that we sent to aggregate. np.average does not ignore nans! However, there is
 another function we can use for this
df.groupby("cancellation_policy").agg({"review_scores_value":np.nanmean})
```

Out[12]:

	review_scores_value
cancellation_policy	
flexible	9.237421
moderate	9.307398
strict	9.081441
super_strict_30	8.537313

In [13]:

```
# We can just extend this dictionary to aggregate by multiple functions or multi
ple columns.
df.groupby("cancellation_policy").agg({"review_scores_value":(np.nanmean,np.nans
td),
                                       "reviews_per_month":np.nanmean})
```

Out[13]:

	review_scores_value		reviews_per_month
	nanmean	nanstd	nanmean
cancellation_policy			
flexible	9.237421	1.096271	1.829210
moderate	9.307398	0.859859	2.391922
strict	9.081441	1.040531	1.873467
super_strict_30	8.537313	0.840785	0.340143

Take a moment to make sure you understand the previous cell, since it's somewhat complex.

1. First we're doing a group by on the dataframe object by the column "cancellation_policy". This creates a new GroupBy object.
2. Then we are invoking the agg() function on that object. The agg function is going to apply one or more functions we specify to the group dataframes and return a single row per dataframe/group.
3. When we called this function we sent it two dictionary entries, each with the key indicating which column we wanted functions applied to.

    - For the first column we actually supplied a tuple of two functions. *Note that these are not function invocations, like np.nanmean(), or function names, like "nanmean" they are references to functions which will return single values.* The groupby object will recognize the tuple and call each
    - For the second column, we indicated another column and another reference to a function that we wanted to run on that column.

1. The groupby object will recognize the tuple on the **1st dict entry** and call the functions **in order**. The result will be in a **hierarchical index**, where the primary index is the mean and the secondary is the standard deviation.

Applying: Transformation

Transformation is different from aggregation. Where agg() returns a single value per column, so one row per group, tranform() returns an object that is the same size as the group. Essentially, it **broadcasts the function you supply over the grouped dataframe, returning a new dataframe.** This makes combining data later easy.

In [14]:

```
# For instance, suppose we want to include the average rating values in a given
 group by cancellation policy,
# but preserve the dataframe shape so that we could generate a difference betwee
n an individual observation
# and the sum.

# First, lets define just some subset of columns we are interested in
cols=['cancellation_policy','review_scores_value']
# Now lets transform it, I'll store this in its own dataframe
transform_df=df[cols].groupby('cancellation_policy').transform(np.nanmean)
transform_df.head()
```

Out[14]:

	review_scores_value
0	9.307398
1	9.307398
2	9.307398
3	9.307398
4	9.237421

Applying Transformed Data To DataFrame

In [15]:

```
# So we can see that the index here is actually the same as the original datafra
me. So lets just join this
# in. Before we do that, lets rename the column in the transformed version
transform_df.rename({'review_scores_value':'mean_review_scores'}, axis='columns'
, inplace=True)
df=df.merge(transform_df, left_index=True, right_index=True)
df.head()
```

Out[15]:

	cancellation_policy	review_scores_value	id	listing_url	
0	moderate	NaN	12147973	https://www.airbnb.com/rooms/12147973	2
1	moderate	9.0	3075044	https://www.airbnb.com/rooms/3075044	2
2	moderate	10.0	6976	https://www.airbnb.com/rooms/6976	2
3	moderate	10.0	1436513	https://www.airbnb.com/rooms/1436513	2
4	flexible	10.0	7651065	https://www.airbnb.com/rooms/7651065	2

5 rows × 96 columns

In [16]:

```
# Great, we can see that our new column is in place, the mean_review_scores. So
 now we could create, for
# instance, the difference between a given row and it's group (the cancellation
 policy) means.
df['mean_diff']=np.absolute(df['review_scores_value']-df['mean_review_scores'])
#vectorisation
df['mean_diff'].head()
```

Out[16]:

```
0         NaN
1    0.307398
2    0.692602
3    0.692602
4    0.762579
Name: mean_diff, dtype: float64
```

Filtering

The GroupBy object has build in support for filtering groups as well. It's often that you'll want to group by some feature, then make some transformation to the groups, then **drop certain groups as part of your cleaning routines**. The filter() function takes in a function which it applies to each group dataframe and returns either a True or a False, depending upon whether that group should be included in the results.

In [17]:

```
# For instance, if we only want those groups which have a mean rating above 9 in
cluded in our results
df.groupby('cancellation_policy').filter(lambda x: np.nanmean(x['review_scores_v
alue'])>9.2)
```

Out[17]:

	cancellation_policy	review_scores_value	id	listing_ur
0	moderate	NaN	12147973	https://www.airbnb.com/rooms/1214797
1	moderate	9.0	3075044	https://www.airbnb.com/rooms/307504
2	moderate	10.0	6976	https://www.airbnb.com/rooms/697
3	moderate	10.0	1436513	https://www.airbnb.com/rooms/143651
4	flexible	10.0	7651065	https://www.airbnb.com/rooms/765106
5	flexible	10.0	12386020	https://www.airbnb.com/rooms/1238602
7	moderate	10.0	2843445	https://www.airbnb.com/rooms/284344
8	moderate	10.0	753446	https://www.airbnb.com/rooms/75344
10	flexible	10.0	12023024	https://www.airbnb.com/rooms/1202302
11	flexible	9.0	1668313	https://www.airbnb.com/rooms/166831
12	flexible	10.0	2684840	https://www.airbnb.com/rooms/268484
13	moderate	10.0	13547301	https://www.airbnb.com/rooms/1354730

	cancellation_policy	review_scores_value	id	listing_ur
14	moderate	9.0	5434353	https://www.airbnb.com/rooms/543435
16	flexible	10.0	3420384	https://www.airbnb.com/rooms/342038
17	flexible	10.0	13512930	https://www.airbnb.com/rooms/1351293
20	flexible	10.0	2583074	https://www.airbnb.com/rooms/258307
21	moderate	10.0	13251243	https://www.airbnb.com/rooms/1325124
23	flexible	10.0	6400432	https://www.airbnb.com/rooms/640043
24	moderate	10.0	5498472	https://www.airbnb.com/rooms/549847
25	moderate	10.0	894539	https://www.airbnb.com/rooms/89453
27	moderate	10.0	9218312	https://www.airbnb.com/rooms/921831
28	flexible	9.0	321328	https://www.airbnb.com/rooms/32132
29	flexible	10.0	1810172	https://www.airbnb.com/rooms/181017
30	flexible	9.0	6513924	https://www.airbnb.com/rooms/651392
32	flexible	NaN	14690527	https://www.airbnb.com/rooms/1469052

	cancellation_policy	review_scores_value	id	listing_ur
34	flexible	10.0	1861070	https://www.airbnb.com/rooms/186107(
39	moderate	10.0	4085362	https://www.airbnb.com/rooms/408536(
40	flexible	9.0	3755609	https://www.airbnb.com/rooms/375560(
41	flexible	10.0	13768853	https://www.airbnb.com/rooms/1376885(
42	flexible	10.0	1936861	https://www.airbnb.com/rooms/193686(
...	...	...	...	..
3533	flexible	8.0	6373729	https://www.airbnb.com/rooms/637372(
3534	flexible	10.0	8921130	https://www.airbnb.com/rooms/892113(
3535	flexible	NaN	10436811	https://www.airbnb.com/rooms/1043681(
3536	moderate	NaN	14026059	https://www.airbnb.com/rooms/1402605(
3539	moderate	NaN	4332035	https://www.airbnb.com/rooms/433203(
3540	moderate	NaN	9807292	https://www.airbnb.com/rooms/980729(
3543	flexible	NaN	14866400	https://www.airbnb.com/rooms/1486640(

	cancellation_policy	review_scores_value	id	listing_ur
3545	flexible	10.0	14400847	https://www.airbnb.com/rooms/1440084
3548	moderate	10.0	14774426	https://www.airbnb.com/rooms/1477442
3549	flexible	NaN	14487346	https://www.airbnb.com/rooms/1448734
3550	flexible	NaN	14550228	https://www.airbnb.com/rooms/1455022
3551	moderate	6.0	14566657	https://www.airbnb.com/rooms/1456665
3552	flexible	NaN	14773792	https://www.airbnb.com/rooms/1477379
3554	flexible	10.0	14533848	https://www.airbnb.com/rooms/1453384
3555	moderate	9.0	13015653	https://www.airbnb.com/rooms/1301565
3557	flexible	NaN	14604429	https://www.airbnb.com/rooms/1460442
3559	moderate	10.0	12915510	https://www.airbnb.com/rooms/1291551
3562	flexible	NaN	14881840	https://www.airbnb.com/rooms/1488184
3563	moderate	NaN	11452297	https://www.airbnb.com/rooms/1145229
3565	flexible	10.0	14335003	https://www.airbnb.com/rooms/1433500

	cancellation_policy	review_scores_value	id	listing_ur
3566	flexible	NaN	14871445	https://www.airbnb.com/rooms/1487144
3569	flexible	NaN	14426586	https://www.airbnb.com/rooms/1442658
3571	flexible	10.0	14592547	https://www.airbnb.com/rooms/1459254
3573	flexible	10.0	14504583	https://www.airbnb.com/rooms/1450458
3574	moderate	8.0	14743129	https://www.airbnb.com/rooms/1474312
3576	flexible	NaN	14689681	https://www.airbnb.com/rooms/1468968
3577	flexible	NaN	13750763	https://www.airbnb.com/rooms/1375076
3579	flexible	NaN	14852179	https://www.airbnb.com/rooms/1485217
3582	flexible	NaN	14585486	https://www.airbnb.com/rooms/1458548
3584	flexible	NaN	14504422	https://www.airbnb.com/rooms/1450442

1918 rows × 97 columns

Notice that the results are still indexed, but that any of the results which were in a group with a mean review score of less than or equal to 9.2 were not copied over.

Using Function `apply()`

Using apply can be slower than using some of the specialized functions, especially agg(). But, if your dataframes are not huge, it's a solid general purpose approach.

Groupby is a powerful and commonly used tool for data cleaning and data analysis. Once you have grouped the data by some category you have a dataframe of just those values and you can conduct aggregated analsyis on the segments that you are interested. The groupby() function follows a split-apply-combine approach - first the data is split into subgroups, then you can apply some transformation, filtering, or aggregation, then the results are combined automatically by pandas for us.

In [18]:

```
# By far the most common operation I invoke on groupby objects is the apply() fu
nction. This allows you to
# apply an arbitrary function to each group, and stitch the results back for eac
h apply() into a single
# dataframe where the index is preserved.

# Lets look at an example using our airbnb data, I'm going to get a clean copy o
f the dataframe
df=pd.read_csv("datasets/listings.csv")
# And lets just include some of the columns we were interested in previously
df=df[['cancellation_policy','review_scores_value']]
df.head()
```

Out[18]:

	cancellation_policy	review_scores_value
0	moderate	NaN
1	moderate	9.0
2	moderate	10.0
3	moderate	10.0
4	flexible	10.0

In [19]:

```
# In previous work we wanted to find the average review score of a listing and i
ts deviation from the group
# mean. This was a two step process, first we used transform() on the groupby ob
ject and then we had to
# broadcast to create a new column. With apply() we could wrap this logic in one
place
def calc_mean_review_scores(group):
    # group is a dataframe just of whatever we have grouped by, e.g. cancellatio
n policy, so we can treat
    # this as the complete dataframe
    avg=np.nanmean(group["review_scores_value"])
    # now broadcast our formula and create a new column
    group["review_scores_mean"]=np.abs(avg-group["review_scores_value"])
    return group

# Now just apply this to the groups
df.groupby('cancellation_policy').apply(calc_mean_review_scores).head()
```

Out[19]:

	cancellation_policy	review_scores_value	review_scores_mean
0	moderate	NaN	NaN
1	moderate	9.0	0.307398
2	moderate	10.0	0.692602
3	moderate	10.0	0.692602
4	flexible	10.0	0.762579