Grouping and Sorting

Scale up your level of insight. The more complex the dataset, the more this matters

Tutorial Data

Learn Tutorial
Pandas

Course step
4 of 6

Introduction

Maps allow us to transform data in a DataFrame or Series one value at a time for an entire column. However, often we want to group our data, and then do something specific to the group the data is in.

As you'll learn, we do this with the `groupby()` operation. We'll also cover some additional topics, such as more complex ways to index your DataFrames, along with how to sort your data.

Groupwise analysis ¶

One function we've been using heavily thus far is the `value_counts()` function. We can replicate what `value_counts()` does by doing the following:

Hide code

In [1]:
```
import pandas as pd
reviews = pd.read_csv("../input/wine-reviews/winemag-data-130k-v2.csv", index_col=0)
pd.set_option("display.max_rows", 5)
```

In [2]:
```
reviews.groupby('points').points.count()
```

Out[2]:
```
points
80     397
81     692
      ...
99      33
100     19
Name: points, Length: 21, dtype: int64
```

`groupby()` created a group of reviews which allotted the same point values to the given wines. Then, for each of these groups, we grabbed the `points()` column and counted how many times it appeared. `value_counts()` is just a shortcut to this `groupby()` operation.

We can use any of the summary functions we've used before with this data. For example, to get the cheapest wine in each point value category, we can do the following:

In [3]:
```
reviews.groupby('points').price.min()
```

Out[3]:
```
points
80      5.0
81      5.0
       ...
99     44.0
100    80.0
Name: price, Length: 21, dtype: float64
```

You can think of each group we generate as being a slice of our DataFrame containing only data with values that match. This DataFrame is accessible to us directly using the `apply()` method, and we can then manipulate the data in any way we see fit. For example, here's one way of selecting the name of the first wine reviewed from each winery in the dataset:

In [4]:
```
reviews.groupby('winery').apply(lambda df: df.title.iloc[0])
```

Out[4]:
```
winery
1+1=3                          1+1=3 NV Rosé Sparkling (Cava)
10 Knots                 10 Knots 2010 Viognier (Paso Robles)
                                       ...
àMaurice     àMaurice 2013 Fred Estate Syrah (Walla Walla V...
Štoka                          Štoka 2009 Izbrani Teran (Kras)
Length: 16757, dtype: object
```

For even more fine-grained control, you can also group by more than one column. For an example, here's how we would pick out the best wine by country *and* province:

In [5]:
```
reviews.groupby(['country', 'province']).apply(lambda df: df.loc[df.points.idxmax()])
```

		country	description	designation	points	price	province	region_1	region_2	taster_name	taster_twitter_handle
country	province										
Argentina	Mendoza Province	Argentina	If the color doesn't tell the full story, the ...	Nicasia Vineyard	97	120.0	Mendoza Province	Mendoza	NaN	Michael Schachner	@wineschach
	Other	Argentina	Take note, this could be the best wine Colomé ...	Reserva	95	90.0	Other	Salta	NaN	Michael Schachner	@wineschach
...	...	...	...	...	...	...	...	...	...	...	...
Uruguay	San Jose	Uruguay	Baked, sweet, heavy aromas turn earthy with ti...	El Preciado Gran Reserva	87	50.0	San Jose	NaN	NaN	Michael Schachner	@wineschach
	Uruguay	Uruguay	Cherry and berry aromas are ripe, healthy and ...	Blend 002 Limited Edition	91	22.0	Uruguay	NaN	NaN	Michael Schachner	@wineschach

Another `groupby()` method worth mentioning is `agg()`, which lets you run a bunch of different functions on your DataFrame simultaneously. For example, we can generate a simple statistical summary of the dataset as follows:

In [6]:
```
reviews.groupby(['country']).price.agg([len, min, max])
```

Out[6]:

	len	min	max
country			
Argentina	3800	4.0	230.0
Armenia	2	14.0	15.0
...	...	...	...
Ukraine	14	6.0	13.0
Uruguay	109	10.0	130.0

43 rows × 3 columns

Effective use of `groupby()` will allow you to do lots of really powerful things with your dataset.

Multi-indexes

In all of the examples we've seen thus far we've been working with DataFrame or Series objects with a single-label index. `groupby()` is slightly different in the fact that, depending on the operation we run, it will sometimes result in what is called a multi-index.

A multi-index differs from a regular index in that it has multiple levels. For example:

In [7]:
```
countries_reviewed = reviews.groupby(['country', 'province']).description.agg([len])
countries_reviewed
```

Out[7]:

		len
country	province	
Argentina	Mendoza Province	3264
	Other	536
...	...	...
Uruguay	San Jose	3
	Uruguay	24

425 rows × 1 columns

In [8]:
```
mi = countries_reviewed.index
type(mi)
```

Out[8]:
```
pandas.core.indexes.multi.MultiIndex
```

Multi-indices have several methods for dealing with their tiered structure which are absent for single-level indices. They also require two levels of labels to retrieve a value. Dealing with multi-index output is a common "gotcha" for users new to pandas.

The use cases for a multi-index are detailed alongside instructions on using them in the MultiIndex / Advanced Selection section of the pandas documentation.

However, in general the multi-index method you will use most often is the one for converting back to a regular index, the `reset_index()` method:

In [9]:
```
countries_reviewed.reset_index()
```

Out[9]:

	country	province	len
0	Argentina	Mendoza Province	3264
1	Argentina	Other	536
...	...	...	...
423	Uruguay	San Jose	3
424	Uruguay	Uruguay	24

425 rows × 3 columns

Sorting

Looking again at `countries_reviewed` we can see that grouping returns data in index order, not in value order. That is to say, when outputting the result of a `groupby`, the order of the rows is dependent on the values in the index, not in the data.

To get data in the order want it in we can sort it ourselves. The `sort_values()` method is handy for this.

In [10]:
```
countries_reviewed = countries_reviewed.reset_index()
countries_reviewed.sort_values(by='len')
```

Out[10]:

	country	province	len
179	Greece	Muscat of Kefallonian	1
192	Greece	Sterea Ellada	1
...	...	...	...
415	US	Washington	8639
392	US	California	36247

425 rows × 3 columns

`sort_values()` defaults to an ascending sort, where the lowest values go first. However, most of the time we want a descending sort, where the higher numbers go first. That goes thusly:

In [11]:
```
countries_reviewed.sort_values(by='len', ascending=False)
```

Out[11]:

	country	province	len
392	US	California	36247
415	US	Washington	8639
...	...	...	...
63	Chile	Coelemu	1
149	Greece	Beotia	1

425 rows × 3 columns

To sort by index values, use the companion method `sort_index()`. This method has the same arguments and default order:

In [12]:
```
countries_reviewed.sort_index()
```

Out[12]:

	country	province	len
0	Argentina	Mendoza Province	3264
1	Argentina	Other	536
...	...	...	...
423	Uruguay	San Jose	3
424	Uruguay	Uruguay	24

425 rows × 3 columns

Finally, know that you can sort by more than one column at a time:

In [13]:
```
countries_reviewed.sort_values(by=['country', 'len'])
```

Out[13]:

	country	province	len
1	Argentina	Other	536
0	Argentina	Mendoza Province	3264
...	...	...	...
424	Uruguay	Uruguay	24
419	Uruguay	Canelones	43

425 rows × 3 columns