Lecture 04.3 MergeGroupby

September 6, 2022

1 Merge and Groupby

Duncan Callaway

This notebook gives an introduction to using Pandas' `merge` and `groupby` methods.

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

1.1 Row and column labels

The columns are identified with a list of values. Let's look at the fruit data set again:

```
[ ]: fruit_info_df = pd.read_csv('fruit_info.csv', index_col= False)
     fruit_info_df
```

1.1.1 Q: How do I print out just the columns?

```
[ ]: fruit_info_df.columns
```

1.1.2 Q: And the rows?

The rows are similarly labeled:

```
[ ]: fruit_info_df.index
```

1.2 Merging

Lets make another data frame and tack it on to the first

```
[ ]: price_df = pd.DataFrame({'price':[0.5, 0.65, 1, 0.15],
                             'frut':['apple', 'banana', 'orange', 'rasberry']})
     price_df
```

Now let's blindly merge:

```
[ ]: pd.merge(price_df,fruit_info_df)
```

What went wrong?

First, we didn't spell fruit correctly. Two ways to fix. First, specify the columns directly:

```
[ ]: pd.merge(price_df,fruit_info_df, left_on = 'frut', right_on = 'fruit')
```

Second, fix the spelling and *don't* tell pandas. In this case pandas works to figure out what's in common.

```
[ ]: price_df.columns[0]='fruit'
```

Bummer! Can't mutate index values. What to do?

```
[ ]: col_list = list(price_df.columns)
     col_list
```

```
[ ]: col_list[1] = 'fruit'
```

```
[ ]: price_df.columns = col_list
     price_df
```

```
[ ]: pd.merge(fruit_info_df,price_df)
```

Note we can use different syntax:

```
[ ]: fruit_info_df.merge(price_df)
```

1.2.1 Q: Now we're still missing raspberries – why?

Again, spelling error in the new frame. Let's fix:

```
[ ]: price_df.loc[3,'fruit'] = 'raspberry'
```

Note we could change individual entries in the data frame itself. They are mutable.

```
[ ]: fruit_info_df.merge(price_df)
```

Another few things to takeaway from this 1. Merge can be brutal. That is, it'll drop data without telling you. BUT that's if we use the default 'inner' merge. In a few lecture we'll talk about alternative ways to merge that are a little less draconian. 2. It's important to review your results. How many rows do you expect? How many do you actually get? Did something important get chucked out? The ensuing solutions are the non-glamorous tasks of data cleaning.

Note, there are other commands – `join`, `concat`, and these do similar things to `merge`.

I've found merge seems to work well for most purposes.

FWIW, `pd.concat` seems to be a little more brute force – requires more careful syntax, but likely does unexpected things less often once you understand the syntax.

```
[ ]: merged_df = fruit_info_df.merge(price_df)
     merged_df
```

We can streamline by replacing the index number with the fruit column.

1.2.2 Q: in the following, what's the `inplace` command for?

```
[ ]: merged_df.set_index('fruit', inplace = True)
     merged_df
```

1.2.3 A: It means the re-defined dataframe is assigned to the original name.

This is advantageous in memory constrained situations.

1.3 Multilevel indexing

We can also assign "multilevel" column or row names, like so:

```
[ ]: levels = [('categorical', 'color'),('quantitative',␣
      ↪'weight'),('quantitative','price')]
     levels
```

Note the use of tuples (sets of values in parentheses) in setting up multiindex. This will come again later.

```
[ ]: merged_df.columns = pd.MultiIndex.from_tuples(levels)
     merged_df
```

Now we have categories and subcategories of columns:

```
[ ]: merged_df['quantitative']
```

1.3.1 Q: How can we get data from an individual column?

Aim to get the `weight` column:

```
[ ]: merged_df[('quantitative','weight')]
```

1.4 Advanced multilevel (did not do in lecture)

Note, we can also drop and add things. With multilevel indexing things get a little tricky.

First, we can drop everything from the top level:

```
[ ]: merged_test_df = merged_df.drop(columns=[('quantitative',)], axis = 1)
     merged_test_df
```

Note that I put the column identifier inside the parens, like a tuple, but it's not essential there.

However if we want to drop only a column from the second level, we get an error without the tuple syntax:

```
[ ]: merged_test_df = merged_df.drop(columns=[('quantitative','price')], axis = 1)
     merged_test_df
```

We can also drop rows:

```
[ ]: merged_df.drop(index=[('apple')], axis = 0, inplace = True)
     merged_df
```

Note indexing multilevels with `.loc` gets a little tricky. The thing to keep in mind is that you're working with tuples in each index location:

```
[ ]: merged_df.loc['banana', ('quantitative', 'price')]
```

If you leave an entry of the tuple empty you get all values.

```
[ ]: merged_df.loc['banana', ('quantitative', )]
```

You can also loop through the columns of the multilevel data frame like this:

```
[ ]: for i, j in merged_df:
         print(merged_df.loc['banana', (i, j)])
```

Some added thoughts: 1. Multilevel indexing works for columns and index 2. It can be a powerful way to summarize your data and quickly reference subsets of it. 4. However it can also be a colossal pain in the rear – indexing with multilevel is often very hard to parse and debug.

1.5 Groupby

First, let's have another look at today's power point file. Now we'll learn about how groupby works.

Back to the notebook, let's make a toy DF (example taken from Wes McKinney's Python for Data Analysis:

```
[ ]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                        'key2' : ['one', 'two', 'one', 'two', 'one'],
                        'data1' : np.random.randn(5),
                        'data2' : np.random.randn(5)})
     df
```

Let's group just the `data1` column by the `key1` column. A call to `groupby` does that.

Note, the syntax is to begin by invoking the portion of the dataframe we want to group (here, `df['data1']`), then we apply the groupby method with the portion of hte dataframe we want to group on (here `df['key1']`)

What is the object that results?

```
[ ]: grouped = df['data1'].groupby(df['key1'])
     grouped
```

As we see, it's not simply a new DataFrame. Instead, it's an object, in this case `SeriesGroupBy`. We'll see in a moment that if we group many columns of data we get a `DataFrameGroupBy` object.

To look inside we need to use different syntax. The specific thing we're looking for are the groups of the object…but let's tab in to the grouped object to see what's there.

```
[ ]: grouped.groups
```

That gave us the groups (a and b) and the indices of elements in the groups, but nothing else.

You can see this structure looks like a dict. a and b are the keys, and the data are lists associated with each key – the values.

But the `grouped` object is capable of making computations across all groups – this is where it gets powerful.

We can try things like `sum`, `min` and `max`.

Notice if you don't put the parens after the method, pandas returns information about what the method does, but not it's actual output.

```
[ ]: grouped.sum()
```

You can also pass `numpy` functions into the aggregate command.

But it can be informative to look at what's inside. We can iterate over a `groupby` object, as we iterate we get pairs of `(name, group)`, where the `group` is either a `Series` or a `DataFrame`, depending on whether the `groupby` object is a `SeriesGroupBy` (as above) or a `DataFrameGroupBy` (see below).

Something quirky to note about the interaction between the grouped object and the for loop structure below: we're going to define variables `name` and `group` as being things in `grouped`. But there are no `name` or `group` attributes associated with the `grouped` object.

```
[ ]: for name, group in grouped:
         print('Name:', name)
         display(group)
```

We can group on multiple keys, and the result is grouping by tuples:

```
[ ]: g2 = df['data1'].groupby([df['key1'], df['key2']])
     g2
```

```
[ ]: g2.groups
```

Now we have a groupby object that has tuples as the keys.

```
[ ]: g2.mean()
```

1.5.1 Aside (did not do in lecture)

We can also group the entire dataframe – not just one column of it – on a single key. This results in a `DataFrameGroupBy` object as the result:

```
[ ]: k1g = df.groupby('key1')
     k1g
```

```
[ ]: k1g.groups
```

That output actually looks a lot like the output when we were only grouping one of the columns of the dataframe. But there is actually more information in the group itself.

```
[ ]: for name, group in k1g:
         print('Name:', name)
         display(group)
```

The contents of each group is another dataframe.

```
[ ]: k1g.mean()
```

Where did column `key2` go in the mean above? It's a *nuisance column*, which gets automatically eliminated from an operation where it doesn't make sense (such as a numerical mean).

1.5.2 Aside (did not do in lecture): Grouping over a different dimension

Above, we've been grouping data along the rows, using column keys as our selectors.

But we can also group along the *columns*,

What's even more cool? We can group by *data type*.

Here we'll group along columns, by data type:

```
[ ]: df.dtypes
```

```
[ ]: grouped = df.groupby(df.dtypes, axis=1)
     for dtype, group in grouped:
         print(dtype)
         display(group)
```

1.6 Using groupby to re-ask our question

Which hour had the lowest average wind production?

```
[ ]: cds = pd.read_csv('CAISO_2017to2018_stack.csv', index_col= 0)
```

```
[ ]: cds.head()
```

It will help to have a column of hour of day values:

```
[ ]: cds_time = pd.to_datetime(cds.index)
     type(cds_time)
```

Let's add that list of values into the data frame.

```
[ ]: cds['hour'] = cds_time.hour
```

```
[ ]: cds.head(10)
```

1.6.1 Q: What groupby syntax would you use to arrange the data…

…so that you can examine production by hour and source?

See if you can do it yourself: we want to group MWh values by source AND hour.

```
[ ]: cds_grouped = cds['MWh'].groupby([cds['Source'],cds['hour']])
```

1.6.2 Q: How to get *all* the means for all sources and hours?

Didn't need to do any fancy logical indexing or looping!

```
[ ]: cds_grouped.mean()
```

Now it would be nice to see that information in a dataframe, wouldn't it?

```
[ ]: averages = pd.DataFrame(cds_grouped.mean())
```

```
[ ]: averages
```

And lo and behold, we have a multilevel index for the rows!

```
[ ]: averages.loc[('WIND TOTAL',),:]
```

But now we can look at other sources

```
[ ]: averages.loc[('SMALL HYDRO',),:]
```

Let's plot:

```
[ ]: import matplotlib.pyplot as plt
```

```
[ ]: plt.plot(averages.loc[('SMALL HYDRO',),:]);
```

```
[ ]: plt.plot(averages.loc[('GEOTHERMAL',),:]);
```

```
[ ]: plt.plot(averages.loc[('SOLAR PV',),:]);
```

1.7 Pivot

Pivot is used to examine aggregates with respect to two characteristics. You might construct a pivot of sales data if you wanted to look at average sales broken down by year and market.

The pivot operation is essentially a `groupby` operation that transforms the rows *and the columns.*

[ ]:
```
cds.pivot_table(
    values  = 'MWh', # the entry to aggregate over
    index   = 'hour',  # the row grouping attributes
    columns = 'Source',    # the column grouping attributes
    aggfunc = 'mean'   # the aggregation function
)
```

1.7.1 Q: In class challenge:

create a pivot table where the columns are the hours, source is the row and the returned value is the standard deviation.

Hint: write `std` to represent standard deviation.

[ ]:
```
cds.pivot_table(
    values  = 'MWh',
    index   = 'Source',
    columns = 'hour',
    aggfunc = 'std'
)
```