Lecture 5 Notebook

September 6, 2018

```
In [ ]: import pandas as pd
```

0.1 Review Merge

First let's look at today's power point slides (5 Merge-Groupby-EnergyAndDevelopment.pptx).
ppt slides show an overview of how merging works.

0.2 Merging

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

```
In [ ]: fruit_info={'fruit':['apple','banana','orange','raspberry'],
                              'color':['red','yellow','orange','pink'],
                              'weight':[120,150,250,15]
                  }
        fruit_info_df = pd.DataFrame(data = fruit_info)
        fruit_info_df
```

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

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

Note, the order was different for each row, but Pandas figured it out.
Note, there are other commands -- `join`, `concat`, and these do similar things.
I haven't learned enough to carefully choose between them, but merge seems to work well, and `join` seems to be less functional.
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.
We can streamline by replacing the index number with the fruit column.
What's the `inplace` command for? It means the re-defined dataframe is assigned to the original name. This is advantageous in memory constrained situations.

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

Here's a cool little factoid about data frames: you can write for loops that burn through the columns of the frame.

```
In [ ]: for i in merged_df:
            print(merged_df.loc['raspberry',i])
```

0.3 Groupby

(these notes adapted from last Spring's DS100 notebook)

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:

```
In [ ]: import numpy as np
```

```
In [ ]: 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?

```
In [ ]: 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.

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

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

If we call `grouped.groups` elements, we don't get much of use; we wind up just retrieving the elements of the list above:

```
In [ ]: grouped.groups['a'][2]
```

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

We can try things like `.count()`, `.min()` and `.mean()`.

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.

```
In [ ]: grouped.aggregate(min)
```

There are a number of functions you can pass in to `aggregate`, like `sum` and `max`.

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):

```
In [ ]: from IPython.display import display  # like print, but for complex objects

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

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

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

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

Let's look at the dataframe again, for a reminder:

```
In [ ]: df
```

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

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:

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

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

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

But let's look at what's inside of k1g:

```
In [ ]: for n, g in k1g:
            print('name:', n)
            display(g)
```

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).

0.3.1 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:

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

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

0.4 Let's take the quiz.

0.5 Using groupby to re-ask our question

Which hour had the lowest average wind production?

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

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

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

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

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

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

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

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

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

```
In [ ]: cds_grouped.groups
```

Now we can see *all* the means for all sources and hours.
Didn't need to do any fancy logical indexing or looping!

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

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

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

```
In [ ]: averages
```

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

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

But now we can look at other sources

```
In [ ]: averages.index
```

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

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

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

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

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

```
In [ ]: plt.plot(cds.loc[cds.loc[:,'Source']=='SOLAR PV','MWh'])
```

0.6 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*. For example consider the following **groupby** operation:

```
In [ ]: cds_grouped.groups
```

We can use `pivot` to do similar things:

```
In [ ]: 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
        )
```

In class test: create a pivot table where the columns are the hours, source is the column and the returned value is the standard deviation.

Hint: write `std` to represent standard deviation.

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