Lecture 4 Notebook

September 4, 2018

1 Lecture 4 Notebook

Duncan Callaway September 4 2018

This lecture continues introducing the class to Pandas and goes into "groupby"

In class I worked in "Duncan's Lecture 4 in class workbook.ipynb"

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

1.1 Recap last lecture

Data frame vs dict of lists

```
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)
        print(fruit_info)
        fruit_info_df
```

The data frame has 1. column headers, 2. An index column 2. rows 4. columns 5. numeric and text entries -- but columns are all the same type.

Note, last time I tried getting the "index" valus from teh fruit info dataframe:

"the index" Note this is different from locational indexing. We're talking about the column that identifies the row of the frame.

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

What I was *expecting* was a list from 0 to 3, e.g. `[0, 1, 2, 3]`

But I got the above. This is just an alternative way of giving the same information. However note, if I do this:

```
In [ ]: fruit_info_df.index = ['zero', 'one', 'two', 'three']
```

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

...then I get what I expected. More on indices in a moment.

loc and iloc loc identifies location by column and header names.

```
In [ ]: fruit_info_df.loc['zero':'two', 'fruit':'weight']
```

note, loc is inclusive!
.iloc identifies location by number -- just like indexing in numpy.

```
In [ ]: fruit_info_df.iloc[0:3,1:3]
```

.iloc is exclusive on the end location value.

1.2 Back to our question: which hour had the most wind...

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

Let's make the name shorter to save me typing:

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

Let's look at some info about the data:

```
In [ ]: cds.shape
```

```
In [ ]: cds.size
```

What did those two commands give us? `shape`: (number of rows, number of columns of *data*) `size`: total number of cell entries.
Notice these numbers don't include what's in the index.
Here's something fun --

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

1.3 Logical indexing

Logical indexing is an extremely powerful way to pull data out of a frame.
For example, with the stacked data frame, let's pull out only wind generation.
First, I'll show you a boolean series based on comparisons to the 'Source' data column:

```
In [ ]: (cds['Source']=='WIND TOTAL').head()
```

Now we can embed that inside the `.loc` method:

```
In [ ]: cds.loc[cds['Source']=='WIND TOTAL',:].head()
```

Ok. Any ideas how we can use that to get the information we want? Reminder, the question is:
What hour of the day had the lowest average wind power in California in the last 12 months?

```
In [ ]: wind = cds.loc[cds['Source']=='WIND TOTAL',:]
```

What is the data structure of `wind`?

```
In [ ]: type(wind)
```

Next week we'll use pivots to do this better, but for now let's use a for loop to get information by hour.

First thing to do is figure out how to get the hour out of the index.

`datetime.strptime` is useful for this if you're working on individual dates.

But `pd.to_datetime` is even better, especially if you're working on a lot of values in a list (or as the case will be, values in a pandas series).

```
In [ ]: windex = pd.to_datetime(wind.index)
        windex.hour
```

```
In [ ]: wind_ave = [] # initalizes a list to populate
        for i in range(0,24):
            wind_ave.append(np.mean(wind.loc[windex.hour == i,:]))
```

```
In [ ]: print(wind_ave)
```

```
In [ ]: type(wind_ave)
```

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

```
In [ ]: plt.plot(wind_ave)
```

We can see pretty clearly that the min is 10 or 11...let's dig a little more.

One way to do this is to drop the data into a data frame and then *sort* the data frame.

```
In [ ]: df_wind = pd.DataFrame(wind_ave)
        df_wind
```

I'm going to be adding more MWh values to the data frame in just a moment, so let's be clear that this is the average

```
In [ ]: df_wind.columns = ['Average MWh']
```

```
In [ ]: df_wind.sort_values(by='Average MWh',ascending=True).head()
```

Ok -- so it looks as though mid-day is the minimum *average*.

Nice to see that the index values were preserved

But what's the range?

```
In [ ]: wind_min = [] # initalizes a list to populate
        wind_max = [] # initalizes a list to populate
        for i in range(0,24):
            wind_min.append(np.min(wind.loc[windex.hour == i,:]))
            wind_max.append(np.max(wind.loc[windex.hour == i,:]))
```

```
In [ ]: wind_max[0]
```

```
In [ ]: df_wind['min MWh']=pd.DataFrame(wind_min)['MWh']
        df_wind['max MWh']=pd.DataFrame(wind_max)['MWh']
```

```
In [ ]: df_wind
```

```
In [ ]: plt.plot(df_wind)
```

1.4 Row and column labels

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

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

```
In [ ]: type(fruit_info_df.columns)
```

The rows are similarly labeled:

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

```
In [ ]: type(fruit_info_df.index)
```

They are both the same data type within Pandas -- the "Index"
Note, we can do a bunch of other stuff:

1.5 Merging

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

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

```
In [ ]: fruit_info_df
```

```
In [ ]: 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:

```
In [ ]: 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.

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

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

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

```
In [ ]: col_list[0] = 'fruit'
```

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

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

Note we can use different syntax:

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

Now we're still missing raspberries -- why?
Again, spelling error in the new frame. Let's fix:

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

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

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

Note the fruit_info data frame is still intact, you'd need to assign it to a data frame name to save it.

```
In [ ]: fruit_info_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 fruit_info_df:
            print(fruit_info_df.loc['one',i])
```

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.
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.

```
In [ ]: pd.concat([fruit_info_df,price_df])
```

You can see in the above that setting `axis` equal to 1 will join by appending columns rather than rows.

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

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
```

1.6 Multilevel indexing

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

```
In [ ]: 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.

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

Now we have categories and subcategories of columns:

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

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:

```
In [ ]: 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:

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

We can also drop rows:

```
In [ ]: 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:

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

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

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

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

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

1.7 Groupby

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

Let's make a toy DF (example taken from Wes McKinney's Python for Data Analysis:

```
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.count()
```

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).

1.7.1 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)
```

1.8 Let's take the quiz.

1.9 Using groupby to re-ask our question

Which hour had the lowest average wind production?

```
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 [ ]: 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'])
```