Acquiring and wrangling with data

April 23, 2015

In previous sessions, we've talked about the underlying data structures in the `Pandas` library. We've seen how to manipulate `DataFrame` and `Series` objects in order to answer questions regarding the data. Last week, we also saw how to use `matplotlib` to visualize our analysis.

This week we will be diving into an important topic in `Pandas`: aggregating and performing operations on specified groups of data without modifying the underlying structure. According to Wes McKinney,

> Categorizing a data set and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow. After loading, merging, and preparing a data set, a familiar task is to compute group statistics or possibly *pivot tables* for reporting or visualization purposes. `Pandas` provides a flexible and high-performance `groupby` facility, enabling you to slice and dice, and summarize data sets in a natural way.

```
In [1]: from __future__ import division
        from numpy.random import randn
        import numpy as np
        import os
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        plt.rc('figure', figsize=(10, 6))
        from pandas import Series, DataFrame
        import pandas as pd
        np.set_printoptions(precision=4)
```

1 Data Aggregation and Group Operations

```
In [2]: pd.options.display.notebook_repr_html = False
        %matplotlib inline
```

1.1 GroupBy mechanics

`Pandas` was designed with a considerable deference to the progress made in data aggregation techniques by developers for the `R` programming language. The main mechanism is the *split-apply-combine* paradigm:

1. Data is *split* into groups based on one or more provided *keys*,
2. A function is *applied* to each group,
3. The results of all the function applications are *combined* into a result object.

As we will see, grouping keys are very flexible in nature. Some possible types of keys are

- A list or array of values sharing the length of the grouped column
- A value indicating a column name
- A dict or `Series` giving a correspondence between the values on the axis being grouped and the group names

- A function to be invoked on the axis index or the individual labels in the index

```
In [3]: df = DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                        'key2' : ['one', 'two', 'one', 'two', 'one'],
                        'data1' : np.random.randn(5),
                        'data2' : np.random.randn(5)})
        df
```

```
Out[3]:       data1     data2 key1 key2
        0 -0.204708  1.393406    a  one
        1  0.478943  0.092908    a  two
        2 -0.519439  0.281746    b  one
        3 -0.555730  0.769023    b  two
        4  1.965781  1.246435    a  one
```

We can compute the mean of the column corresponding to "data1" by using the group labels from "key1". This can be done in a number of ways, but here is a straightforward example:

```
In [4]: grouped = df['data1'].groupby(df['key1'])
        grouped
```

```
Out[4]: <pandas.core.groupby.SeriesGroupBy object at 0x7f37d9157050>
```

Notice that `grouped` is its own `Pandas` object, just like a `Series` or `DataFrame`. Now we can compute simple statistics just like we would with other objects.

```
In [5]: grouped.mean()
```

```
Out[5]: key1
        a       0.746672
        b      -0.537585
        Name: data1, dtype: float64
```

We can also pass an array of keys:

```
In [6]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()
        means
```

```
Out[6]: key1  key2
        a     one     0.880536
              two     0.478943
        b     one    -0.519439
              two    -0.555730
        Name: data1, dtype: float64
```

This forms a grouping using a heierarchical index, as we have seen earlier. To flatten the hierarchical index, as we have seen, we can call `unstack()`.

```
In [7]: means.unstack()
```

```
Out[7]: key2       one       two
        key1
        a     0.880536  0.478943
        b    -0.519439 -0.555730
```

In these examples, the keys refer to `Series`, though they could really be anything so long as the lengths match up. For example, consider the following key arrays.

```
In [8]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
        years = np.array([2005, 2005, 2006, 2005, 2006])
        df['data1'].groupby([states, years]).mean()
```

```
Out[8]: California  2005    0.478943
                    2006   -0.519439
        Ohio        2005   -0.380219
                    2006    1.965781
        Name: data1, dtype: float64
```

If you're just interested in the column names, you can simply pass the identifying string, or list of strings, as in the following examples:

```
In [9]: df.groupby('key1').mean()
```

```
Out[9]:          data1     data2
        key1
        a     0.746672  0.910916
        b    -0.537585  0.525384
```

```
In [10]: df.groupby(['key1', 'key2']).mean()
```

```
Out[10]:               data1     data2
         key1 key2
         a    one   0.880536  1.319920
              two   0.478943  0.092908
         b    one  -0.519439  0.281746
              two  -0.555730  0.769023
```

```
In [11]: df.groupby(['key1', 'key2']).size()
```

```
Out[11]: key1  key2
         a     one     2
               two     1
         b     one     1
               two     1
         dtype: int64
```

1.1.1 Iterating over groups

`groupby()` supports iteration. Specifically, `groupby()` produces tuples containing the group name along the relevant data. For example:

```
In [12]: for name, group in df.groupby('key1'):
             print(name)
             print(group)
```

```
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

If multiple keys are being passed, you can overload the name by making a n-tuple of keys. For example:

```
In [13]: for (k1, k2), group in df.groupby(['key1', 'key2']):
             print((k1, k2))
             print(group)

('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
     data1     data2 key1 key2
3 -0.55573  0.769023    b  two
```

This process is in general quite flexible. For example, suppose you want to store the relevant `DataFrame` groups as a native-`Python` dict. In this case, we can just wrap the `groupby()` call with a list and a dict, which will thus be stored into the dict.

```
In [14]: pieces = dict(list(df.groupby('key1')))
         pieces['b']

Out[14]:       data1     data2 key1 key2
         2 -0.519439  0.281746    b  one
         3 -0.555730  0.769023    b  two
```

By default, `groupby` groups on `axis=0`, which corresponds to treating columns of data. Of course, you can specify which axis you desire.

```
In [15]: df.dtypes

Out[15]: data1    float64
         data2    float64
         key1      object
         key2      object
         dtype: object

In [16]: grouped = df.groupby(df.dtypes, axis=1)
         dict(list(grouped))

Out[16]: {dtype('float64'):       data1     data2
          0 -0.204708  1.393406
          1  0.478943  0.092908
          2 -0.519439  0.281746
          3 -0.555730  0.769023
          4  1.965781  1.246435, dtype('O'):   key1 key2
          0    a  one
          1    a  two
          2    b  one
          3    b  two
          4    a  one}
```

1.1.2 Selecting a column or subset of columns

Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the efect of *selecting those columns* for aggregation. This means that:

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

is effectively identical to

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Why is this useful? If you're working with a large dataset for which aggregating the entire DataFrame is out of the question, you can speed up the process by specifying the particular columns you are interested in.

Here's an example: we can group a DataFrame according to a set of keys, specify a particular column (in this case, data2), and take the resulting mean.

```
In [17]: df.groupby(['key1', 'key2'])[['data2']].mean()
```

```
Out[17]:                data2
         key1 key2
         a    one   1.319920
              two   0.092908
         b    one   0.281746
              two   0.769023
```

The object returned here is a grouped DataFrame if a list or array is passed and a grouped Series if just a column name is passed.

```
In [18]: s_grouped = df.groupby(['key1', 'key2'])['data2']
         s_grouped
```

```
Out[18]: <pandas.core.groupby.SeriesGroupBy object at 0x7f37d9119350>
```

```
In [19]: s_grouped.mean()
```

```
Out[19]: key1  key2
         a     one     1.319920
               two     0.092908
         b     one     0.281746
               two     0.769023
         Name: data2, dtype: float64
```

1.1.3 Grouping with dicts and Series

Grouping information may exist in a form other than an array. Let's consider another example DataFrame.

```
In [20]: people = DataFrame(np.random.randn(5, 5),
                            columns=['a', 'b', 'c', 'd', 'e'],
                            index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
         people.ix[2:3, ['b', 'c']] = np.nan # Add a few NA values
         people
```

```
Out[20]:                a         b         c         d         e
         Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
         Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
         Wes    -0.539741       NaN       NaN -1.021228 -0.577087
         Jim     0.124121  0.302614  0.523772  0.000940  1.343810
         Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Suppose we have a group correspondence for the columns and want to sum together the columns by group.

```
In [21]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
                    'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

The `groupby` method can use the dict natively, allowing you to form `GroupBy` objects on the fly.

```
In [22]: by_column = people.groupby(mapping, axis=1)
         by_column.sum()
```

```
Out[22]:            blue       red
         Joe     0.503905  1.063885
         Steve   1.297183 -1.553778
         Wes    -1.021228 -1.116829
         Jim     0.524712  1.770545
         Travis -4.230992 -2.405455
```

The same holds for `Series` objects, which are structurally similar to dicts.

```
In [23]: map_series = Series(mapping)
         map_series
```

```
Out[23]: a       red
         b       red
         c      blue
         d      blue
         e       red
         f    orange
         dtype: object
```

```
In [24]: people.groupby(map_series, axis=1).count()
```

```
Out[24]:         blue  red
         Joe        2    3
         Steve      2    3
         Wes        1    2
         Jim        2    3
         Travis     2    3
```

1.1.4 Grouping with functions

In `Python`, functions are simply another data type. You can actually use `groupby` to isolate members of your data set according to a rule, defined by a function. For example:

```
In [25]: people.groupby(len).sum()
```

```
Out[25]:          a         b         c         d         e
         3  0.591569 -0.993608  0.798764 -0.791374  2.119639
         5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
         6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

You can mix and match functions with other data types that we have seen above.

```
In [26]: key_list = ['one', 'one', 'one', 'two', 'two']
         people.groupby([len, key_list]).min()
```

```
Out[26]:              a         b         c         d         e
         3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
           two  0.124121  0.302614  0.523772  0.000940  1.343810
         5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
         6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

1.1.5 Grouping by index levels

Finally, you can use heierarchical indexing to perform `groupby` operations. To do this, pass the level number or name using the `level` keyword.

```
In [27]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
                                              [1, 3, 5, 1, 3]],
                                              names=['cty', 'tenor'])
         hier_df = DataFrame(np.random.randn(4, 5), columns=columns)
         hier_df
```

```
Out[27]: cty          US                            JP
         tenor         1         3         5         1         3
         0      0.560145 -1.265934  0.119827 -1.063512  0.332883
         1     -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
         2      0.286350  0.377984 -0.753887  0.331286  1.349742
         3      0.069877  0.246674 -0.011862  1.004812  1.327195
```

```
In [28]: hier_df.groupby(level='cty', axis=1).count()
```

```
Out[28]: cty  JP  US
         0     2   3
         1     2   3
         2     2   3
         3     2   3
```

1.2 Data aggregation

Wes McKinney defines *data aggregation* as any transformation that takes a dataset or other array and produces scalar values. For example, the simple statistical functions such as

- mean
- max
- min
- sum

are examples of operations taking arrays to numbers. Many of the aggregations that we have seen so far have been optimized for performance, but `Pandas` gives you the functionality to implement customized aggregators.

```
In [29]: df
```

```
Out[29]:       data1     data2 key1 key2
         0 -0.204708  1.393406    a  one
         1  0.478943  0.092908    a  two
         2 -0.519439  0.281746    b  one
         3 -0.555730  0.769023    b  two
         4  1.965781  1.246435    a  one
```

For example, we can use `quantile(x)` (not explicitly implemented for `GroupBy`), which determines the value of xth percentile given a `Series` of data.

```
In [30]: grouped = df.groupby('key1')
         grouped['data1'].quantile(0.9)
```

```
Out[30]: key1
         a       1.668413
         b      -0.523068
         Name: data1, dtype: float64
```

More to the point, you can define your own functions and do groupby operations with them. For example, if you are interested in the range of your data sets, you can use:

```
In [31]: def peak_to_peak(arr):
             return arr.max() - arr.min()
         grouped.agg(peak_to_peak)
```

```
Out[31]:          data1     data2
         key1
         a     2.170488  1.300498
         b     0.036292  0.487276
```

Even method that aren't really aggregations, such as `describe`, still perform useful operations on `GroupBy` objects.

```
In [32]: grouped.describe()
```

```
Out[32]:                 data1     data2
         key1
         a    count  3.000000  3.000000
              mean   0.746672  0.910916
              std    1.109736  0.712217
              min   -0.204708  0.092908
              25%    0.137118  0.669671
              50%    0.478943  1.246435
              75%    1.222362  1.319920
              max    1.965781  1.393406
         b    count  2.000000  2.000000
              mean  -0.537585  0.525384
              std    0.025662  0.344556
              min   -0.555730  0.281746
              25%   -0.546657  0.403565
              50%   -0.537585  0.525384
              75%   -0.528512  0.647203
              max   -0.519439  0.769023
```

We will continue with the `tips` dataset from previous weeks to show off some of the more advanced features of aggregation. The data set can be found on the course webpage, or here, if you're lazy (like us).

```
In [33]: tips = pd.read_csv('tips.csv')
         # Add tip percentage of total bill
         tips['tip_pct'] = tips['tip'] / tips['total_bill']
         tips[:6]
```

```
Out[33]:    total_bill   tip     sex smoker  day    time  size   tip_pct
         0       16.99  1.01  Female     No  Sun  Dinner     2  0.059447
         1       10.34  1.66    Male     No  Sun  Dinner     3  0.160542
         2       21.01  3.50    Male     No  Sun  Dinner     3  0.166587
         3       23.68  3.31    Male     No  Sun  Dinner     2  0.139780
         4       24.59  3.61  Female     No  Sun  Dinner     4  0.146808
         5       25.29  4.71    Male     No  Sun  Dinner     4  0.186240
```

1.2.1 Column-wise and multiple function application

As we've seen, aggregating a `Series` or all of the columns of a `DataFrame` is a matter of using `aggregate` with the desired function or calling a method like `mean` or `std`. However, you may want to aggregate using a different function depending on the column or multiple functions at once. Fotunately, this is straightforward to do, which we will illustrate through a number of examples. First, let's group the tips by `sex` and `smoker`.

```
In [34]: grouped = tips.groupby(['sex', 'smoker'])
```

Descriptive statistics, such as `mean`, can be passed to the aggregator as a string.

```
In [35]: grouped_pct = grouped['tip_pct']
         grouped_pct.agg('mean')
```

```
Out[35]: sex     smoker
         Female  No        0.156921
                 Yes       0.182150
         Male    No        0.160669
                 Yes       0.152771
         Name: tip_pct, dtype: float64
```

You can also pass a list of functions to do aggregation. If the function is built-in, it passes as a string. Otherwise, one can simply pass the function on its own.

```
In [36]: grouped_pct.agg(['mean', 'std', peak_to_peak])
```

```
Out[36]:                    mean       std  peak_to_peak
         sex    smoker
         Female No      0.156921  0.036421      0.195876
                Yes     0.182150  0.071595      0.360233
         Male   No      0.160669  0.041849      0.220186
                Yes     0.152771  0.090588      0.674707
```

To label the columns assigned by `agg`, pass a tuple in for each function, with the first element corresponding to the label of the column.

```
In [37]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
```

```
Out[37]:                     foo       bar
         sex    smoker
         Female No      0.156921  0.036421
                Yes     0.182150  0.071595
         Male   No      0.160669  0.041849
                Yes     0.152771  0.090588
```

With a `DataFame` you have more options as you can specify as list of functions to apply to all of the columns or different functions per column.

```
In [38]: functions = ['count', 'mean', 'max']
         result = grouped['tip_pct', 'total_bill'].agg(functions)
         result
```

```
Out[38]:              tip_pct                     total_bill
                        count      mean       max      count       mean    max
         sex    smoker
         Female No         54  0.156921  0.252672         54  18.105185  35.83
                Yes        33  0.182150  0.416667         33  17.977879  44.30
         Male   No         97  0.160669  0.291990         97  19.791237  48.33
                Yes        60  0.152771  0.710345         60  22.284500  50.81
```

Here we are using what effectively amounts to a hierarchical index, which we can then slice by choosing columns and subcolumns.

```
In [39]: result['tip_pct']
```

```
Out[39]:                  count      mean       max
        sex    smoker
        Female No         54  0.156921  0.252672
               Yes        33  0.182150  0.416667
        Male   No         97  0.160669  0.291990
               Yes        60  0.152771  0.710345
```

As above, a list of tuples with custom names can be passed:

```
In [40]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
         grouped['tip_pct', 'total_bill'].agg(ftuples)
```

```
Out[40]:                      tip_pct                total_bill
                         Durchschnitt Abweichung Durchschnitt Abweichung
        sex    smoker
        Female No         0.156921   0.001327    18.105185  53.092422
               Yes        0.182150   0.005126    17.977879  84.451517
        Male   No         0.160669   0.001751    19.791237  76.152961
               Yes        0.152771   0.008206    22.284500  98.244673
```

You can also specify the particular column of a `DataFrame` that you want to do aggregation on by passing a dict of information. For example,

```
In [41]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
```

```
Out[41]:                   tip  size
        sex    smoker
        Female No          5.2   140
               Yes         6.5    74
        Male   No          9.0   263
               Yes        10.0   150
```

You can overload a particular column's aggregator functions by making the dict key corresponding to the column reference a list rather than just one function.

```
In [42]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
                      'size' : 'sum'})
```

```
Out[42]:                   tip_pct                                   size
                              min       max      mean       std   sum
        sex    smoker
        Female No        0.056797  0.252672  0.156921  0.036421   140
               Yes       0.056433  0.416667  0.182150  0.071595    74
        Male   No        0.071804  0.291990  0.160669  0.041849   263
               Yes       0.035638  0.710345  0.152771  0.090588   150
```

1.2.2 Returning aggregated data in "unindexed" form

Of course, you can unindex a hierarchical indexed `GroupBy` object by Specifying the `as_index` optional paramter.

```
In [43]: tips.groupby(['sex', 'smoker'], as_index=False).mean()
```

```
Out[43]:       sex smoker  total_bill       tip      size   tip_pct
        0  Female     No   18.105185  2.773519  2.592593  0.156921
        1  Female    Yes   17.977879  2.931515  2.242424  0.182150
        2    Male     No   19.791237  3.113402  2.711340  0.160669
        3    Male    Yes   22.284500  3.051167  2.500000  0.152771
```

1.3 Group-wise operations and transformations

Aggregation is but one kind of group operation. It is a special case of more general data transformations, taking one-dimensional arrays and reducing them to scalars. Here we will generalize this notion by showing you how to use `apply` and `transform` methods on `DataFrame` objects. Let's revisit our old `DataFrame` of random data.

```
In [44]: df
```

```
Out[44]:       data1     data2 key1 key2
         0 -0.204708  1.393406    a  one
         1  0.478943  0.092908    a  two
         2 -0.519439  0.281746    b  one
         3 -0.555730  0.769023    b  two
         4  1.965781  1.246435    a  one
```

Suppose we want to add a column containing group means for each index. One way to do this is to aggregate, then merge:

```
In [45]: k1_means = df.groupby('key1').mean().add_prefix('mean_')
         k1_means
```

```
Out[45]:       mean_data1  mean_data2
         key1
         a       0.746672    0.910916
         b      -0.537585    0.525384
```

```
In [46]: pd.merge(df, k1_means, left_on='key1', right_index=True)
```

```
Out[46]:       data1     data2 key1 key2  mean_data1  mean_data2
         0 -0.204708  1.393406    a  one    0.746672    0.910916
         1  0.478943  0.092908    a  two    0.746672    0.910916
         4  1.965781  1.246435    a  one    0.746672    0.910916
         2 -0.519439  0.281746    b  one   -0.537585    0.525384
         3 -0.555730  0.769023    b  two   -0.537585    0.525384
```

This works but is somewhat inflexible. You can think of the operation as transforming the two data columns using the `np.mean` function. Returning to the `people DataFrame` from before, we can use the `transform` method on `GroupBy`.

```
In [47]: key = ['one', 'two', 'one', 'two', 'one']
         people.groupby(key).mean()
```

```
Out[47]:             a         b         c         d         e
         one -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
         two  0.505275 -0.849512  0.075965  0.834983  0.452620
```

```
In [48]: people.groupby(key).transform(np.mean)
```

```
Out[48]:                a         b         c         d         e
         Joe    -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
         Steve   0.505275 -0.849512  0.075965  0.834983  0.452620
         Wes    -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
         Jim     0.505275 -0.849512  0.075965  0.834983  0.452620
         Travis -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
```

What is going on here is that `transform` applies a function to each group, then places the results in the appropriate locations. When this reduces to the special case of scalar values, the answer is simply broadcasted across all the relevant locations.

Suppose instead you wanted to subtract the mean value from each group. To do so, let's define a function `demean`, and proceed by

```
In [49]: def demean(arr):
             return arr - arr.mean()
         demeaned = people.groupby(key).transform(demean)
         demeaned
```

```
Out[49]:                a         b         c         d         e
         Joe     1.089221 -0.232534  1.322612  1.113271  1.381226
         Steve   0.381154 -1.152125 -0.447807  0.834043 -0.891190
         Wes    -0.457709       NaN       NaN -0.136869 -0.548778
         Jim    -0.381154  1.152125  0.447807 -0.834043  0.891190
         Travis -0.631512  0.232534 -1.322612 -0.976402 -0.832448
```

You can check that `demeanded` now has zero group means:

```
In [50]: demeaned.groupby(key).mean()
```

```
Out[50]:                 a             b  c             d  e
         one  0.000000e+00 -1.110223e-16  0  7.401487e-17  0
         two -2.775558e-17  0.000000e+00  0  0.000000e+00  0
```

We will soon see that demeaning can be achieved using `apply` as well.

1.3.1 Apply: General split-apply-combine

There are three data transformation tools that we can use to build analyses on our data. The first two, `aggregate` and `transform`, are somewhat rigid in their capabilities. On the flip side, this makes it easier on you the data analyst to perform data transformations. The third tool is `apply`, which gives you immense flexibility at the expense of intuitivity.

Returning to the `tips.csv` data set, suppose we want to select the top five `tip_pct` values by group. We can write a function to identify the top values of a `DataFrame` very easily:

```
In [51]: def top(df, n=5, column='tip_pct'):
             return df.sort_index(by=column)[-n:]
         top(tips, n=6)
```

```
Out[51]:      total_bill   tip     sex smoker  day    time  size   tip_pct
         109       14.31  4.00  Female    Yes  Sat  Dinner     2  0.279525
         183       23.17  6.50    Male    Yes  Sun  Dinner     4  0.280535
         232       11.61  3.39    Male     No  Sat  Dinner     2  0.291990
         67         3.07  1.00  Female    Yes  Sat  Dinner     1  0.325733
         178        9.60  4.00  Female    Yes  Sun  Dinner     2  0.416667
         172        7.25  5.15    Male    Yes  Sun  Dinner     2  0.710345
```

Now if we group by `smoker`, say, and call `apply` with this function, we get

```
In [52]: tips.groupby('smoker').apply(top)
```

```
Out[52]:            total_bill   tip     sex smoker   day    time  size   tip_pct
         smoker
         No     88        24.71  5.85    Male     No  Thur   Lunch     2  0.236746
                185       20.69  5.00    Male     No   Sun  Dinner     5  0.241663
```

```
     51        10.29  2.60  Female     No   Sun  Dinner     2  0.252672
     149        7.51  2.00    Male     No  Thur   Lunch     2  0.266312
     232       11.61  3.39    Male     No   Sat  Dinner     2  0.291990
Yes  109       14.31  4.00  Female    Yes   Sat  Dinner     2  0.279525
     183       23.17  6.50    Male    Yes   Sun  Dinner     4  0.280535
     67         3.07  1.00  Female    Yes   Sat  Dinner     1  0.325733
     178        9.60  4.00  Female    Yes   Sun  Dinner     2  0.416667
     172        7.25  5.15    Male    Yes   Sun  Dinner     2  0.710345
```

top is called on each piece of the DataFrame, then the results are glued together using pandas.concat, labeling the pieces with the group names.

If you pass a function to apply that takes other arguments or keywords, you can pass these after the function:

```
In [53]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
```

```
Out[53]:                  total_bill    tip     sex smoker   day    time  size  \
         smoker day
         No     Fri   94       22.75   3.25  Female     No   Fri  Dinner     2
                Sat   212      48.33   9.00    Male     No   Sat  Dinner     4
                Sun   156      48.17   5.00    Male     No   Sun  Dinner     6
                Thur  142      41.19   5.00    Male     No  Thur   Lunch     5
         Yes    Fri   95       40.17   4.73    Male    Yes   Fri  Dinner     4
                Sat   170      50.81  10.00    Male    Yes   Sat  Dinner     3
                Sun   182      45.35   3.50    Male    Yes   Sun  Dinner     3
                Thur  197      43.11   5.00  Female    Yes  Thur   Lunch     4

                          tip_pct
         smoker day
         No     Fri   94   0.142857
                Sat   212  0.186220
                Sun   156  0.103799
                Thur  142  0.121389
         Yes    Fri   95   0.117750
                Sat   170  0.196812
                Sun   182  0.077178
                Thur  197  0.115982
```

Recall that describe seems to work okay on a GroupBy object.

```
In [54]: result = tips.groupby('smoker')['tip_pct'].describe()
         result
```

```
Out[54]: smoker
         No      count    151.000000
                 mean       0.159328
                 std        0.039910
                 min        0.056797
                 25%        0.136906
                 50%        0.155625
                 75%        0.185014
                 max        0.291990
         Yes     count     93.000000
                 mean       0.163196
                 std        0.085119
```

```
         min        0.035638
         25%        0.106771
         50%        0.153846
         75%        0.195059
         max        0.710345
dtype: float64
```

In [55]: `result.unstack('smoker')`

Out[55]:

smoker	No	Yes
count	151.000000	93.000000
mean	0.159328	0.163196
std	0.039910	0.085119
min	0.056797	0.035638
25%	0.136906	0.106771
50%	0.155625	0.153846
75%	0.185014	0.195059
max	0.291990	0.710345

What's really happening (for all you 151ers) is that when you invoke a method like `describe`, it is actually just a shortcut for:

```
f = lambda x: x.describe()
grouped.apply(f)
```

Suppressing the group keys One point of style: if you prefer not working with hierarchical indices, you can specify in the `groupby` call to treat the underlying `DataFrame` as flat by choosing `group_keys=False`.

In [56]: `tips.groupby('smoker', group_keys=False).apply(top)`

Out[56]:

	total_bill	tip	sex	smoker	day	time	size	tip_pct
88	24.71	5.85	Male	No	Thur	Lunch	2	0.236746
185	20.69	5.00	Male	No	Sun	Dinner	5	0.241663
51	10.29	2.60	Female	No	Sun	Dinner	2	0.252672
149	7.51	2.00	Male	No	Thur	Lunch	2	0.266312
232	11.61	3.39	Male	No	Sat	Dinner	2	0.291990
109	14.31	4.00	Female	Yes	Sat	Dinner	2	0.279525
183	23.17	6.50	Male	Yes	Sun	Dinner	4	0.280535
67	3.07	1.00	Female	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Female	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Male	Yes	Sun	Dinner	2	0.710345

1.3.2 Example: Filling missing values with group-specific values

When cleaning up missing data, in some cases you will filter out data observations using `dropna`, but in others you may want to impute (fill in) the NA values using a fixed value or some value derived from the data. `fillna` is the right tool to use; for example, we can fill in NA values with the mean:

In [57]:
```
s = Series(np.random.randn(6))
s[::2] = np.nan
s
```

Out[57]:
```
0         NaN
1   -1.549106
2         NaN
3    0.758363
```

```
4         NaN
5    0.862580
dtype: float64
```

```
In [58]: s.fillna(s.mean())
```

```
Out[58]: 0    0.023946
         1   -1.549106
         2    0.023946
         3    0.758363
         4    0.023946
         5    0.862580
         dtype: float64
```

Suppose you need the fill value to vary by group. As you may guess, you need only group the data and use `apply` with a function that calls `fillna` on each data chunk. Here is some sample data on some US states divided into eastern and western states.

```
In [59]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
                   'Oregon', 'Nevada', 'California', 'Idaho']
         group_key = ['East'] * 4 + ['West'] * 4
         data = Series(np.random.randn(8), index=states)
         data[['Vermont', 'Nevada', 'Idaho']] = np.nan
         data
```

```
Out[59]: Ohio         -0.010032
         New York      0.050009
         Vermont            NaN
         Florida       0.852965
         Oregon       -0.955869
         Nevada             NaN
         California   -2.304234
         Idaho              NaN
         dtype: float64
```

```
In [60]: data.groupby(group_key).mean()
```

```
Out[60]: East    0.297647
         West   -1.630051
         dtype: float64
```

We can fill the NA values using the group means:

```
In [61]: fill_mean = lambda g: g.fillna(g.mean())
         data.groupby(group_key).apply(fill_mean)
```

```
Out[61]: Ohio         -0.010032
         New York      0.050009
         Vermont       0.297647
         Florida       0.852965
         Oregon       -0.955869
         Nevada       -1.630051
         California   -2.304234
         Idaho        -1.630051
         dtype: float64
```

In another case, you might have pre-defined fill values in your code that vary by group. Since the groups have a `name` attribute set, internally, we can use that:

```
In [62]: fill_values = {'East': 0.5, 'West': -1}
         fill_func = lambda g: g.fillna(fill_values[g.name])

         data.groupby(group_key).apply(fill_func)

Out[62]: Ohio         -0.010032
         New York      0.050009
         Vermont       0.500000
         Florida       0.852965
         Oregon       -0.955869
         Nevada       -1.000000
         California   -2.304234
         Idaho        -1.000000
         dtype: float64
```

1.3.3 Example: Random sampling and permutation

Suppose you wanted to draw a random sample from a large dataset for Monte Carlo simulation purposes or some other application. There are a number of ways to perform the "draws"; some are much more efficient than others. One way is to select the first `K` elements of `np.random.permutation(N)`, where `N` is the size of your complete dataset and `K` the desired sample size. As a more fun example, here's a way to construct a deck of English-style playing cards:

```
In [63]: # Hearts, Spades, Clubs, Diamonds
         suits = ['H', 'S', 'C', 'D']
         card_val = (range(1, 11) + [10] * 3) * 4
         base_names = ['A'] + range(2, 11) + ['J', 'K', 'Q']
         cards = []
         for suit in ['H', 'S', 'C', 'D']:
             cards.extend(str(num) + suit for num in base_names)

         deck = Series(card_val, index=cards)
```

So now we have a `Series` of length 52 whose index contains card names and values are the ones used in blackjack and other games (to keep things simple, I just let the ace be 1).

```
In [64]: deck[:13]

Out[64]: AH      1
         2H      2
         3H      3
         4H      4
         5H      5
         6H      6
         7H      7
         8H      8
         9H      9
         10H    10
         JH     10
         KH     10
         QH     10
         dtype: int64
```

Now, based on what we've just discussed, drawing a hand of five cards from the desk could be written as:

```
In [65]: def draw(deck, n=5):
             return deck.take(np.random.permutation(len(deck))[:n])
         draw(deck)
```

```
Out[65]: QC    10
         4H     4
         KC    10
         5H     5
         8C     8
         dtype: int64
```

Suppose you wnted two random cards from each suit. Because the suit is the last character of each card name, we can group based on this and use `apply`:

```
In [66]: get_suit = lambda card: card[-1] # last letter is suit
         deck.groupby(get_suit).apply(draw, n=2)
```

```
Out[66]: C  8C      8
            9C      9
         D  KD     10
            6D      6
         H  10H    10
            7H      7
         S  3S      3
            4S      4
         dtype: int64
```

```
In [67]: # alternatively
         deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
```

```
Out[67]: 4C      4
         10C    10
         AD      1
         4D      4
         JH     10
         7H      7
         KS     10
         AS      1
         dtype: int64
```

1.3.4 Example: Group weighted average and correlation

Under the split-apply-combine paradigm of `groupby` operations between columns in a `DataFrame` or two `Series`, such as group weighted average, become a routine affair. As an example, take this dataset containing group keys, values, and some weights:

```
In [68]: df = DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
                         'data': np.random.randn(8),
                         'weights': np.random.rand(8)})
         df
```

```
Out[68]:   category      data   weights
         0        a -1.218302  0.832619
```

```
1         a -0.733233  0.371132
2         a -0.851944  0.040553
3         a -1.623093  0.554671
4         b -0.279937  0.451246
5         b  1.155034  0.725301
6         b  0.227328  0.378451
7         b -1.095511  0.840662
```

The group weighted average by category would then be:

```
In [69]: grouped = df.groupby('category')
         get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
         grouped.apply(get_wavg)

Out[69]: category
         a           -1.23478
         b           -0.05155
         dtype: float64
```

As a less trivial example, consider a data set from Yahoo! Finance containing end of day prices for a few stocks and the S&P 500 index (the `SPX` ticker):

```
In [70]: close_px = pd.read_csv('stock_px.csv', parse_dates=True, index_col=0)
         close_px.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM     2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB
```

One task of interest might be to compute a `DataFrame` consisting of the yearly correlations of daily returns (computed from percent changes) with `SPX`. Here is one way to do it:

```
In [71]: close_px[-4:]

Out[71]:               AAPL   MSFT    XOM      SPX
         2011-10-11  400.29  27.00  76.27  1195.54
         2011-10-12  402.19  26.96  77.16  1207.25
         2011-10-13  408.43  27.18  76.37  1203.66
         2011-10-14  422.00  27.27  78.11  1224.58

In [72]: rets = close_px.pct_change().dropna()
         spx_corr = lambda x: x.corrwith(x['SPX'])
         by_year = rets.groupby(lambda x: x.year)
         by_year.apply(spx_corr)

Out[72]:           AAPL      MSFT       XOM  SPX
         2003  0.541124  0.745174  0.661265    1
         2004  0.374283  0.588531  0.557742    1
         2005  0.467540  0.562374  0.631010    1
         2006  0.428267  0.406126  0.518514    1
```

```
2007  0.508118  0.658770  0.786264    1
2008  0.681434  0.804626  0.828303    1
2009  0.707103  0.654902  0.797921    1
2010  0.710105  0.730118  0.839057    1
2011  0.691931  0.800996  0.859975    1
```

There is of course nothing to stop you from computing inter-column correlation:

```
In [73]: # Annual correlation of Apple with Microsoft
         by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
```

```
Out[73]: 2003    0.480868
         2004    0.259024
         2005    0.300093
         2006    0.161735
         2007    0.417738
         2008    0.611901
         2009    0.432738
         2010    0.571946
         2011    0.581987
         dtype: float64
```

1.4 Pivot tables and Cross-tabulation

A *pivot table* is a data summerization tool. Pivot tables work by aggregating a table of data by keys, where the data is organized rectangularly with the group keys along the rows and columns. We can use pivot tables in Python by using the `groupby` methodology. Using DataFrames allows us to apply the `pivot_table` method and we can use the `pandas.pivot_table` function. Besides acting as a great way to access `groupby`, `pivot_table` can add partial totals or margins.

We can use the `pivot_table` to calculate the group means of people by `sex` and `smoker`.

```
In [74]: tips=pd.read_csv('tips.csv')
         tips['tip_pct'] = tips['tip']/tips['total_bill']

         tips.pivot_table(index=['sex', 'smoker'])
```

```
Out[74]:                   size       tip   tip_pct  total_bill
         sex    smoker
         Female No      2.592593  2.773519  0.156921   18.105185
                Yes     2.242424  2.931515  0.182150   17.977879
         Male   No      2.711340  3.113402  0.160669   19.791237
                Yes     2.500000  3.051167  0.152771   22.284500
```

Suppose that now we only care about tip percentage, size of the group, and day of the week. We can put `smoker` in the columns and `day` in the rows, so that we yield a tables showing the group averages of `tip_pct` and `size` based on `smoker` and `day`.

```
In [75]: tips.pivot_table(['tip_pct','size'], index=['sex', 'day'],
                          columns='smoker')
```

```
Out[75]:              tip_pct                 size
         smoker            No       Yes         No        Yes
         sex    day
         Female Fri   0.165296  0.209129  2.500000  2.000000
                Sat   0.147993  0.163817  2.307692  2.200000
                Sun   0.165710  0.237075  3.071429  2.500000
```

```
        Thur  0.155971  0.163073  2.480000  2.428571
Male    Fri   0.138005  0.144730  2.000000  2.125000
        Sat   0.162132  0.139067  2.656250  2.629630
        Sun   0.158291  0.173964  2.883721  2.600000
        Thur  0.165706  0.164417  2.500000  2.300000
```

Furthermore, if we also want data about general tipping percentages and size of parties without regard to people smoking, we can use the `margins` argument to calculate to corresponding group statistic. Using `margins=True` calculates the partial totals of each column.

```
In [76]: tips.pivot_table(['tip_pct', 'size'], index=['sex', 'day'],
                          columns='smoker', margins=True)
```

```
Out[76]:               tip_pct                           size
         smoker             No       Yes       All         No       Yes       All
         sex    day
         Female Fri   0.165296  0.209129  0.199388  2.500000  2.000000  2.111111
                Sat   0.147993  0.163817  0.156470  2.307692  2.200000  2.250000
                Sun   0.165710  0.237075  0.181569  3.071429  2.500000  2.944444
                Thur  0.155971  0.163073  0.157525  2.480000  2.428571  2.468750
         Male   Fri   0.138005  0.144730  0.143385  2.000000  2.125000  2.100000
                Sat   0.162132  0.139067  0.151577  2.656250  2.629630  2.644068
                Sun   0.158291  0.173964  0.162344  2.883721  2.600000  2.810345
                Thur  0.165706  0.164417  0.165276  2.500000  2.300000  2.433333
         All          0.159328  0.163196  0.160803  2.668874  2.408602  2.569672
```

The `All` columns show the average tipping percentage and size of parties without regard to smoking. To use a different aggregate function, we may use the `aggfunc` argument. For example we may use the `len` function to calculate the frequency of group sizes.

```
In [77]: tips.pivot_table('tip_pct', index=['sex', 'smoker'], columns='day',
                          aggfunc=len, margins=True)
```

```
Out[77]: day            Fri  Sat  Sun  Thur  All
         sex    smoker
         Female No        2   13   14    25   54
                Yes       7   15    4     7   33
         Male   No        2   32   43    20   97
                Yes       8   27   15    10   60
         All             19   87   76    62  244
```

To replace empty values with zero, we can use the `fill_value` argument.

```
In [78]: tips.pivot_table('size', index=['time', 'sex', 'smoker'],
                          columns='day', aggfunc='sum',fill_value=0)
```

```
Out[78]: day                  Fri  Sat  Sun  Thur
         time   sex    smoker
         Dinner Female No       2   30   43     2
                       Yes      8   33   10     0
                Male   No       4   85  124     0
                       Yes     12   71   39     0
         Lunch  Female No       3    0    0    60
                       Yes      6    0    0    17
                Male   No       0    0    0    50
                       Yes      5    0    0    23
```

1.4.1 Cross-tabulations: crosstab

A *cross-tabulation* is a special case of a pivot table that computes group frequencies.

In [79]:
```
from StringIO import StringIO
data = """\
Sample    Gender    Handedness
1    Female    Right-handed
2    Male    Left-handed
3    Female    Right-handed
4    Male    Right-handed
5    Male    Left-handed
6    Male    Right-handed
7    Female    Right-handed
8    Female    Left-handed
9    Male    Right-handed
10    Female    Right-handed"""
data = pd.read_table(StringIO(data), sep='\s+')

data
```

Out[79]:
```
   Sample  Gender    Handedness
0       1  Female  Right-handed
1       2    Male   Left-handed
2       3  Female  Right-handed
3       4    Male  Right-handed
4       5    Male   Left-handed
5       6    Male  Right-handed
6       7  Female  Right-handed
7       8  Female   Left-handed
8       9    Male  Right-handed
9      10  Female  Right-handed
```

We could use pivot_table to do this calculation, but pandas.crosstab is a convenient.

In [80]:
```
pd.crosstab(data.Gender, data.Handedness, margins=True)
```

Out[80]:
```
Handedness  Left-handed  Right-handed  All
Gender
Female                1             4    5
Male                  2             3    5
All                   3             7   10
```

When using crosstab, we may use either an array or Series or a list of arrays.

In [81]:
```
pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
```

Out[81]:
```
smoker        No  Yes  All
time   day
Dinner Fri     3    9   12
       Sat    45   42   87
       Sun    57   19   76
       Thur    1    0    1
Lunch  Fri     1    6    7
       Thur   44   17   61
All          151   93  244
```

1.5 Example: 2012 Federal Election Commission Database

We will be working with data from the 2012 US Presidential Election. This dataset focuses on campaign contributions for presidential candidates. The data can be loaded from:

```
In [82]: fec = pd.read_csv('P00000001-ALL.csv')

         fec.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id              1001731 non-null object
cand_id              1001731 non-null object
cand_nm              1001731 non-null object
contbr_nm            1001731 non-null object
contbr_city          1001712 non-null object
contbr_st            1001727 non-null object
contbr_zip           1001620 non-null object
contbr_employer      988002 non-null object
contbr_occupation    993301 non-null object
contb_receipt_amt    1001731 non-null float64
contb_receipt_dt     1001731 non-null object
receipt_desc         14166 non-null object
memo_cd              92482 non-null object
memo_text            97770 non-null object
form_tp              1001731 non-null object
file_num             1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 129.9+ MB

/home/alethiometryst/anaconda/lib/python2.7/site-packages/pandas/io/parsers.py:1159: DtypeWarning: Colu
  data = self._reader.read(nrows)
```

A sample data frame looks like this:

```
In [83]: fec.ix[123456]

Out[83]: cmte_id                            C00431445
         cand_id                            P80003338
         cand_nm                        Obama, Barack
         contbr_nm                        ELLMAN, IRA
         contbr_city                            TEMPE
         contbr_st                                 AZ
         contbr_zip                         852816719
         contbr_employer      ARIZONA STATE UNIVERSITY
         contbr_occupation                  PROFESSOR
         contb_receipt_amt                         50
         contb_receipt_dt                   01-DEC-11
         receipt_desc                             NaN
         memo_cd                                  NaN
         memo_text                                NaN
         form_tp                                SA17A
         file_num                              772372
         Name: 123456, dtype: object
```

One interesting aspect of this data set is the lack of partisanship as a way to classify candidates. We can add this information to the dataset. The way we are going to solve this problem is to create a dictionary indicating the politcal party of each candidate. First, we need to find out who all of the candidates are.

```
In [84]: unique_cands = fec.cand_nm.unique()
         unique_cands

Out[84]: array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
                "Roemer, Charles E. 'Buddy' III", 'Pawlenty, Timothy',
                'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, Herman',
                'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon',
                'Perry, Rick'], dtype=object)

In [85]: unique_cands[2]

Out[85]: 'Obama, Barack'
```

We use `parties` to specify a dictionary over all of the candidates.

```
In [86]: parties = {'Bachmann, Michelle': 'Republican',
                    'Cain, Herman': 'Republican',
                    'Gingrich, Newt': 'Republican',
                    'Huntsman, Jon': 'Republican',
                    'Johnson, Gary Earl': 'Republican',
                    'McCotter, Thaddeus G': 'Republican',
                    'Obama, Barack': 'Democrat',
                    'Paul, Ron': 'Republican',
                    'Pawlenty, Timothy': 'Republican',
                    'Perry, Rick': 'Republican',
                    "Roemer, Charles E. 'Buddy' III": 'Republican',
                    'Romney, Mitt': 'Republican',
                    'Santorum, Rick': 'Republican'}
```

We can test our dictionary by viewing a section of the dataset to first view the candidate and then view their political affiliation.

```
In [87]: fec.cand_nm[123456:123461]

Out[87]: 123456    Obama, Barack
         123457    Obama, Barack
         123458    Obama, Barack
         123459    Obama, Barack
         123460    Obama, Barack
         Name: cand_nm, dtype: object

In [88]: fec.cand_nm[123456:123461].map(parties)

Out[88]: 123456    Democrat
         123457    Democrat
         123458    Democrat
         123459    Democrat
         123460    Democrat
         Name: cand_nm, dtype: object
```

To calculate the number of contributions to each party, we use the `value_counts` function to sum the number of contributions of each party.

```
In [89]: # Add it as a column

         fec['party'] = fec.cand_nm.map(parties)

         fec['party'].value_counts()

Out[89]: Democrat      593746
         Republican    407985
         dtype: int64
```

Unfortunately, this counts both the positive and negative contributions to candidate's campaigns (negative values indcate refunds). Thus, to see the total number of donations to candidates in the 2012 US election we subset the receipt values.

```
In [90]: (fec.contb_receipt_amt > 0).value_counts()

Out[90]: True     991475
         False     10256
         dtype: int64
```

To just use positive contibutions we use the following code.

```
In [91]: fec = fec[fec.contb_receipt_amt > 0]

         fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

1.5.1 Donation statistics by occupation and employer

One interesting question is the occupation of donors for each party. For example, do lawyers donate more to Democrats or Republicans? To which party do business executives donate more money?

```
In [92]: fec.contbr_occupation.value_counts()[:10]

Out[92]: RETIRED                                   233990
         INFORMATION REQUESTED                      35107
         ATTORNEY                                   34286
         HOMEMAKER                                  29931
         PHYSICIAN                                  23432
         INFORMATION REQUESTED PER BEST EFFORTS     21138
         ENGINEER                                   14334
         TEACHER                                    13990
         CONSULTANT                                 13273
         PROFESSOR                                  12555
         dtype: int64
```

We can again use a dictionary to better define the occupation of the donors, as well as the employers of the donors.

```
In [93]: occ_mapping = {
            'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
            'INFORMATION REQUESTED' : 'NOT PROVIDED',
            'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
            'C.E.O.': 'CEO'
         }
```

```
In [94]: # If no mapping provided, return x
         f = lambda x: occ_mapping.get(x, x)
         fec.contbr_occupation = fec.contbr_occupation.map(f)

         emp_mapping = {
            'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
            'INFORMATION REQUESTED' : 'NOT PROVIDED',
            'SELF' : 'SELF-EMPLOYED',
            'SELF EMPLOYED' : 'SELF-EMPLOYED',
         }
```

```
In [95]: # If no mapping provided, return x
         f = lambda x: emp_mapping.get(x, x)
         fec.contbr_employer = fec.contbr_employer.map(f)
```

Using a `pivot_table` we can view data on people who donated at least $2 million.

```
In [96]: by_occupation = fec.pivot_table('contb_receipt_amt',
                                         index='contbr_occupation',
                                         columns='party', aggfunc='sum')

         over_2mm = by_occupation[by_occupation.sum(1) > 2000000]
         over_2mm
```

```
Out[96]: party                  Democrat      Republican
         contbr_occupation
         ATTORNEY            11141982.97   7477194.430000
         CEO                  2074974.79   4211040.520000
         CONSULTANT           2459912.71   2544725.450000
         ENGINEER              951525.55   1818373.700000
         EXECUTIVE            1355161.05   4138850.090000
         HOMEMAKER            4248875.80  13634275.780000
         INVESTOR              884133.00   2431768.920000
         LAWYER               3160478.87    391224.320000
         MANAGER               762883.22   1444532.370000
         NOT PROVIDED         4866973.96  20565473.010000
         OWNER                1001567.36   2408286.920000
         PHYSICIAN            3735124.94   3594320.240000
         PRESIDENT            1878509.95   4720923.760000
         PROFESSOR            2165071.08    296702.730000
         REAL ESTATE           528902.09   1625902.250000
         RETIRED             25305116.38  23561244.489999
         SELF-EMPLOYED         672393.40   1640252.540000
```

```
In [97]: over_2mm.plot(kind='barh')
```

```
Out[97]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37d8e6db90>
```

Alternatively, we can view donors who gave to the campaigns of Barack Obama or Mitt Romney. We do this by grouping by candidate name using the `top` method that we learned earlier.

```
In [98]: def get_top_amounts(group, key, n=5):
             totals = group.groupby(key)['contb_receipt_amt'].sum()

             # Order totals by key in descending order
             return totals.order(ascending=False)[-n:]

         grouped = fec_mrbo.groupby('cand_nm')
         grouped.apply(get_top_amounts, 'contbr_occupation', n=7)

         grouped.apply(get_top_amounts, 'contbr_employer', n=10)
```

```
Out[98]: cand_nm        contbr_employer
         Obama, Barack  SOLIYA                                3.0
                        CARR ENTERPRISES                      3.0
                        PENN STATE DICKINSON SCHOOL OF LAW    3.0
                        CADUCEUS OCCUPATIONAL MEDICINE        3.0
                        N.A.                                  3.0
                        REAL ENERGY CONSULTING SERVICES       3.0
                        JPDSYSTEMS, LLC                       3.0
                        CASS REGIONAL MED. CENTER             2.5
                        ARCON CORP                            2.0
                        THE VICTORIA GROUP, INC.              2.0
         Romney, Mitt   EASTHAM CAPITAL                       5.0
                        GREGORY GALLIVAN                      5.0
                        DIRECT LENDERS LLC                    5.0
                        LOUGH INVESTMENT ADVISORY LLC         4.0
                        WATERWORKS INDUSRTIES                 3.0
                        WILL MERRIFIELD                       3.0
                        HONOLD COMMUNICTAIONS                 3.0
                        INDEPENDENT PROFESSIONAL              3.0
                        UPTOWN CHEAPSKATE                     3.0
                        UN                                    3.0
         Name: contb_receipt_amt, dtype: float64
```

1.5.2 Bucketing donation amounts

A useful way to analyze data is to use the `cut` function to partition the data into comparable buckets.

```
In [99]: bins = np.array([0, 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000])
         labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)
         labels
```

```
Out[99]: 411      (10, 100]
         412    (100, 1000]
         413    (100, 1000]
         414      (10, 100]
         415      (10, 100]
         416      (10, 100]
         417    (100, 1000]
         418      (10, 100]
         419    (100, 1000]
         420      (10, 100]
         421      (10, 100]
         422    (100, 1000]
         423    (100, 1000]
```

```
424    (100, 1000]
425    (100, 1000]
...
701371        (10, 100]
701372        (10, 100]
701373        (10, 100]
701374        (10, 100]
701375        (10, 100]
701376    (1000, 10000]
701377        (10, 100]
701378        (10, 100]
701379      (100, 1000]
701380    (1000, 10000]
701381        (10, 100]
701382      (100, 1000]
701383          (1, 10]
701384        (10, 100]
701385      (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, object): [(0, 1] < (1, 10] < (10, 100] < (100, 1000] < (1000, 10000] < (10000, 
```

Grouping the data by name and bin, we get a histogram by donation size.

In [100]:
```python
grouped = fec_mrbo.groupby(['cand_nm', labels])
grouped.size().unstack(0)
```

Out[100]:
```
cand_nm                Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                           493            77
(1, 10]                        40070          3681
(10, 100]                     372280         31853
(100, 1000]                   153991         43357
(1000, 10000]                  22284         26186
(10000, 100000]                    2             1
(100000, 1000000]                  3           NaN
(1000000, 10000000]                4           NaN
```

The data shows that Barack Obama received significantly more contributions of smaller donation sizes. We can also sum the contribution amounts and normalize the data to view a percentage of total donations of each size by candidate:

In [101]:
```python
bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)
bucket_sums
```

Out[101]:
```
cand_nm                Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                        318.24         77.00
(1, 10]                    337267.62      29819.66
(10, 100]                20288981.41    1987783.76
(100, 1000]              54798531.46   22363381.69
(1000, 10000]            51753705.67   63942145.42
(10000, 100000]             59100.00      12700.00
(100000, 1000000]         1490683.08           NaN
(1000000, 10000000]       7148839.76           NaN
```

In [102]:
```python
normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)
normed_sums
```

```
Out[102]: cand_nm               Obama, Barack  Romney, Mitt
          contb_receipt_amt
          (0, 1]                     0.805182      0.194818
          (1, 10]                    0.918767      0.081233
          (10, 100]                  0.910769      0.089231
          (100, 1000]                0.710176      0.289824
          (1000, 10000]              0.447326      0.552674
          (10000, 100000]            0.823120      0.176880
          (100000, 1000000]          1.000000           NaN
          (1000000, 10000000]        1.000000           NaN
```

```
In [103]: normed_sums[:-2].plot(kind='barh', stacked=True)
```

```
Out[103]: <matplotlib.axes._subplots.AxesSubplot at 0x7f38001d0b90>
```

1.5.3 Donation statistics by state

We can also aggregate donations by candidate and state:

```
In [104]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])
          totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)
          totals = totals[totals.sum(1) > 100000]
          totals[:10]
```

```
Out[104]: cand_nm    Obama, Barack  Romney, Mitt
          contbr_st
          AK             281840.15      86204.24
          AL             543123.48     527303.51
          AR             359247.28     105556.00
          AZ            1506476.98    1888436.23
          CA           23824984.24   11237636.60
          CO            2132429.49    1506714.12
          CT            2068291.26    3499475.45
          DC            4373538.80    1025137.50
          DE             336669.14      82712.00
          FL            7318178.58    8338458.81
```

Additionally, we may obtain the relative percentage of total donations by state for each candidate.

```
In [105]: percent = totals.div(totals.sum(1), axis=0)
          percent[:10]
```

```
Out[105]: cand_nm    Obama, Barack  Romney, Mitt
          contbr_st
          AK              0.765778      0.234222
          AL              0.507390      0.492610
          AR              0.772902      0.227098
          AZ              0.443745      0.556255
          CA              0.679498      0.320502
          CO              0.585970      0.414030
          CT              0.371476      0.628524
          DC              0.810113      0.189887
          DE              0.802776      0.197224
          FL              0.467417      0.532583
```