Data Aggregation and Group Operations

Part 2

GroupBy Mechanics

Part 2

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 = pd.DataFrame(np.random.randn(5, 5),
                                  columns=['a', 'b', 'c', 'd', 'e'],
                                  index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
          people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values
          people
Out[20]:
                 1.007189 -1.296221
                                    0.274992
                                             0.228913 1.352917
                 0.886429 -2.001637 -0.371843
                                                      -0.438570
                 -0.539741
                              NaN
                                        NaN -1.021228 -0.577087
                          0.302614
                                    0.523772
                                                      1.343810
           Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

 Now, suppose we have a group correspondence for the columns and want to sum together the columns by group:

 Now, you could construct an array from this dict to pass to groupby, but instead we can just pass the dict (the key 'f' is included to highlight that unused grouping keys are OK):



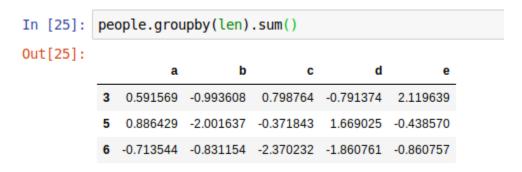
• The same functionality holds for Series, which can be viewed as a fixed-size mapping:

```
map series = pd.Series(mapping)
In [23]:
         map series
Out[23]: a
                  red
                  red
                 blue
                 blue
                  red
              orange
         dtype: object
         people.groupby(map series, axis=1).count()
Out[24]:
                blue red
            Joe
           Steve
           Wes
            Jim
          Travis
```

Grouping with Functions

- Using Python functions is a more generic way of defining a group mapping compared with a dict or Series.
- Any function passed as a group key will be called once per index value, with the return values being used as the group names.

- More concretely, consider the example DataFrame from the previous section, which has people's first names as index values.
- Suppose you wanted to group by the length of the names; while you could compute an array of string lengths, it's simpler to just pass the len function:



 Mixing functions with arrays, dicts, or Series is not a problem as everything gets converted to arrays internally:

Grouping by Index Levels

- A final convenience for hierarchically indexed datasets is the ability to aggregate using one of the levels of an axis index.
- Let's look at an example:

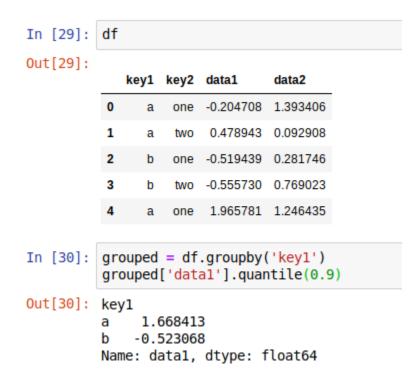
• To group by level, pass the level number or name using the level keyword:

Data Aggregation

Optimized groupby methods

Function name	Description
count	Number of non-NA values in the group
sum	Sum of non-NA values
mean	Mean of non-NA values
median	Arithmetic median of non-NA values
std, var	Unbiased (n – 1 denominator) standard deviation and variance
min, max	Minimum and maximum of non-NA values
prod	Product of non-NA values
first, last	First and last non-NA values

- You can use aggregations of your own devising and additionally call any method that is also defined on the grouped object.
- For example, you might recall that quantile computes sample quantiles of a Series or a DataFrame's columns.



• To use your own aggregation functions, pass any function that aggregates an array to the aggregate or agg method:

• You may notice that some methods like describe also work, even though they are not aggregations, strictly speaking:

In [33]:	group	ed.des	scribe()													
Out[33]:		data1								data2						
		count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%
	key1															
	a	3.0	0.746672	1.109736	-0.204708	0.137118	0.478943	1.222362	1.965781	3.0	0.910916	0.712217	0.092908	0.669671	1.246435	1.319920
	b	2.0	-0.537585	0.025662	-0.555730	-0.546657	-0.537585	-0.528512	-0.519439	2.0	0.525384	0.344556	0.281746	0.403565	0.525384	0.64720

Column-Wise and Multiple Function Application

- Let's return to the tipping dataset from earlier examples.
- After loading it with read_csv, we add a tipping percentage column tip pct:

```
In [34]: tips = pd.read csv('examples/tips.csv')
          # Add tip percentage of total bill
          tips['tip pct'] = tips['tip'] / tips['total bill']
          tips[:6]
Out[34]:
                      tip smoker day
             total bill
                                        time size
                                                    tip_pct
               16.99 1.01
                                               2 0.059447
                              No Sun Dinner
               10.34 1.66
                              No Sun Dinner
                                                3 0.160542
               21.01 3.50
                              No Sun Dinner
                                                3 0.166587
               23.68 3.31
                              No Sun Dinner
                                                2 0.139780
               24.59 3.61
                              No Sun Dinner
                                                4 0.146808
                25.29 4.71
                              No Sun Dinner
                                                4 0.186240
```

- As you've already 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.
- Fortunately, this is possible to do.

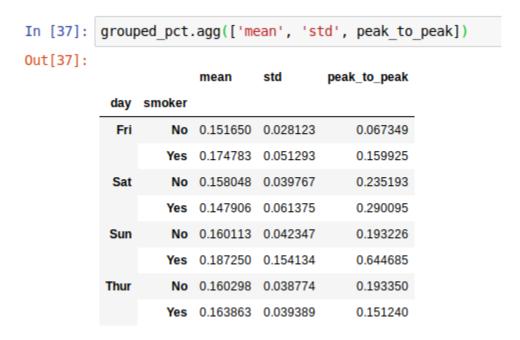
• First, we'll group the tips by day and smoker:

```
In [35]: grouped = tips.groupby(['day', 'smoker'])
```

 Note that for descriptive statistics, you can pass the name of the function as a string:

```
In [36]: grouped pct = grouped['tip pct']
         grouped pct.agg('mean')
Out[36]: day
               smoker
         Fri
               No
                         0.151650
               Yes
                         0.174783
               No
                         0.158048
         Sat
               Yes
                         0.147906
         Sun
                         0.160113
               Yes
                         0.187250
              No
                         0.160298
         Thur
               Yes
                         0.163863
         Name: tip pct, dtype: float64
```

• If you pass a list of functions or function names instead, you get back a DataFrame with column names taken from the functions:



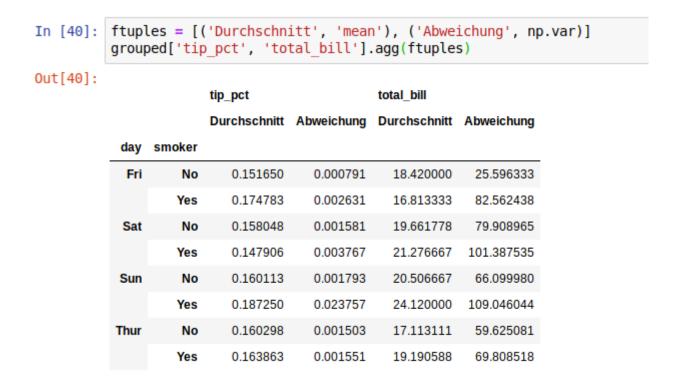
- You don't need to accept the names that GroupBy gives to the columns; notably, lambda functions have the name '<lambda>', which makes them hard to identify (you can see for yourself by looking at a function's name attribute).
- Thus, if you pass a list of (name, function) tuples, the first element of each tuple will be used as the DataFrame column names (you can think of a list of 2-tuples as an ordered mapping):



- With a DataFrame you have more options, as you can specify a list of functions to apply to all of the columns or different functions per column.
- To start, suppose we wanted to compute the same three statistics for the tip pct and total bill columns:

```
In [39]: functions = ['count', 'mean', 'max']
          result = grouped['tip pct', 'total bill'].agg(functions)
          result
Out[39]:
                                                total bill
                        tip_pct
                        count mean
                                       max
                                                count mean
                                                                max
            day smoker
            Fri
                            4 0.151650 0.187735
                                                    4 18.420000 22.75
                    No
                           15 0.174783 0.263480
                    Yes
                                                   15 16.813333 40.17
                                                   45 19.661778 48.33
            Sat
                           45 0.158048 0.291990
                           42 0.147906 0.325733
                                                   42 21.276667 50.81
                    Yes
            Sun
                           57 0.160113 0.252672
                                                   57 20.506667 48.17
                   Yes
                           19 0.187250 0.710345
                                                   19 24.120000 45.35
                          45 0.160298 0.266312
                                                   45 17.113111 41.19
           Thur
                                                  17 19.190588 43.11
                    Yes
                           17 0.163863 0.241255
```

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



- Now, suppose you wanted to apply potentially different functions to one or more of the columns.
- To do this, pass a dict to agg that contains a mapping of column names to any of the function specifications listed so far.

```
In [41]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[41]:
                       tip
                             size
           day smoker
                   No
                        3.50
                               9
            Fri
                   Yes
                        4.73
                              31
                        9.00 115
                   No
            Sat
                   Yes 10.00 104
                        6.00
                             167
           Sun
                   No
                        6.50
                   Yes
                              49
           Thur
                        6.70 112
                   Yes
                        5.00
                              40
In [42]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
                        'size' : 'sum'})
Out[42]:
                       tip_pct
                                                          size
                       min
                                max
                                         mean
                                                  std
                                                           sum
           day smoker
                    No 0.120385 0.187735 0.151650 0.028123
            Fri
                   Yes 0.103555 0.263480 0.174783 0.051293
           Sat
                   No 0.056797 0.291990 0.158048 0.039767
                                                          115
                   Yes 0.035638 0.325733 0.147906 0.061375 104
                    No 0.059447 0.252672 0.160113 0.042347 167
           Sun
                   Yes 0.065660 0.710345 0.187250 0.154134
                   No 0.072961 0.266312 0.160298 0.038774 112
           Thur
                   Yes 0.090014 0.241255 0.163863 0.039389
                                                            40
```

Returning Aggregated Data Without Row Indexes

- In all of the examples up until now, the aggregated data comes back with an index, potentially hierarchical, composed from the unique group key combinations.
- Since this isn't always desirable, you can disable this behavior in most cases by passing as index=False to groupby:

	<pre>tips.groupby(['day', 'smoker'], as_index=False)</pre>										
]:		day	smoker	total_bill	tip	size	tip_pct				
	0	Fri	No	18.420000	2.812500	2.250000	0.151650				
	1	Fri	Yes	16.813333	2.714000	2.066667	0.174783				
	2	Sat	No	19.661778	3.102889	2.555556	0.158048				
	3	Sat	Yes	21.276667	2.875476	2.476190	0.147906				
	4	Sun	No	20.506667	3.167895	2.929825	0.160113				
	5	Sun	Yes	24.120000	3.516842	2.578947	0.187250				
	6	Thur	No	17.113111	2.673778	2.488889	0.160298				
	7	Thur	Yes	19.190588	3.030000	2.352941	0.163863				