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]:

	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

- Now, 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'}
```

- 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):

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

```
In [23]: map_series = pd.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

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:

```
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

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

In [26]:
```python
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

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:

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

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

```
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

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.

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

Out[29]:

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

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

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

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

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

In [32]: grouped.agg(peak_to_peak)
Out[32]:
```

	data1	data2
key1		
a	2.170488	1.300498
b	0.036292	0.487276

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

```
In [33]: grouped.describe()
```

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	1.31992(
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]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808
5	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
         Sat   No        0.158048
               Yes       0.147906
         Sun   No        0.160113
               Yes       0.187250
         Thur  No        0.160298
               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:

```
In [37]: grouped_pct.agg(['mean', 'std', peak_to_peak])
Out[37]:
```

day	smoker	mean	std	peak_to_peak
Fri	No	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	No	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	No	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	No	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

- 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):

```
In [38]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[38]:
```

day	smoker	foo	bar
Fri	No	0.151650	0.028123
	Yes	0.174783	0.051293
Sat	No	0.158048	0.039767
	Yes	0.147906	0.061375
Sun	No	0.160113	0.042347
	Yes	0.187250	0.154134
Thur	No	0.160298	0.038774
	Yes	0.163863	0.039389

- 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]:

		tip_pct			total_bill		
		count	mean	max	count	mean	max
day	smoker						
Fri	No	4	0.151650	0.187735	4	18.420000	22.75
	Yes	15	0.174783	0.263480	15	16.813333	40.17
Sat	No	45	0.158048	0.291990	45	19.661778	48.33
	Yes	42	0.147906	0.325733	42	21.276667	50.81
Sun	No	57	0.160113	0.252672	57	20.506667	48.17
	Yes	19	0.187250	0.710345	19	24.120000	45.35
Thur	No	45	0.160298	0.266312	45	17.113111	41.19
	Yes	17	0.163863	0.241255	17	19.190588	43.11

- As before, 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
day	smoker				
Fri	No	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	No	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535
Sun	No	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	No	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

- 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]:

day	smoker	tip	size
Fri	No	3.50	9
	Yes	4.73	31
Sat	No	9.00	115
	Yes	10.00	104
Sun	No	6.00	167
	Yes	6.50	49
Thur	No	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					
Fri	No	0.120385	0.187735	0.151650	0.028123	9
	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	No	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	No	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	No	0.072961	0.266312	0.160298	0.038774	112
	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`:

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

	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