Data Aggregation and Group Operations

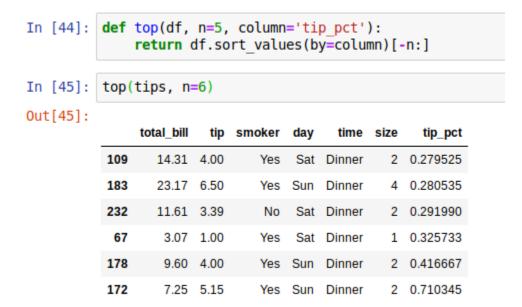
Part 3

Apply: General split-apply-combine

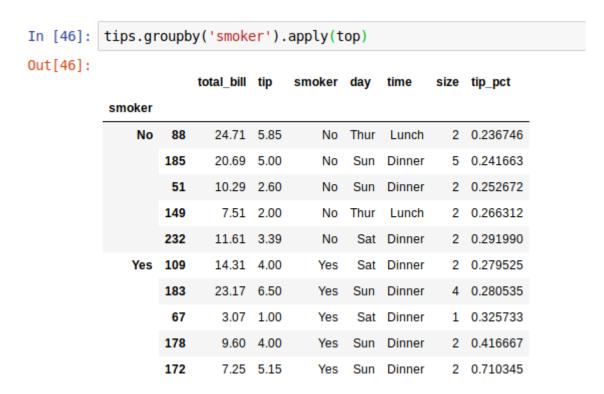
Part 1

• apply splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concatenate the pieces together.

- Returning to the tipping dataset from before, suppose you wanted to select the top five tip pct values by group.
- First, write a function that selects the rows with the largest values in a particular column:



 Now, if we group by smoker, say, and call apply with this function, we get the following:



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

In [47]:	tips.gr	oupby	/([':	smoker',	'day']).apply(top, n=1, column='tot					
Out[47]:										
				total_bill	tip	smoker	day	time	size	tip_pct
	smoker	day								
	No	Fri	94	22.75	3.25	No	Fri	Dinner	2	0.142857
		Sat	212	48.33	9.00	No	Sat	Dinner	4	0.186220
		Sun	156	48.17	5.00	No	Sun	Dinner	6	0.103799
		Thur	142	41.19	5.00	No	Thur	Lunch	5	0.121389
	Yes	Fri	95	40.17	4.73	Yes	Fri	Dinner	4	0.117750
		Sat	170	50.81	10.00	Yes	Sat	Dinner	3	0.196812
		Sun	182	45.35	3.50	Yes	Sun	Dinner	3	0.077178
		Thur	197	43.11	5.00	Yes	Thur	Lunch	4	0.115982

Suppressing the Group Keys

- In the preceding examples, you see that the resulting object has a hierarchical index formed from the group keys along with the indexes of each piece of the original object.
- You can disable this by passing group keys=False to groupby:

In [48]:	tips.groupby('smoker', group_keys=False).apply											
Out[48]:		total_bill	tip	smoker	day	time	size	tip_pct				
	88	24.71	5.85	No	Thur	Lunch	2	0.236746				
	185	20.69	5.00	No	Sun	Dinner	5	0.241663				
	51	10.29	2.60	No	Sun	Dinner	2	0.252672				
	149	7.51	2.00	No	Thur	Lunch	2	0.266312				
	232	11.61	3.39	No	Sat	Dinner	2	0.291990				
	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525				
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535				
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733				
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667				
	172	7.25	5.15	Yes	Sun	Dinner	2	0.710345				

Quantile and Bucket Analysis

- pandas has some tools, in particular cut and qcut, for slicing data up into buckets with bins of your choosing or by sample quantiles.
- Combining these functions with groupby makes it convenient to perform bucket or quantile analysis on a dataset.

 Consider a simple random dataset and an equal-length bucket categorization using cut:

```
In [49]: frame = pd.DataFrame({'data1': np.random.randn(1000),
                                'data2': np.random.randn(1000)})
         quartiles = pd.cut(frame.data1, 4)
         quartiles[:10]
Out[49]: 0
               (-1.23, 0.489]
              (-2.956, -1.23]
               (-1.23, 0.489]
               (0.489, 2.208]
               (-1.23, 0.489]
               (0.489, 2.208]
               (-1.23, 0.489]
               (-1.23, 0.489]
               (0.489, 2.208]
               (0.489, 2.208]
         Name: data1, dtype: category
         Categories (4, interval[float64]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.208] < (2.208, 3.928]]
```

- The Categorical object returned by cut can be passed directly to groupby.
- So we could compute a set of statistics for the data2 column like so:

```
In [50]: def get stats(group):
               return {'min': group.min(), 'max': group.max(),
                       'count': group.count(), 'mean': group.mean()}
In [51]: grouped = frame.data2.groupby(quartiles)
In [52]: grouped.apply(get stats).unstack()
Out[52]:
                       count max
                                      mean
                                               min
                 data1
                        95.0 1.670835 -0.039521 -3.399312
           (-2.956, -1.23]
           (-1.23, 0.489) 598.0 3.260383 -0.002051 -2.989741
           (0.489, 2.208) 297.0 2.954439 0.081822 -3.745356
                        10.0 1.765640 0.024750 -1.929776
           (2.208, 3.928)
```

- These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use qcut.
- We'll pass labels=False to just get quantile numbers:

```
In [53]: # Return quantile numbers
          grouping = pd.gcut(frame.data1, 10, labels=False)
          grouped = frame.data2.groupby(grouping)
          grouped.apply(get stats).unstack()
Out[53]:
                                         min
                 count max
                                mean
           data1
              0 100.0 1.670835 -0.049902 -3.399312
              1 100.0 2.628441
                                0.030989 -1.950098
              2 100.0 2.527939 -0.067179 -2.925113
              3 100.0 3.260383
                                0.065713 -2.315555
              4 100.0 2.074345 -0.111653 -2.047939
                       2.184810
                                0.052130
                                         -2.989741
              6 100.0 2.458842
                                -0.021489 -2.223506
              7 100.0 2.954439
                                -0.026459 -3.056990
              8 100.0 2.735527
                                0.103406 -3.745356
                                0.220122 -2.064111
              9 100.0 2.377020
```

Example: Filling Missing Values with Group-Specific Values

 When cleaning up missing data, in some cases you will replace data observations using dropna, but in others you may want to impute (fill in) the null (NA) values using a fixed value or some value derived from the data.

```
In [54]: s = pd.Series(np.random.randn(6))
         s[::2] = np.nan
Out[54]: 0
                    NaN
             -0.125921
                    NaN
              -0.884475
                    NaN
              0.227290
         dtype: float64
In [55]: s.fillna(s.mean())
Out[55]: 0
              -0.261035
            -0.125921
             -0.261035
             -0.884475
              -0.261035
         dtype: float64
```

- Suppose you need the fill value to vary by group.
- One way to do this is to group the data and use apply with a function that calls fillna on each data chunk.
- Here is some sample data on US states divided into eastern and western regions:

```
In [56]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
                    'Oregon', 'Nevada', 'California', 'Idaho']
         group key = ['East'] * 4 + ['West'] * 4
         data = pd.Series(np.random.randn(8), index=states)
         data
Out[56]: Ohio
                       0.922264
         New York
                       -2.153545
         Vermont
                       -0.365757
         Florida
                       -0.375842
         0regon
                       0.329939
         Nevada
                       0.981994
         California
                       1.105913
         Idaho
                       -1.613716
         dtype: float64
```

• Let's set some values in the data to be missing:

```
In [57]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan
         data
Out[57]: Ohio
                       0.922264
         New York
                      -2.153545
         Vermont
                            NaN
         Florida
                      -0.375842
                       0.329939
         0regon
         Nevada
                            NaN
         California
                       1.105913
         Idaho
                            NaN
         dtype: float64
In [58]: data.groupby(group_key).mean()
Out[58]: East
                -0.535707
         West
                 0.717926
         dtype: float64
```

• We can fill the NA values using the group means like so:

```
In [59]: fill_mean = lambda g: g.fillna(g.mean())
         data.groupby(group key).apply(fill mean)
Out[59]: Ohio
                       0.922264
         New York
                      -2.153545
                      -0.535707
         Vermont
         Florida
                      -0.375842
                       0.329939
         0regon
         Nevada
                       0.717926
         California
                       1.105913
         Idaho
                       0.717926
         dtype: float64
```

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

```
In [60]: 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[60]: Ohio
                       0.922264
         New York
                       -2.153545
         Vermont
                       0.500000
         Florida
                       -0.375842
         0regon
                       0.329939
         Nevada
                       -1.000000
         California
                       1.105913
         Idaho
                       -1.000000
         dtype: float64
```

Example: Random Sampling and Permutation

- Suppose you wanted to draw a random sample (with or without replacement) from a large dataset for Monte Carlo simulation purposes or some other application.
- There are a number of ways to perform the "draws"; here we use the sample method for Series.

 To demonstrate, here's a way to construct a deck of English-style playing 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, we just let the ace 'A' be 1):

• Drawing a hand of five cards from the deck could be written as:

```
In [63]: def draw(deck, n=5):
    return deck.sample(n)

In [64]: draw(deck)

Out[64]: AD     1
     8C     8
    5H     5
     KC     10
     2C     2
     dtype: int64
```

- Suppose you wanted 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:

• Alternatively, we could write:

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

Out[67]: KC    10
        JC    10
        AD     1
        5D    5
        5H    5
        6H    6
        7S    7
        KS    10
        dtype: int64
```