Chapter_10_Data_Aggregation_and_Group_Operations

March 7, 2024

Categorizing a dataset 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 dataset, you may need to compute group statistics or possibly pivot tables for reporting or visualization purposes. pandas provides a flexible groupby interface, enabling you to slice, dice, and summarize datasets in a natural way

One reason for the popularity of relational databases and SQL (which stands for "structured query language") is the ease with which data can be joined, filtered, trans- formed, and aggregated. However, query languages like SQL are somewhat con- strained in the kinds of group operations that can be performed. As you will see, with the expressiveness of Python and pandas, we can perform quite complex group oper- ations by utilizing any function that accepts a pandas object or NumPy array. In this chapter, you will learn how to:

- Split a pandas object into pieces using one or more keys (in the form of func- tions, arrays, or DataFrame column names)

- Calculate group summary statistics, like count, mean, or standard deviation, or a user-defined function

- Apply within-group transformations or other manipulations, like normalization, linear regression, rank, or subset selection

- Compute pivot tables and cross-tabulations

- Perform quantile analysis and other statistical group analyses

Aggregation of time series data, a special use case of groupby, is referred to as resampling in this book and will receive separate treatment in Chapter 11.

0.1 10.1 GroupBy Mechanics

Hadley Wickham, an author of many popular packages for the R programming lan- guage, coined the term split-apply-combine for describing group operations. In the first stage of the process, data contained in a pandas object, whether a Series, Data- Frame, or otherwise, is split into groups based on one or more keys that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1). Once this is done, a function is applied to each group, producing a new value. Finally, the results of all those function applications are combined into a result object. The form of the result- ing object will usually depend on what's being done to the data. See Figure 10-1 for a mockup of a simple group aggregation.

Each grouping key can take many forms, and the keys do not have to be all of the same type:

- A list or array of values that is the same length as the axis being grouped
- A value indicating a column name in a DataFrame

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

Note that the latter three methods are shortcuts for producing an array of values to be used to split up the object.

[3]:
```
import pandas as pd
import numpy as np
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],'key2' : ['one', 'two',␣
↪'one', 'two', 'one'],'data1' : np.random.randn(5),'data2' : np.random.
↪randn(5)})
```

[4]:
```
df
```

[4]:
```
  key1 key2     data1     data2
0    a  one  0.827214 -0.766832
1    a  two -0.542409  1.331633
2    b  one  0.011403 -1.302483
3    b  two  0.791102  1.625497
4    a  one  0.215320  0.775183
```

Suppose you wanted to compute the mean of the data1 column using the labels from key1. There are a number of ways to do this. One is to access data1 and call groupby with the column (a Series) at key1:

[5]:
```
grouped = df['data1'].groupby(df['key1'])
grouped
```

[5]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001E27E0C1B70>
```

This grouped variable is now a GroupBy object. It has not actually computed anything yet except for some intermediate data about the group key df['key1']. The idea is that this object has all of the information needed to then apply some operation to each of the groups. For example, to compute group means we can call the GroupBy's mean method:

[6]:
```
grouped.mean()
```

[6]:
```
key1
a    0.166708
b    0.401253
Name: data1, dtype: float64
```

Later, I'll explain more about what happens when you call .mean(). The important thing here is that the data (a Series) has been aggregated according to the group key, producing a new Series that is now indexed by the unique values in the key1 column

The result index has the name ‘key1’ because the DataFrame column df[‘key1’] did.

If instead we had passed multiple arrays as a list, we’d get something different:

[7]:
```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
```

[7]:
```
key1  key2
a     one     0.521267
      two    -0.542409
b     one     0.011403
      two     0.791102
Name: data1, dtype: float64
```

Here we grouped the data using two keys, and the resulting Series now has a hier- archical index consisting of the unique pairs of keys observed:

[8]:
```
means.unstack()
```

[8]:
```
key2       one       two
key1
a     0.521267 -0.542409
b     0.011403  0.791102
```

In this example, the group keys are all Series, though they could be any arrays of the right length:

[9]:
```
states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
years = np.array([2005, 2005, 2006, 2005, 2006])
df['data1'].groupby([states, years]).mean()
```

[9]:
```
California  2005   -0.542409
            2006    0.011403
Ohio        2005    0.809158
            2006    0.215320
Name: data1, dtype: float64
```

Frequently the grouping information is found in the same DataFrame as the data you want to work on. In that case, you can pass column names (whether those are strings, numbers, or other Python objects) as the group keys:

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

[10]:
```
         data1     data2
key1
a     0.166708  0.446661
b     0.401253  0.161507
```

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

[11]:
```
                data1     data2
key1 key2
a    one   0.521267  0.004175
     two  -0.542409  1.331633
b    one   0.011403 -1.302483
     two   0.791102  1.625497
```

You may have noticed in the first case df.groupby('key1').mean() that there is no key2 column in the result. Because df['key2'] is not numeric data, it is said to be a nuisance column, which is therefore excluded from the result. By default, all of the numeric columns are aggregated, though it is possible to filter down to a subset, as you'll see soon.

Regardless of the objective in using groupby, a generally useful GroupBy method is size, which returns a Series containing group sizes:

[12]:
```
df.groupby(['key1', 'key2']).size()
```

[12]:
```
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

Take note that any missing values in a group key will be excluded from the result.

0.1.1 Iterating Over Groups

The GroupBy object supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data. Consider the following:

[13]:
```
for name, group in df.groupby('key1'):
    print(name)
    print(group)
```

```
a
  key1 key2     data1     data2
0    a  one  0.827214 -0.766832
1    a  two -0.542409  1.331633
4    a  one  0.215320  0.775183
b
  key1 key2     data1     data2
2    b  one  0.011403 -1.302483
3    b  two  0.791102  1.625497
```

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

[14]:
```
for (k1, k2), group in df.groupby(['key1', 'key2']):
    print((k1, k2))
    print(group)
```

```
('a', 'one')
  key1 key2     data1     data2
0    a  one  0.827214 -0.766832
4    a  one  0.215320  0.775183
('a', 'two')
  key1 key2     data1     data2
1    a  two -0.542409  1.331633
('b', 'one')
  key1 key2     data1     data2
2    b  one  0.011403 -1.302483
('b', 'two')
  key1 key2     data1     data2
3    b  two  0.791102  1.625497
```

Of course, you can choose to do whatever you want with the pieces of data. A recipe you may find useful is computing a dict of the data pieces as a one-liner:

[15]:
```
pieces = dict(list(df.groupby('key1')))
pieces['b']
```

[15]:
```
  key1 key2     data1     data2
2    b  one  0.011403 -1.302483
3    b  two  0.791102  1.625497
```

By default groupby groups on axis=0, but you can group on any of the other axes. For example, we could group the columns of our example df here by dtype like so:

[16]:
```
df.dtypes
```

[16]:
```
key1      object
key2      object
data1    float64
data2    float64
dtype: object
```

[17]:
```
grouped = df.groupby(df.dtypes, axis=1)
```

We can print out the groups like so:

[18]:
```
for dtype, group in grouped:
    print(dtype)
    print(group)
```

```
float64
      data1     data2
0  0.827214 -0.766832
1 -0.542409  1.331633
2  0.011403 -1.302483
3  0.791102  1.625497
4  0.215320  0.775183
```

```
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

0.2 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 effect of column subsetting for aggregation. This means that:

[19]:
```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

[19]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001E27F213FD0>
```

are syntactic sugar for:

[20]:
```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

[20]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001E27EF4F208>
```

Especially for large datasets, it may be desirable to aggregate only a few columns. For example, in the preceding dataset, to compute means for just the data2 column and get the result as a DataFrame, we could write:

[21]:
```
df.groupby(['key1', 'key2'])[['data2']].mean()
```

[21]:
```
              data2
key1 key2
a    one   0.004175
     two   1.331633
b    one  -1.302483
     two   1.625497
```

The object returned by this indexing operation is a grouped DataFrame if a list or array is passed or a grouped Series if only a single column name is passed as a scalar:

[22]:
```
s_grouped = df.groupby(['key1', 'key2'])['data2']
s_grouped
```

[22]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001E27EF137F0>
```

[23]:
```
s_grouped.mean()
```

[23]:
```
key1  key2
a     one     0.004175
```

```
      two     1.331633
b     one    -1.302483
      two     1.625497
Name: data2, dtype: float64
```

0.3 Grouping with Dicts and Series

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

[24]:
```
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
```

[24]:
```
               a         b         c         d         e
Joe    -1.828972  0.019845 -0.068707 -0.340500 -0.338840
Steve   0.369133  0.787083  0.883550 -0.937417  1.704376
Wes     0.354648       NaN       NaN -0.594835  0.157820
Jim    -0.556810  0.459263 -1.985396 -0.037469  0.158352
Travis -0.249089 -0.317395 -0.157472  0.650558  0.009976
```

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

[25]:
```
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 (I included the key 'f' to highlight that unused grouping keys are OK):

[26]:
```
by_column = people.groupby(mapping, axis=1)
by_column.sum()
```

[26]:
```
            blue       red
Joe    -0.409207 -2.147967
Steve  -0.053867  2.860591
Wes    -0.594835  0.512468
Jim    -2.022865  0.060805
Travis  0.493086 -0.556507
```

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

[27]:
```
map_series = pd.Series(mapping)
map_series
```

[27]:
```
a        red
b        red
c       blue
```

```
d      blue
e       red
f    orange
dtype: object
```

[28]:
```
people.groupby(map_series, axis=1).count()
```

[28]:
```
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

0.4 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, con- sider 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:

[29]:
```
people.groupby(len).sum()
```

[29]:
```
          a         b         c         d         e
3 -2.031133  0.479107 -2.054103 -0.972805 -0.022668
5  0.369133  0.787083  0.883550 -0.937417  1.704376
6 -0.249089 -0.317395 -0.157472  0.650558  0.009976
```

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

[30]:
```
key_list = ['one', 'one', 'one', 'two', 'two']
```

[31]:
```
people.groupby([len, key_list]).min()
```

[31]:
```
              a         b         c         d         e
3 one -1.828972  0.019845 -0.068707 -0.594835 -0.338840
  two -0.556810  0.459263 -1.985396 -0.037469  0.158352
5 one  0.369133  0.787083  0.883550 -0.937417  1.704376
6 two -0.249089 -0.317395 -0.157472  0.650558  0.009976
```

0.4.1 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:

[32]:
```
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)
```

[33]:
```
hier_df
```

[33]:
```
cty          US                          JP
tenor         1         3         5         1         3
0     -0.343958  1.466424  1.511583  0.399633 -1.169836
1     -0.917280 -0.134536 -0.102389 -0.525327  1.341449
2     -1.214173  1.006286  0.405594 -0.490307  1.021310
3      1.957075 -0.567380 -0.009704  0.536272 -0.094344
```

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

[34]:
```
hier_df.groupby(level='cty', axis=1).count()
```

[34]:
```
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

0.5 10.2 Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. The preceding examples have used several of them, including mean, count, min, and sum. You may wonder what is going on when you invoke mean() on a GroupBy object. Many common aggregations, such as those found in Table 10-1, have optimized implementations. However, you are not limited to only this set of methods.

0.6 Table 10-1. 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.

While quantile is not explicitly implemented for GroupBy, it is a Series method and thus available for use. Internally, GroupBy efficiently slices up the Series, calls piece.quantile(0.9) for each piece, and then assembles those results together into the result object:

[35]:
```
df
```

[35]:
```
  key1 key2     data1     data2
0    a  one  0.827214 -0.766832
1    a  two -0.542409  1.331633
2    b  one  0.011403 -1.302483
3    b  two  0.791102  1.625497
4    a  one  0.215320  0.775183
```

[36]:
```
grouped = df.groupby('key1')
```

[37]:
```
grouped['data1'].quantile(0.9)
```

[37]:
```
key1
a    0.704835
b    0.713132
Name: data1, dtype: float64
```

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

[38]:
```
def peak_to_peak(arr):
    return arr.max() - arr.min()
```

[39]:
```
grouped.agg(peak_to_peak)
```

[39]:
```
         data1     data2
key1
a     1.369623  2.098465
b     0.779699  2.927980
```

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

[40]:
```
grouped.describe()
```

[40]:
```
     data1                                                            \
     count      mean       std       min       25%       50%       75%
key1
a      3.0  0.166708  0.686104 -0.542409 -0.163545  0.215320  0.521267
b      2.0  0.401253  0.551331  0.011403  0.206328  0.401253  0.596177
```

```
              data2                                                    \
                max count      mean       std       min       25%       50%
key1
a          0.827214   3.0  0.446661  1.087122 -0.766832  0.004175  0.775183
b          0.791102   2.0  0.161507  2.070395 -1.302483 -0.570488  0.161507


                75%       max
key1
a          1.053408  1.331633
b          0.893502  1.625497
```

Custom aggregation functions are generally much slower than the optimized functions found in Table 10-1. This is because there is some extra overhead (function calls, data rearrangement) in con- structing the intermediate group data chunks.

0.6.1 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:

[41]:
```
tips = pd.read_csv('tips.csv')
# Add tip percentage of total bill
tips['tip_pct'] = tips['tip'] / tips['total_bill']
tips[:6]
```

[41]:

	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

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, which I'll illustrate through a number of examples. First, I'll group the tips by day and smoker:

[42]:
```
grouped = tips.groupby(['day', 'smoker'])
```

Note that for descriptive statistics like those in Table 10-1, you can pass the name of the function as a string:

[43]:
```
grouped_pct = grouped['tip_pct']
grouped_pct.agg('mean')
```

[43]:
```
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:

[44]:
```
grouped_pct.agg(['mean', 'std', peak_to_peak])
```

[44]:
```
                 mean       std  peak_to_peak
day  smoker
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
```

Here we passed a list of aggregation functions to agg to evaluate indepedently on the data groups.

You don't need to accept the names that GroupBy gives to the columns; notably, lambda functions have the name ‘’, 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):

[45]:
```
grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
```

[45]:
```
                  foo       bar
day  smoker
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:

[46]:
```
functions = ['count', 'mean', 'max']
result = grouped['tip_pct', 'total_bill'].agg(functions)
result
```

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Pra
ctice_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-
packages\ipykernel_launcher.py:2: FutureWarning: Indexing with multiple keys
(implicitly converted to a tuple of keys) will be deprecated, use a list
instead.
```

[46]:

		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 you can see, the resulting DataFrame has hierarchical columns, the same as you would get aggregating each column separately and using concat to glue the results together using the column names as the keys argument:

[47]:
```
result['tip_pct']
```

[47]:

		count	mean	max
day	smoker			
Fri	No	4	0.151650	0.187735
	Yes	15	0.174783	0.263480
Sat	No	45	0.158048	0.291990
	Yes	42	0.147906	0.325733
Sun	No	57	0.160113	0.252672
	Yes	19	0.187250	0.710345
Thur	No	45	0.160298	0.266312
	Yes	17	0.163863	0.241255

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

[48]:
```
ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
grouped['tip_pct', 'total_bill'].agg(ftuples)
```

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Pra
ctice_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-
packages\ipykernel_launcher.py:2: FutureWarning: Indexing with multiple keys
```

```
(implicitly converted to a tuple of keys) will be deprecated, use a list
instead.
```

[48]:

		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:

[49]:
```python
grouped.agg({'tip' : np.max, 'size' : 'sum'})
```

[49]:

		tip	size
day	smoker		
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

[50]:
```python
grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],'size' : 'sum'})
```

[50]:

		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

A DataFrame will have hierarchical columns only if multiple functions are applied to at least one column.

0.6.2 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:

[51]:
```
tips.groupby(['day', 'smoker'], as_index=False).mean()
```

[51]:
```
    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
```

Of course, it's always possible to obtain the result in this format by calling reset_index on the result. Using the as_index=False method avoids some unneces- sary computations

0.7 10.3 Apply: General split-apply-combine

The most general-purpose GroupBy method is apply, which is the subject of the rest of this section. As illustrated in Figure 10-2, apply splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concate- nate 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:

[52]:
```
def top(df, n=5, column='tip_pct'):
    return df.sort_values(by=column)[-n:]
top(tips, n=6)
```

[52]:
```
     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 the following:

[53]:
```
tips.groupby('smoker').apply(top)
```

[53]:
```
            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
```

What has happened here? The top function is called on each row group from the DataFrame, and then the results are glued together using pandas.concat, labeling the pieces with the group names. The result therefore has a hierarchical index whose inner level contains index values from the original DataFrame.

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

[54]:
```
tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
```

[54]:
```
                 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
```

Beyond these basic usage mechanics, getting the most out of apply may require some creativity. What occurs inside the function passed is up to you; it only needs to return a pandas object or a scalar value. The rest of this chapter will mainly consist of examples showing you how to solve various problems using groupby.

You may recall that I earlier called describe on a GroupBy object:

[55]:
```
result = tips.groupby('smoker')['tip_pct'].describe()
result
```

[55]:
```
        count      mean       std       min       25%       50%       75%  \
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625  0.185014
Yes      93.0  0.163196  0.085119  0.035638  0.106771  0.153846  0.195059

             max
smoker
No      0.291990
Yes     0.710345
```

[56]:
```
result.unstack('smoker')
```

[56]:
```
       smoker
count  No        151.000000
       Yes        93.000000
mean   No          0.159328
       Yes         0.163196
std    No          0.039910
       Yes         0.085119
min    No          0.056797
       Yes         0.035638
25%    No          0.136906
       Yes         0.106771
50%    No          0.155625
       Yes         0.153846
75%    No          0.185014
       Yes         0.195059
max    No          0.291990
       Yes         0.710345
dtype: float64
```

Inside GroupBy, when you invoke a method like describe, it is actually just a short- cut for:

[57]:
```
f = lambda x: x.describe()
grouped.apply(f)
```

[57]:
```
                   total_bill       tip  size   tip_pct
day  smoker
Fri  No     count    4.000000  4.000000  4.00  4.000000
            mean    18.420000  2.812500  2.25  0.151650
            std      5.059282  0.898494  0.50  0.028123
            min     12.460000  1.500000  2.00  0.120385
            25%     15.100000  2.625000  2.00  0.137239
…                         …       …    …         …
Thur Yes    min     10.340000  2.000000  2.00  0.090014
```

```
25%     13.510000  2.000000  2.00  0.148038
50%     16.470000  2.560000  2.00  0.153846
75%     19.810000  4.000000  2.00  0.194837
max     43.110000  5.000000  4.00  0.241255

[64 rows x 4 columns]
```

0.8 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:

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

```
[58]:      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
```

0.9 Quantile and Bucket Analysis

As you may recall from Chapter 8, 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:

```
[59]: frame = pd.DataFrame({'data1': np.random.randn(1000),'data2': np.random.
       ↪randn(1000)})
      quartiles = pd.cut(frame.data1, 4)
      quartiles[:10]
```

```
[59]: 0     (-0.456, 1.486]
      1     (-0.456, 1.486]
      2    (-2.399, -0.456]
      3     (-0.456, 1.486]
      4      (1.486, 3.429]
      5    (-2.399, -0.456]
      6     (-0.456, 1.486]
      7     (-0.456, 1.486]
      8     (-0.456, 1.486]
```

```
9     (-0.456, 1.486]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-4.349, -2.399] < (-2.399, -0.456] <
(-0.456, 1.486] < (1.486, 3.429]]
```

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:

[60]:
```
def get_stats(group):
    return {'min': group.min(), 'max': group.max(),'count': group.count(),␣
 ↪'mean': group.mean()}
grouped = frame.data2.groupby(quartiles)
```

[61]:
```
grouped.apply(get_stats).unstack()
```

[61]:
```
                       min       max  count      mean
data1
(-4.349, -2.399] -0.835328  0.373406    6.0 -0.180055
(-2.399, -0.456] -3.256739  2.944085  327.0 -0.029769
(-0.456, 1.486]  -2.387808  4.353219  607.0  0.004063
(1.486, 3.429]   -2.939968  1.795497   60.0 -0.119500
```

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

[62]:
```
# Return quantile numbers
grouping = pd.qcut(frame.data1, 10, labels=False)
grouped = frame.data2.groupby(grouping)
grouped.apply(get_stats).unstack()
```

[62]:
```
            min       max  count      mean
data1
0     -2.807301  2.559122  100.0 -0.102531
1     -2.749794  2.944085  100.0 -0.004431
2     -3.256739  2.829118  100.0 -0.011057
3     -2.032182  2.449186  100.0  0.000366
4     -1.875563  2.199241  100.0 -0.153643
5     -2.387808  2.228758  100.0 -0.017548
6     -1.932804  1.990339  100.0 -0.003607
7     -2.077586  4.353219  100.0  0.062931
8     -2.342889  2.557004  100.0  0.119070
9     -2.939968  2.116759  100.0 -0.044735
```

0.10 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. fillna is the right tool to use; for example, here I fill in NA values with the mean:

[63]:
```
s = pd.Series(np.random.randn(6))
s[::2] = np.nan
s
```

[63]:
```
0         NaN
1   -1.121237
2         NaN
3   -0.305042
4         NaN
5   -1.470527
dtype: float64
```

[64]:
```
s.fillna(s.mean())
```

[64]:
```
0   -0.965602
1   -1.121237
2   -0.965602
3   -0.305042
4   -0.965602
5   -1.470527
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:

[65]:
```
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
```

[65]:
```
Ohio         -0.869998
New York      0.034081
Vermont      -0.162439
Florida      -0.094330
Oregon       -1.394791
Nevada       -2.416494
California   -0.738579
Idaho        -0.877142
dtype: float64
```

Note that the syntax [‘East’] * 4 produces a list containing four copies of the ele- ments in [‘East’]. Adding lists together concatenates them.

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

[66]:
```
data[['Vermont', 'Nevada', 'Idaho']] = np.nan
data
```

```
[66]: Ohio          -0.869998
      New York       0.034081
      Vermont             NaN
      Florida       -0.094330
      Oregon        -1.394791
      Nevada              NaN
      California    -0.738579
      Idaho               NaN
      dtype: float64
```

```
[67]: data.groupby(group_key).mean()
```

```
[67]: East   -0.310082
      West   -1.066685
      dtype: float64
```

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

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

```
[68]: Ohio          -0.869998
      New York       0.034081
      Vermont       -0.310082
      Florida       -0.094330
      Oregon        -1.394791
      Nevada        -1.066685
      California    -0.738579
      Idaho         -1.066685
      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:

```
[69]: fill_values = {'East': 0.5, 'West': -1}
      fill_func = lambda g: g.fillna(fill_values[g.name])
      data.groupby(group_key).apply(fill_func)
```

```
[69]: Ohio          -0.869998
      New York       0.034081
      Vermont        0.500000
      Florida       -0.094330
      Oregon        -1.394791
      Nevada        -1.000000
      California    -0.738579
      Idaho         -1.000000
      dtype: float64
```

0.11 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:

[70]:
```
# Hearts, Spades, Clubs, Diamonds
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(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 = pd.Series(card_val, index=cards)
deck
```

[70]:
```
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
AS      1
2S      2
3S      3
4S      4
5S      5
6S      6
7S      7
8S      8
9S      9
10S    10
JS     10
KS     10
QS     10
AC      1
2C      2
3C      3
4C      4
```

```
5C      5
6C      6
7C      7
8C      8
9C      9
10C    10
JC     10
KC     10
QC     10
AD      1
2D      2
3D      3
4D      4
5D      5
6D      6
7D      7
8D      8
9D      9
10D    10
JD     10
KD     10
QD     10
dtype: int64
```

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 'A' be 1):

[71]:
```
deck[:13]
```

[71]:
```
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 I said before, drawing a hand of five cards from the deck could be written as:

[72]:
```
def draw(deck, n=5):
    return deck.sample(n)
```

```
draw(deck)
```

[72]:
```
10C    10
4D      4
2D      2
KH     10
JS     10
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:

[73]:
```
get_suit = lambda card: card[-1] # last letter is suit
deck.groupby(get_suit).apply(draw, n=2)
```

[73]:
```
C  AC      1
   2C      2
D  3D      3
   2D      2
H  KH     10
   AH      1
S  10S    10
   5S      5
dtype: int64
```

Alternatively, we could write:

[74]:
```
deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
```

[74]:
```
4C     4
2C     2
9D     9
8D     8
3H     3
7H     7
KS    10
7S     7
dtype: int64
```

0.12 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 a group weighted average, are possible. As an example, take this dataset containing group keys, values, and some weights:

[75]:
```
df = pd.DataFrame({'category': ['a', 'a', 'a', 'a','b', 'b', 'b', 'b'],'data':␣
↪np.random.randn(8),'weights': np.random.rand(8)})
df
```

[75]:
```
   category      data   weights
0         a -0.445467  0.890658
1         a  1.415744  0.762799
2         a  0.198141  0.402200
3         a  0.760635  0.870618
4         b  0.448171  0.616429
5         b  0.473703  0.468402
6         b  2.247355  0.979886
7         b -0.527863  0.952929
```

The group weighted average by category would then be:

[76]:
```
grouped = df.groupby('category')
get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
```

[77]:
```
grouped.apply(get_wavg)
```

[77]:
```
category
a    0.486996
b    0.728145
dtype: float64
```

As another example, consider a financial dataset originally obtained from Yahoo! Finance containing end-of-day prices for a few stocks and the S&P 500 index (the SPX symbol):

[81]:
```
close_px = pd.read_csv('stock_px.csv', parse_dates=True,index_col=0)
close_px.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5472 entries, 1990-02-01 to 2011-10-14
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   AA      5472 non-null   float64
 1   AAPL    5472 non-null   float64
 2   GE      5472 non-null   float64
 3   IBM     5472 non-null   float64
 4   JNJ     5472 non-null   float64
 5   MSFT    5472 non-null   float64
 6   PEP     5471 non-null   float64
 7   SPX     5472 non-null   float64
 8   XOM     5472 non-null   float64
dtypes: float64(9)
memory usage: 427.5 KB
```

[82]:
```
close_px[-4:]
```

[82]:
```
               AA    AAPL     GE     IBM    JNJ   MSFT    PEP      SPX    XOM
2011-10-11  10.30  400.29  16.14  185.00  63.96  27.00  60.95  1195.54  76.27
```

```
2011-10-12  10.05  402.19  16.40  186.12  64.33  26.96  62.70  1207.25  77.16
2011-10-13  10.10  408.43  16.22  186.82  64.23  27.18  62.36  1203.66  76.37
2011-10-14  10.26  422.00  16.60  190.53  64.72  27.27  62.24  1224.58  78.11
```

One task of interest might be to compute a DataFrame consisting of the yearly corre- lations of daily returns (computed from percent changes) with SPX. As one way to do this, we first create a function that computes the pairwise correlation of each column with the 'SPX' column:

[83]:
```
spx_corr = lambda x: x.corrwith(x['SPX'])
```

Next, we compute percent change on close_px using pct_change:

[84]:
```
rets = close_px.pct_change().dropna()
```

Lastly, we group these percent changes by year, which can be extracted from each row label with a one-line function that returns the year attribute of each datetime label:

[85]:
```
get_year = lambda x: x.year
by_year = rets.groupby(get_year)
by_year.apply(spx_corr)
```

[85]:
```
            AA      AAPL        GE       IBM       JNJ      MSFT       PEP  \
1990  0.595024  0.545067  0.752187  0.738361  0.801145  0.586691  0.783168
1991  0.453574  0.365315  0.759607  0.557046  0.646401  0.524225  0.641775
1992  0.398180  0.498732  0.632685  0.262232  0.515740  0.492345  0.473871
1993  0.259069  0.238578  0.447257  0.211269  0.451503  0.425377  0.385089
1994  0.428549  0.268420  0.572996  0.385162  0.372962  0.436585  0.450516
1995  0.291532  0.161829  0.519126  0.416390  0.315733  0.453660  0.413144
1996  0.292344  0.191482  0.750724  0.388497  0.569232  0.564015  0.421477
1997  0.564427  0.211435  0.827512  0.646823  0.703538  0.606171  0.509344
1998  0.533802  0.379883  0.815243  0.623982  0.591988  0.698773  0.494213
1999  0.099033  0.425584  0.710928  0.486167  0.517061  0.631315  0.336593
2000  0.265359  0.440161  0.610362  0.445114  0.189765  0.538005  0.077525
2001  0.624069  0.577152  0.794632  0.696038  0.111493  0.696447  0.133975
2002  0.748021  0.580548  0.822373  0.716490  0.584758  0.784728  0.487211
2003  0.690466  0.545582  0.777643  0.741775  0.562399  0.750534  0.541487
2004  0.591485  0.374283  0.728626  0.601740  0.354690  0.588531  0.466854
2005  0.564267  0.467540  0.675637  0.516846  0.444728  0.562374  0.489559
2006  0.487638  0.428267  0.612388  0.598636  0.394026  0.406126  0.335054
2007  0.642427  0.508118  0.796945  0.603906  0.568423  0.658770  0.651911
2008  0.781057  0.681434  0.777337  0.833074  0.801005  0.804626  0.709264
2009  0.735642  0.707103  0.713086  0.684513  0.603146  0.654902  0.541474
2010  0.745700  0.710105  0.822285  0.783638  0.689896  0.730118  0.626655
2011  0.882045  0.691931  0.864595  0.802730  0.752379  0.800996  0.592029

      SPX       XOM
1990  1.0  0.517586
1991  1.0  0.569335
1992  1.0  0.318408
```

```
1993  1.0  0.318952
1994  1.0  0.395078
1995  1.0  0.368752
1996  1.0  0.538736
1997  1.0  0.695653
1998  1.0  0.369264
1999  1.0  0.315383
2000  1.0  0.084163
2001  1.0  0.336869
2002  1.0  0.759933
2003  1.0  0.662775
2004  1.0  0.557742
2005  1.0  0.631010
2006  1.0  0.518514
2007  1.0  0.786264
2008  1.0  0.828303
2009  1.0  0.797921
2010  1.0  0.839057
2011  1.0  0.859975
```

You could also compute inter-column correlations. Here we compute the annual cor- relation between Apple and Microsoft:

[87]:
```
by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
```

[87]:
```
1990    0.408271
1991    0.266807
1992    0.450592
1993    0.236917
1994    0.361638
1995    0.258642
1996    0.147539
1997    0.196144
1998    0.364106
1999    0.329484
2000    0.275298
2001    0.563156
2002    0.571435
2003    0.486262
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
```

0.13 Example: Group-Wise Linear Regression

In the same theme as the previous example, you can use groupby to perform more complex group-wise statistical analysis, as long as the function returns a pandas object or scalar value. For example, I can define the following regress function (using the statsmodels econometrics library), which executes an ordinary least squares (OLS) regression on each chunk of data:

[88]:
```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

Now, to run a yearly linear regression of AAPL on SPX returns, execute:

[89]:
```
by_year.apply(regress, 'AAPL', ['SPX'])
```

[89]:

	SPX	intercept
1990	1.512772	0.001395
1991	1.187351	0.000396
1992	1.832427	0.000164
1993	1.390470	-0.002657
1994	1.190277	0.001617
1995	0.858818	-0.001423
1996	0.829389	-0.001791
1997	0.749928	-0.001901
1998	1.164582	0.004075
1999	1.384989	0.003273
2000	1.733802	-0.002523
2001	1.676128	0.003122
2002	1.080795	-0.000219
2003	1.187770	0.000690
2004	1.363463	0.004201
2005	1.766415	0.003246
2006	1.645496	0.000080
2007	1.198761	0.003438
2008	0.968016	-0.001110
2009	0.879103	0.002954
2010	1.052608	0.001261
2011	0.806605	0.001514

0.14 10.4 Pivot Tables and Cross-Tabulation

A pivot table is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible through the groupby facility described in this chapter combined with reshape opera- tions utilizing hierarchical indexing. DataFrame has a pivot_table method, and there is also a top-level pandas.pivot_table function. In addition to providing a convenience interface to groupby, pivot_table can add partial totals, also known as margins.

Returning to the tipping dataset, suppose you wanted to compute a table of group means (the default pivot_table aggregation type) arranged by day and smoker on the rows:

[90]:
```
tips.pivot_table(index=['day', 'smoker'])
```

[90]:
```
                 size       tip   tip_pct  total_bill
day  smoker
Fri  No      2.250000  2.812500  0.151650   18.420000
     Yes     2.066667  2.714000  0.174783   16.813333
Sat  No      2.555556  3.102889  0.158048   19.661778
     Yes     2.476190  2.875476  0.147906   21.276667
Sun  No      2.929825  3.167895  0.160113   20.506667
     Yes     2.578947  3.516842  0.187250   24.120000
Thur No      2.488889  2.673778  0.160298   17.113111
     Yes     2.352941  3.030000  0.163863   19.190588
```

This could have been produced with groupby directly. Now, suppose we want to aggregate only tip_pct and size, and additionally group by time. I'll put smoker in the table columns and day in the rows:

[91]:
```
tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],columns='smoker')
```

[91]:
```
                 size             tip_pct
smoker             No       Yes        No       Yes
time   day
Dinner Fri   2.000000  2.222222  0.139622  0.165347
       Sat   2.555556  2.476190  0.158048  0.147906
       Sun   2.929825  2.578947  0.160113  0.187250
       Thur  2.000000       NaN  0.159744       NaN
Lunch  Fri   3.000000  1.833333  0.187735  0.188937
       Thur  2.500000  2.352941  0.160311  0.163863
```

We could augment this table to include partial totals by passing margins=True. This has the effect of adding All row and column labels, with corresponding values being the group statistics for all the data within a single tier:

[92]:
```
tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],columns='smoker',␣
↪margins=True)
```

[92]:
```
                 size                          tip_pct
smoker             No       Yes       All        No       Yes       All
time   day
Dinner Fri   2.000000  2.222222  2.166667  0.139622  0.165347  0.158916
       Sat   2.555556  2.476190  2.517241  0.158048  0.147906  0.153152
       Sun   2.929825  2.578947  2.842105  0.160113  0.187250  0.166897
```

```
       Thur  2.000000       NaN  2.000000  0.159744       NaN  0.159744
Lunch  Fri   3.000000  1.833333  2.000000  0.187735  0.188937  0.188765
       Thur  2.500000  2.352941  2.459016  0.160311  0.163863  0.161301
All          2.668874  2.408602  2.569672  0.159328  0.163196  0.160803
```

Here, the All values are means without taking into account smoker versus nonsmoker (the All columns) or any of the two levels of grouping on the rows (the All row).

To use a different aggregation function, pass it to aggfunc. For example, 'count' or len will give you a cross-tabulation (count or frequency) of group sizes:

[93]:
```
tips.pivot_table('tip_pct', index=['time', 'smoker'],␣
↪columns='day',aggfunc=len, margins=True)
```

[93]:
```
day             Fri   Sat   Sun  Thur    All
time   smoker
Dinner No       3.0  45.0  57.0   1.0  106.0
       Yes      9.0  42.0  19.0   NaN   70.0
Lunch  No       1.0   NaN   NaN  44.0   45.0
       Yes      6.0   NaN   NaN  17.0   23.0
All            19.0  87.0  76.0  62.0  244.0
```

If some combinations are empty (or otherwise NA), you may wish to pass a fill_value:

[94]:
```
tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'],columns='day',␣
↪aggfunc='mean', fill_value=0)
```

[94]:
```
day                     Fri       Sat       Sun      Thur
time   size smoker
Dinner 1    No     0.000000  0.137931  0.000000  0.000000
            Yes    0.000000  0.325733  0.000000  0.000000
       2    No     0.139622  0.162705  0.168859  0.159744
            Yes    0.171297  0.148668  0.207893  0.000000
       3    No     0.000000  0.154661  0.152663  0.000000
            Yes    0.000000  0.144995  0.152660  0.000000
       4    No     0.000000  0.150096  0.148143  0.000000
            Yes    0.117750  0.124515  0.193370  0.000000
       5    No     0.000000  0.000000  0.206928  0.000000
            Yes    0.000000  0.106572  0.065660  0.000000
       6    No     0.000000  0.000000  0.103799  0.000000
Lunch  1    No     0.000000  0.000000  0.000000  0.181728
            Yes    0.223776  0.000000  0.000000  0.000000
       2    No     0.000000  0.000000  0.000000  0.166005
            Yes    0.181969  0.000000  0.000000  0.158843
       3    No     0.187735  0.000000  0.000000  0.084246
            Yes    0.000000  0.000000  0.000000  0.204952
       4    No     0.000000  0.000000  0.000000  0.138919
            Yes    0.000000  0.000000  0.000000  0.155410
       5    No     0.000000  0.000000  0.000000  0.121389
```

```
6    No       0.000000  0.000000  0.000000  0.173706
```

See Table 10-2 for a summary of pivot_table methods.

0.15 Table 10-2. pivot_table options

Function name –> Description

values –> Column name or names to aggregate; by default aggregates all numeric columns

index –> Column names or other group keys to group on the rows of the resulting pivot table

columns –> Column names or other group keys to group on the columns of the resulting pivot table

aggfunc –> Aggregation function or list of functions ('mean' by default); can be any function valid in a groupby context

fill_value –> Replace missing values in result table

dropna –> If True, do not include columns whose entries are all NA

margins –> Add row/column subtotals and grand total (False by default)

0.16 Cross-Tabulations: Crosstab

A cross-tabulation (or crosstab for short) is a special case of a pivot table that com- putes group frequencies. Here is an example:

[95]:
```
data
```

[95]:
```
Ohio          -0.869998
New York       0.034081
Vermont             NaN
Florida       -0.094330
Oregon        -1.394791
Nevada              NaN
California    -0.738579
Idaho               NaN
dtype: float64
```

As part of some survey analysis, we might want to summarize this data by nationality and handedness. You could use pivot_table to do this, but the pandas.crosstab function can be more convenient:

[97]:
```
# pd.crosstab(data.Nationality, data.Handedness, margins=True)
```

The first two arguments to crosstab can each either be an array or Series or a list of arrays. As in the tips data:

[99]:
```
pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
```

[99]:
```
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
```

[ ]:
```

```