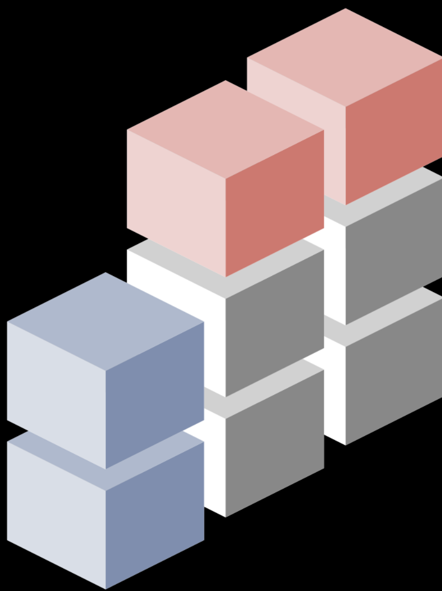
Treading on Python Series



EFFECTIVE PANDAS

Patterns for Data Manipulation

Matt Harrison

Chapter 27

Reshaping By Pivoting and Grouping

This chapter will explore one of the most powerful options for data manipulations, pivot tables. Pandas provides multiple syntaxes for creating them. One uses the `.pivot_table` method, the other common one leverages the `.groupby` method, you can also represent some of these operations with the `pd.crosstab` function.

We will explore all of these using the cleaned-up JetBrains survey data:

```
>>> jb2
       age  are_you_datascientist  ...  years_of_coding  python3_ver
1       21                   True  ...              3.0          3.6
2       30                  False  ...              3.0          3.6
10      21                  False  ...              1.0          3.8
11      21                   True  ...              3.0          3.9
13      30                   True  ...              3.0          3.7
...    ...                    ...  ...              ...          ...
54456   30                  False  ...              6.0          3.6
54457   21                  False  ...              1.0          3.6
54459   21                  False  ...              6.0          3.7
54460   30                   True  ...              3.0          3.7
54461   21                  False  ...              1.0          3.8

[13711 rows x 20 columns]
```

27.1 A Basic Example

When your boss asks you to get numbers ”by X column”, that should be a hint to pivot (or group) your data. Assume your boss asked, ”What is the average age by the country for each employment status?” This is like one of those word problems that you had to learn how to do in math class, and you needed to translate the words into math operations. In this case, we need to pick a pandas operation to use and then map the problem into those operations.

I would translate this problem into:

- Put the country in the index
- Have a column for each employment status
- Put the average age in each cell

These map cleanly to the parameters of the `.pivot_table` method. One solution would look like this:

Pivot Tables

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(auto.pivot_table(aggfunc="max",
  index="year",
  columns="make",
  values="city08")
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

Figure 27.1: The .pivot_table method allows you to pick column(s) for the index, column(s) for the column, and column(s) to aggregate. (If you specify multiple columns to aggregate, you will get hierarchical columns.)

```
>>> (jb2
...   .pivot_table(index='country_live', columns='employment_status',
...       values='age', aggfunc='mean')
... )
employment_status   Fully employed  ...  Working student
country_live                        ...
Algeria                       31.2  ...             <NA>
Argentina                30.632184  ...             23.0
Armenia                  22.071429  ...             <NA>
Australia                32.935622  ...           24.125
Austria                  31.619565  ...             25.5
...                            ...  ...              ...
United States            32.429163  ...        21.842697
Uruguay                       27.0  ...             <NA>
Uzbekistan                    21.0  ...             <NA>
Venezuela                29.769231  ...             30.0
Viet Nam                 22.857143  ...             21.0

[76 rows x 4 columns]
```

Cross Tabulation

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(pd.crosstab(aggfunc="max",
  index=auto.year,
  columns=auto.make,
  values=auto.city08)
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

Figure 27.2: The `pd.crosstab` function allows you to pick column(s) for the index, column(s) for the column, and a column to aggregate. You cannot aggregate multiple columns (unlike `.pivot_table`).

It turns out that we can use the `pd.crosstab` function as well. Because this is a function, we need to provide the data as series rather than the column names:

```
>>> pd.crosstab(index=jb2.country_live, columns=jb2.employment_status,
...     values=jb2.age, aggfunc='mean')
employment_status   Fully employed  ...  Working student
country_live                        ...
Algeria                       31.2  ...             <NA>
Argentina                30.632184  ...             23.0
Armenia                  22.071429  ...             <NA>
Australia                32.935622  ...           24.125
Austria                  31.619565  ...             25.5
...                            ...  ...              ...
United States            32.429163  ...        21.842697
Uruguay                       27.0  ...             <NA>
Uzbekistan                    21.0  ...             <NA>
Venezuela                29.769231  ...             30.0
Viet Nam                 22.857143  ...             21.0

[76 rows x 4 columns]
```

Finally, we can do this with a `.groupby` method call. The call to `.groupby` returns a `DataFrameGroupBy` object. It is a lazy object and does not perform any calculations until we indicate which aggregation to perform. We can also pull off a column and then only perform an aggregation on that column instead of all of the non-grouped columns.

This operation is a little more involved. We pull off the *age* colum and then calculate the mean for each *country_live* and *employment_status* group. Then we leverage `.unstack` to pull out the inner-most index and push it up into a column (we will dive into `.unstack` later). You can think of `.groupby` and subsequent methods as the low-level underpinnings of `.pivot_table` and `pd.crosstab`:

```
>>> (jb2
...   .groupby(['country_live', 'employment_status'])
...   .age
...   .mean()
...   .unstack()
... )
employment_status   Fully employed  ...  Working student
country_live                        ...
Algeria                       31.2  ...             <NA>
Argentina                30.632184  ...             23.0
Armenia                  22.071429  ...             <NA>
Australia                32.935622  ...           24.125
Austria                  31.619565  ...             25.5
...                            ...  ...              ...
United States            32.429163  ...        21.842697
Uruguay                       27.0  ...             <NA>
Uzbekistan                    21.0  ...             <NA>
Venezuela                29.769231  ...             30.0
Viet Nam                 22.857143  ...             21.0

[76 rows x 4 columns]
```

Many programmers and SQL analysts find the `.groupby` syntax intuitive, while Excel junkies often feel more at home with the `.pivot_table` method. The `crosstab` function works in some situations but is not as flexible. It makes sense to learn the different options. The `.groupby` method is the foundation of the other two, but a cross-tabulation may be more convenient.

27.2 Using a Custom Aggregation Function

Your boss thanks you for providing insight on the age of employment status by country and says she has a more important question: ”What is the percentage of Emacs users by country?”

We will need a function that takes a group (in this case, a series) of country respondents about IDE preference and returns the percent that chose emacs:

```
>>> def per_emacs(ser):
...     return ser.str.contains('Emacs').sum() / len(ser) * 100
```

Note

When you need to calculate a percentage in pandas, you can use the `.mean` method. The following code is equivalent to the above:

```
>>> def per_emacs(ser):
...     return ser.str.contains('Emacs').mean() * 100
```

Groupby Operation

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(auto.groupby(['year','make'])
  .city08
  .max()
  .unstack())
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

Figure 27.3: The .groupby method allows you to pick a column(s) for the index and column(s) to aggregate. You can .unstack the inner column to simulate pivot tables and cross-tabulation.

We are now ready to pivot. In this case we still want country in the index, but we only want a single column, the emacs percentage. So we don't provide a columns parameter:

```
>>> (jb2
...   .pivot_table(index='country_live', values='ide_main', aggfunc=per_emacs)
... )
               ide_main
country_live
Algeria        0.000000
Argentina      3.669725
Armenia        0.000000
Australia      3.649635
Austria        1.562500
...                 ...
United States  4.486466
Uruguay        0.000000
Uzbekistan     0.000000
Venezuela      0.000000
Viet Nam       0.000000

[76 rows x 1 columns]
```

Grouping Data

```
auto
```

	make	year	cylinders	drive
0	Alfa Romeo	1985	4.00	Rear-Wheel
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
  .groupby("make")
  .mean())
```

	year	cylinders
AM General	1984.33	5.00
ASC Incorpor	1987.00	6.00
Acura	2005.48	5.24
Alfa Romeo	1998.58	5.10
American Mot	1984.48	5.41
Volkswagen	2002.81	4.55
Volvo	2002.35	4.86
Wallace Envi	1991.50	7.81
Yugo	1988.38	4.00
smart	2013.95	3.00

Figure 27.4: When your boss asks you to get the average values by make, you should recognize that you need to pull out `.groupby('make')`.

Using `pd.crosstab` is a little more complicated as it expects a "cross-tabulation" of two columns, one column going in the index and the other column going in the columns. To get a "column" for the cross tabulation, we will assign a column to a single scalar value, (which will trick the cross tabulation into creating just one column with the name of the scalar value):

```
>>> pd.crosstab(index=jb2.country_live,
...     columns=jb2.assign(iden='emacs_per').iden,
...     values=jb2.ide_main, aggfunc=per_emacs)
iden          emacs_per
country_live
Algeria        0.000000
Argentina      3.669725
Armenia        0.000000
Australia      3.649635
Austria        1.562500
...                 ...
United States  4.486466
Uruguay        0.000000
Uzbekistan     0.000000
Venezuela      0.000000
```

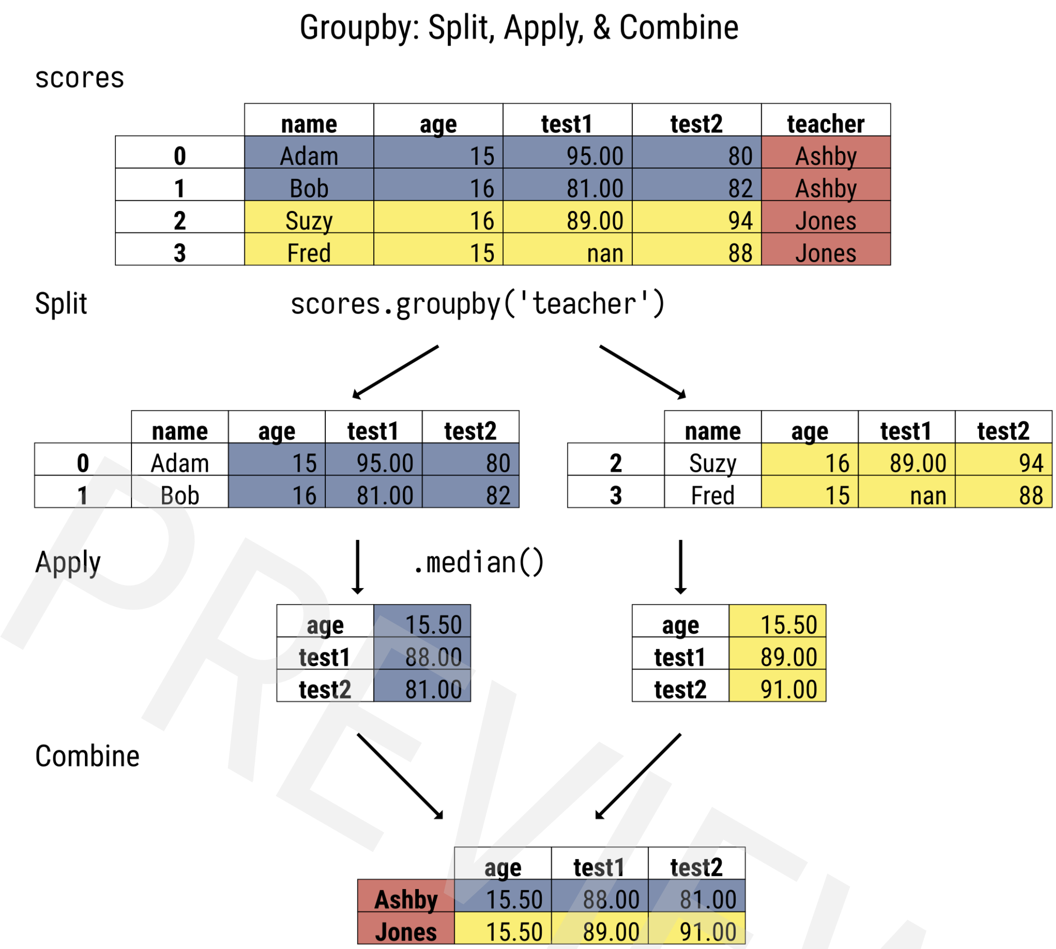


Figure 27.5: A groupby operation splits the data into groups. You can apply aggregate functions to the group. Then the results of the aggregates are combined. The column we are grouping by will be placed in the index.

```
Viet Nam         0.000000

[76 rows x 1 columns]
```

Finally, here is the `.groupby` version. I find this one very clear. Group by the *country_live* column, pull out just the *ide_main* columns. Calculate the percentage of emacs users for each of those groups:

```
>>> (jb2
...     .groupby('country_live')
...     [['ide_main']]
...     .agg(per_emacs)
... )
               ide_main
country_live
Algeria        0.000000
Argentina      3.669725
Armenia        0.000000
Australia      3.649635
Austria        1.562500
...                 ...
```

Grouping Data with Multiple Aggregations

```
auto
```

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
 .groupby('make')
 .agg(['min', 'max']))
```

Hierarchical columns!

	year	year	cylinders	cylinders
	min	max	min	max
Acura	1986	2020	4.00	6.00
Audi	1984	2020	4.00	12.00
BMW	1984	2020	2.00	12.00
BYD	2012	2019	nan	nan
Bentley	1998	2019	8.00	12.00
VPG	2011	2013	8.00	8.00
Vector	1992	1997	8.00	12.00
Volvo	1984	2019	4.00	8.00
Yugo	1986	1990	4.00	4.00
smart	2008	2019	3.00	3.00

Figure 27.6: You can leverage the `.agg` method with `.groupby` to perform multiple aggregations.

```
United States   4.486466
Uruguay         0.000000
Uzbekistan      0.000000
Venezuela       0.000000
Viet Nam        0.000000

[76 rows x 1 columns]
```

27.3 Multiple Aggregations

Assume that your boss asked, "What is the minimum and maximum age for each country?" When you see "for each" or "by", your mind should think that whatever is following either of the terms should go in the index. This question is answered with a pivot table or using groupby. (We can use a cross-tabulation, but you will need to add a column to do this, and it feels unnatural to me).

Here is the `.pivot_table` solution. The *country_live* column goes in the `index` parameter. *age* is what we want to aggregate, so that goes in the `values` parameter. And we need to specify a sequence with `min` and `max` for the `aggfunc` parameter:

```
>>> (jb2
...     .pivot_table(index='country_live', values='age',
...        aggfunc=(min, max))
... )
              max min
country_live
Algeria        60  18
Argentina      60  18
Armenia        30  18
Australia      60  18
Austria        50  18
...            ..  ..
United States  60  18
Uruguay        40  21
Uzbekistan     21  21
Venezuela      50  18
Viet Nam       60  18

[76 rows x 2 columns]
```

When you look at this using the `.groupby` method, you first determine what you want in the index, *country_live*. Then we will pull off the *age* column from each group. Finally, we will apply two aggregate functions, `min` and `max`:

```
>>> (jb2
...  .groupby('country_live')
...  .age
...  .agg([min, max])
... )
               min  max
country_live
Algeria         18   60
Argentina       18   60
Armenia         18   30
Australia       18   60
Austria         18   50
...            ...  ...
United States   18   60
Uruguay         21   40
Uzbekistan      21   21
Venezuela       18   50
Viet Nam        18   60

[76 rows x 2 columns]
```

Here is the example for `pd.crosstab`. I don't recommend this, but provide it to help explain how cross-tabulation works. Again, we want *country_live* in the index. With cross-tabulation, we need to provide a series to splay out in the columns. We cannot use the *age* column as the `columns` parameter because we want to aggregate on those numbers and hence need to set them as the `values` parameter. Instead, I will create a new column that has a single scalar value, the string `'age'`. We can provide both of the aggregations we want to use to the `aggfunc` parameter. Below is my solution. Note that is has hierarchical columns:

```
>>> pd.crosstab(jb2.country_live, values=jb2.age, aggfunc=(min, max),
...     columns=jb2.assign(val='age').val)
              max min
val           age age
country_live
Algeria        60  18
Argentina      60  18
Armenia        30  18
Australia      60  18
Austria        50  18
...            ..  ..
United States  60  18
Uruguay        40  21
Uzbekistan     21  21
Venezuela      50  18
Viet Nam       60  18

[76 rows x 2 columns]
```

27.4 Per Column Aggregations

In the previous example, we looked at applying multiple aggregations to a single column. We can also apply multiple aggregations to many columns. Here we get the minimum and maximum value of each numeric column by country:

```
>>> (jb2
...  .pivot_table(index='country_live',
...               aggfunc=(min, max))
... )
              age      ... years_of_coding
              max min  ...             max  min
country_live           ...
Algeria        60  18  ...            11.0  1.0
Argentina      60  18  ...            11.0  1.0
Armenia        30  18  ...            11.0  1.0
Australia      60  18  ...            11.0  1.0
Austria        50  18  ...            11.0  1.0
...            ..  ..  ...             ...  ...
United States  60  18  ...            11.0  1.0
Uruguay        40  21  ...            11.0  1.0
Uzbekistan     21  21  ...             6.0  1.0
Venezuela      50  18  ...            11.0  1.0
Viet Nam       60  18  ...             6.0  1.0

[76 rows x 32 columns]
```

Here is the groupby version:

```
>>> (jb2
...  .groupby('country_live')
...  .agg([min, max])
... )
              age      ... years_of_coding
              max min  ...             max  min
country_live           ...
Algeria        60  18  ...            11.0  1.0
Argentina      60  18  ...            11.0  1.0
Armenia        30  18  ...            11.0  1.0
Australia      60  18  ...            11.0  1.0
Austria        50  18  ...            11.0  1.0
```

```
...              ..  ..  ...          ...  ...
United States    60  18  ...         11.0  1.0
Uruguay          40  21  ...         11.0  1.0
Uzbekistan       21  21  ...          6.0  1.0
Venezuela        50  18  ...         11.0  1.0
Viet Nam         60  18  ...          6.0  1.0

[76 rows x 32 columns]
```

I'm not going to do this with `pd.crosstab`, and I recommend that you don't as well.

Sometimes, we want to specify aggregations per column. With both the `.pivot_table` and `.groupby` methods, we can provide a dictionary mapping a column to an aggregation function or a list of aggregation functions.

Assume your boss asked: "What are the minimum and maximum ages and the average team size for each country?". Here is the translation to a pivot table:

```
>>> (jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'team_size': 'mean'})
... )
              age     team_size
              max min      mean
country_live
Algeria        60  18  3.722222
Argentina      60  18  4.146789
Armenia        30  18  4.235294
Australia      60  18  3.354015
Austria        50  18  3.132812
...            ..  ..       ...
United States  60  18  4.072673
Uruguay        40  21  3.700000
Uzbekistan     21  21  2.750000
Venezuela      50  18  3.227273
Viet Nam       60  18  4.666667

[76 rows x 3 columns]
```

Here is the groupby version:

```
>>> (jb2
...  .groupby('country_live')
...  .agg({'age': ['min', 'max'],
...       'team_size': 'mean'})
... )
              age     team_size
              min max      mean
country_live
Algeria        18  60  3.722222
Argentina      18  60  4.146789
Armenia        18  30  4.235294
Australia      18  60  3.354015
Austria        18  50  3.132812
...            ..  ..       ...
United States  18  60  4.072673
Uruguay        21  40  3.700000
Uzbekistan     21  21  2.750000
Venezuela      18  50  3.227273
Viet Nam       18  60  4.666667

[76 rows x 3 columns]
```

One nuisance of these results is that they have hierarchical columns. In general, I find these types of columns annoying and confusing to work with. They do come in useful for stacking and unstacking, which we will explore in a later section. However, I like to remove them, and I will also show a general recipe for that later.

But I want to show one last feature that is specific to `.groupby` and may make you favor it as there is no equivalent functionality found in `.pivot_table`. That feature is called *named aggregations*. When calling the `.agg` method on a groupby object, you can use a keyword parameter to pass in a tuple of the column and aggregation function. The keyword parameter will be turned into a (flattened) column name.

We could re-write the previous example like this:

```
>>> (jb2
...  .groupby('country_live')
...  .agg(age_min=('age', min),
...       age_max=('age', max),
...       team_size_mean=('team_size', 'mean')
...      )
... )
               age_min  age_max  team_size_mean
country_live
Algeria             18       60        3.722222
Argentina           18       60        4.146789
Armenia             18       30        4.235294
Australia           18       60        3.354015
Austria             18       50        3.132812
...                ...      ...             ...
United States       18       60        4.072673
Uruguay             21       40        3.700000
Uzbekistan          21       21        2.750000
Venezuela           18       50        3.227273
Viet Nam            18       60        4.666667

[76 rows x 3 columns]
```

Notice that the above result has flat columns.

27.5 Grouping by Hierarchy

I just mentioned how much hierarchical columns bothered me. I'll admit, they are sometimes useful. Now I'm going to show you how to create hierarchical indexes. Suppose your boss asked about minimum and maximum age for each country and editor. We want to have both the country and the editor in the index. We just need to pass in a list of columns we want in the index:

```
>>> (jb2.pivot_table(index=['country_live', 'ide_main'],
...   values='age', aggfunc=[min, max]))
                                            min max
                                            age age
country_live ide_main
Algeria      Atom                            21  60
             Eclipse + Pydev                 18  18
             IDLE                            40  40
             Jupyter Notebook                30  30
             Other                           30  30
...                                          ..  ..
Viet Nam     Other                           21  21
             PyCharm Community Edition       21  30
             PyCharm Professional Edition    21  21
```

Flattening Grouping Data by Multiple Columns

```
auto
```

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
 .groupby(['make', 'year'])
 .max()
 .reset_index())
```

	make	year	cylinders
0	Acura	1986	6.00
1	Acura	1987	6.00
2	Acura	1988	6.00
3	Acura	1989	6.00
4	Acura	1990	6.00
1345	smart	2015	3.00
1346	smart	2016	3.00
1347	smart	2017	3.00
1348	smart	2018	nan
1349	smart	2019	nan

Figure 27.7: Grouping with a list of columns will create a multi-index, an index with hierarchical levels.


```
             VS Code                            18   30
             Vim                                21   40

[813 rows x 2 columns]
```

Here is the groupby version:

```
>>> (jb2
...   .groupby(by=['country_live', 'ide_main'])
...   [['age']]
...   .agg([min, max])
... )
                                                   age
                                                   min    max
country_live ide_main
Algeria      Atom                                   21     60
             Eclipse + Pydev                        18     18
             Emacs                                <NA>   <NA>
             IDLE                                   40     40
             IntelliJ IDEA                        <NA>   <NA>
...                                                ...    ...
Viet Nam     Python Tools for Visual Studio (PTVS) <NA>   <NA>
             Spyder                               <NA>   <NA>
```

```
             Sublime Text                          <NA>  <NA>
             VS Code                                 18    30
             Vim                                     21    40

[1216 rows x 2 columns]
```

Those paying careful attention will note that the results of apply multiple aggregations from `.groupby` and `.pivot_table` are not exactly the same. There are a few differences:

- The hierarchical column levels are swapped (*age* is inside of *min* and *max* when pivotting, but outside when grouping)
- The row count differs

I'm not sure why pandas swaps the levels. You could use the `.swaplevel` method to change that. However, I would personally use a named aggregation with a groupby for flat columns:

```
>>> (jb2
...   .groupby(by=['country_live', 'ide_main'])
...   [['age']]
...   .agg([min, max])
...   .swaplevel(axis='columns')
... )
                                                        min   max
                                                        age   age
country_live ide_main
Algeria      Atom                                        21    60
             Eclipse + Pydev                             18    18
             Emacs                                     <NA>  <NA>
             IDLE                                        40    40
             IntelliJ IDEA                             <NA>  <NA>
...                                                     ...   ...
Viet Nam     Python Tools for Visual Studio (PTVS)     <NA>  <NA>
             Spyder                                    <NA>  <NA>
             Sublime Text                              <NA>  <NA>
             VS Code                                     18    30
             Vim                                         21    40

[1216 rows x 2 columns]

>>> (jb2
...   .groupby(by=['country_live', 'ide_main'])
...   .agg(age_min=('age', min), age_max=('age', max))
... )
                                                    age_min  age_max
country_live ide_main
Algeria      Atom                                        21       60
             Eclipse + Pydev                             18       18
             Emacs                                     <NA>     <NA>
             IDLE                                        40       40
             IntelliJ IDEA                             <NA>     <NA>
...                                                     ...      ...
Viet Nam     Python Tools for Visual Studio (PTVS)     <NA>     <NA>
             Spyder                                    <NA>     <NA>
             Sublime Text                              <NA>     <NA>
             VS Code                                     18       30
             Vim                                         21       40

[1216 rows x 2 columns]
```

The reason the row count is different is a little more nuanced. I have set the *country_live* and *ide_main* columns to be categorical. When you perform a groupby with categorical columns, pandas will create the cartesian product of those columns even if there is no corresponding value. You can see above a few rows with both values of `<NA>`. The pivot table version (at the start of the section) did not have the missing values.

Note

Be careful when grouping with multiple categorical columns with high cardinality. You can generate a very large (and sparse) result!

You could always call `.dropna` after the fact, but I prefer to use the `observed` parameter instead:

```
>>> (jb2
...  .groupby(by=['country_live', 'ide_main'], observed=True)
...  .agg(age_min=('age', min), age_max=('age', max))
... )
                                                 age_min  age_max
country_live       ide_main
India              Atom                               18       40
                   Eclipse + Pydev                    18       40
                   Emacs                              21       40
                   IDLE                               18       40
                   IntelliJ IDEA                      21       30
...                                                  ...      ...
Dominican Republic Vim                                21       21
Morocco            Jupyter Notebook                   30       30
                   PyCharm Community Edition          21       40
                   Sublime Text                       21       30
                   VS Code                            21       30

[813 rows x 2 columns]
```

That's looking better!

27.6 Grouping with Functions

Up until now, we have been grouping by various values found in columns. Sometimes I want to group by something other than an existing column, and I have a few options.

Often, I will create a special column containing the values I want to group by. In addition, both pivot tables and groupby operations support passing in a function instead of a column name. This function accepts a single index label and should return a value to group on. In the example below we group based on whether the index value is even or odd. We then calculate the size of each group. Here is the grouper function and the `.pivot_table` implementation:

```
>>> def even_grouper(idx):
...     return 'odd' if idx % 2 else 'even'

>>> jb2.pivot_table(index=even_grouper, aggfunc='size')
even    6849
odd     6862
dtype: int64
```

And here is the `.groupby` version:

```
>>> (jb2
...  .groupby(even_grouper)
...  .size()
```

```
... )
even    6849
odd     6862
dtype: int64
```

When we look at time series manipulation later, we will see that pandas provides a handy `pd.Grouper` class to allow us to easily group by time attributes.

Method	*Description*
`pd.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins_name='All', dropna=True, normalize=False)`	Create a cross-tabulation (counts by default) from an `index` (series or list of series) and `columns` (series or list of series). Can specify a column (series) to aggregate `values` along with a function, `aggfunc`. Using `margins=True` will add subtotals. Using `dropna=False` will keep columns that have no values. Can normalize over `'all'` values, the rows (`'index'`), or the `'columns'`.
`.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, margins_name='All', dropna=True, observed=False, sort=True)`	Create a pivot table. Use `index` (series, column name, `pd.Grouper`, or list of previous) to specify index entries. Use `columns` (series, column name, `pd.Grouper`, or list of previous) to specify column entries. The `aggfunc` (function, list of functions, dictionary (column name to function or list of functions) specifies function to aggregate values. Missing values are replaced with `fill_value`. Set `margins=True` to add subtotals/totals. Using `dropna=False` will keep columns that have no values. Use `observed=True` to only show values that appeared for categorical groupers.
`.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, observed=False, dropna=True)`	Return a grouper object, grouped using `by` (column name, function (accepts each index value, returns group name/id), series, `pd.Grouper`, or list of column names). Use `as_index=False` to leave grouping keys as columns. Common plot parameters. Use `observed=True` to only show values that appeared for categorical groupers. Using `dropna=False` will keep columns that have no values.
`.stack(level=-1, dropna=True)`	Push column level into the index level. Can specify a column `level` (-1 is innermost).
`.unstack(level=-1, dropna=True)`	Push index level into the column level. Can specify an index `level` (-1 is innermost).

Table 27.1: Dataframe Pivoting and Grouping Methods

Method	*Description*
Column access	Access a column by attribute or index operation.

`g.agg(func=None, *args, engine=None, engine_kwargs=None, **kwargs)`	Apply an aggregate `func` to groups. `func` can be string, function (accepting a column and returning a reduction), a list of the previous, or a dictionary mapping column name to string, function, or list of strings and/or functions.
`g.aggregate`	Same as `g.agg`.
`g.all(skipna=True)`	Collapse each group to `True` if all the values are truthy.
`g.any(skipna=True)`	Collapse each group to `True` if any the values are truthy.
`g.apply(func, *args, **kwargs)`	Apply a function to each group. The function should accept the group (as a dataframe) and return scalar, series, or dataframe. These return a series, dataframe (with each series as a row), and a dataframe (with the index as an inner index of the result) respectively.
`g.count()`	Count of non-missing values for each group.
`g.ewm(com=None, span=None, halflife=None)`	Return an Exponentially Weighted grouper. Can specify center of mass (`com`), decay `span`, or `halflife`. Will need to apply further aggregation to this.
`g.expanding(min_periods=1, center=False, axis=0, method='single')`	Return an expanding Window object. Can specify minimum number of observations per period (`min_periods`), set label at `center` of window, and whether to execute over `'single'` column or whole group (`'table'`). Will need to apply further aggregation to this.
`g.filter(func, dropna=True, *args, **kwargs)`	Return the original dataframe but with filtered groups removed. `func` is a predicate function that accepts a group and returns `True` to keep values from group. If `dropna=False`, groups that evaluate to `False` are filled with `NaN`.
`g.first(numeric_only=False, min_count=-1)`	Return the first row of each group. If `min_count` set to positive value, then group must have that many rows or values are filled with `NaN`.
`g.get_group(name, obj=None)`	Return a dataframe with named group.
`g.groups`	Property with dictionary mapping group name to list of index values. (See `.indices`.)
`g.head(n=5)`	Return the first `n` rows of each group. Uses original index.
`g.idxmax(axis=0, skipna=True)`	Return an index label of maximum value for each group.
`g.idxmin(axis=0, skipna=True)`	Return an index label of minimum value for each group.
`g.indices`	Property with a dictionary mapping group name to `np.array` of index values. (See `.groups`.)
`g.last(numeric_only=False, min_count=-1)`	Return the last row of each group. If `min_count` set to positive value, then group must have that many rows or values are filled with `NaN`.
`g.max(numeric_only=False, min_count=-1)`	Return the maximum row of each group. If `min_count` set to positive value, then group must have that many rows or values are filled with `NaN`.

`g.mean(numeric_only=True)`	Return the mean of each group.
`g.min(numeric_only=False, min_count=-1)`	Return the minimum row of each group. If `min_count` set to positive value, then group must have that many rows or values are filled with `NaN`.
`g.ndim`	Property with the number of dimensions of result.
`g.ngroup(ascending=True)`	Return a series with original index and values for each group number.
`g.ngroups`	Property with the number of groups.
`g.nth(n, dropna=None)`	Take the nth row from each group.
`g.nunique(dropna=True)`	Return a dataframe with unique counts for each group.
`g.ohlc()`	Return a dataframe with open, high, low, and close values for each group.
`g.pipe(func, *args, **kwargs)`	Apply the `func` to each group.
`g.prod(numeric_only=True, min_count=0)`	Return a dataframe with product of each group.
`g.quantile(q=.5, interpolation='linear')`	Return a dataframe with quantile for each group. Can pass a list for `q` and get inner index for each value.
`g.rank(method='average', na_option='keep', ascending=True, pct=False, axis=0)`	Return a dataframe with numerical ranks for each group. `method` allows to specify tie handling. `'average'`, `'min'`, `'max'`, `'first'` (uses order they appear in series), `'dense'` (like `'min'`, but rank only increases by one after tie). `na_option` allows you to specify `NaN` handling. `'keep'` (stay at `NaN`), `'top'` (move to smallest), `'bottom'` (move to largest).
`g.resample(rule, *args, **kwargs)`	Create a resample object with offset alias frequency specified by `rule`. Will need to apply further aggregation to this.
`g.rolling(window_size)`	Create a rolling grouper. Will need to apply further aggregation to this.
`g.sample(n=None, frac=None, replace=False, weights=None, random_state=None)`	Return a dataframe with sample from each group. Uses original index.
`g.sem(ddof=1)`	Return the mean of standard error of mean each group. Can specify degrees of freedom (`ddof`).
`g.shift(periods=1, freq=None, axis=0, fill_value=None`	Create a shifted values for each group. Uses original index.
`g.size()`	Return a series with size of each group.
`g.skew(axis=0, skipna=True, level=None, numeric_only=False)`	Return a series with numeric columns inserted as inner level of grouped index with unbiased skew.
`g.std(ddof=1)`	Return the standard deviation of each group. Can specify degrees of freedom (`ddof`).
`g.sum(numeric_only=True, min_count=0)`	Return a dataframe with the sum of each group.
`g.tail(n=5)`	Return the last `n` rows of each group. Uses original index.
`g.take(indices, axis=0)`	Return a dataframe with the index positions (`indices`) from each group. Positions are relative to group.

`g.transform(func, *args, **kwargs)`	Return a dataframe with the original index. The function will get passed a group and should return dataframe with same dimensions as group.
`g.var(ddof=1)`	Return the variance of each group. Can specify degrees of freedom (`ddof`).

Table 27.2: Groupby Methods and Operations

27.7 Summary

Grouping is one of the most powerful tools that pandas provides. It is the underpinning of the `.pivot_table` method, which in turn implements the `pd.crosstab` function. These constructs can be hard to learn because of the inherent complexity of the operation, the hierarchical nature of the result, and the syntax. If you are using `.groupby` remember to write out your chains and step through them one step at a time. That will help you understand what is going on. You will also need to practice these. Once you learn the syntax, practicing will help you master these concepts.

27.8 Exercises

With a dataset of your choice:

1. Group by a categorical column and take the mean of the numeric columns.
2. Group by a categorical column and take the mean and max of the numeric columns.
3. Group by a categorical column and apply a custom aggregation function that calculates the mode of the numeric columns.
4. Group by two categorical columns and take the mean of the numeric columns.
5. Group by binned numeric column and take the mean of the numeric columns.