7 Advanced grouping, joining, and sorting

In the previous chapter, we used three of the central tools in pandas: grouping data across different columns, joining multiple data frames, and sorting a data frame by its index or one or more columns. As we saw, each of these techniques gives us a powerful way to manipulate our data into a form that allows for better understanding and interpretation.

In this chapter, we'll explore deeper ways to use these techniques, both by themselves and together. We'll turn multiple CSV files into a single data frame, we'll group and sort by multiple columns, and we'll use the `filter` method to keep and reject rows based on group properties. After going through the exercises in this chapter, you'll have an even stronger understanding of these techniques, how they can help you solve problems, and when it's appropriate to use them.

Table 7.1 What you need to know

Concept	What is it?	Example	To learn more
`s.isnull`	Returns a boolean series indicating where there are null (typically `NaN`) values in the series `s`	`s.isnull()`	http://mng.bz/ngYe
`df.sort_index`	Reorders the rows of a data frame based on the values in its index, in ascending order	`df = df.sort_index()`	http://mng.bz/wvB7
`s.isnull`	Returns a boolean series indicating where there are null (typically `NaN`) values in the series `s`	`s.isnull()`	http://mng.bz/Jgyp
`df.sort_index`	Reorders the rows of a data frame based on the values in its index, in ascending order	`df = df.sort_index()`	http://mng.bz/wvB7
`df.sort_values`	Reorders the rows of a data frame based on the values in one or more specified columns	`df = df.sort_values('distance')`	http://mng.bz/qrMK
`df.transpose()` or `df.T`	Returns a new data frame with the same values as `df` but with the columns and index exchanged	`df.transpose()` or `df.T`	http://mng.bz/7DXx
`df.expanding`	Lets us run window functions on an expanding (growing) set of rows	`df.expanding().sum()`	http://mng.bz/mVBn
`df.rolling`	Lets us run window functions on an expanding	`df.rolling(3).mean()`	http://mng.bz/5wp4

	(growing) set of rows		
`df.pct_change`	For a given data frame, indicates the percentage difference between each cell and the corresponding cell in the previous row	`df.pct_change()`	http://mng.bz/4DBB
`df.diff`	For a given data frame, indicates the difference between each cell and the corresponding cell in the previous row	`df.diff()`	http://mng.bz/OPDE
`df.groupby`	Allows us to invoke one or more aggregate methods for each value in a particular column	`df.groupby('year')`	http://mng.bz/vn9x
`df.loc`	Retrieves selected rows and columns	`df.loc[:, 'passen-ger_count'] = df['passenger_count']`	http://mng.bz/nWzv
`s.iloc`	Accesses elements of a series by position	`s.iloc[0]`	http://mng.bz/QPxm
`df.dropna`	Removes rows with `NaN` values	`df = df.dropna()`	http://mng.bz/XN0Y
`s.unique`	Gets the unique values in a series (`drop_duplicates` is better)	`s.unique()`	http://mng.bz/yQrJ
`df.join`	Joins two data frames based on their indexes	`df.join(other_df)`	http://mng.bz/MBo2
`df.merge`	Joins two data frames based on any columns	`df.merge(other_df)`	http://mng.bz/a1wJ
`df.corr`	Shows the correlation	`df.corr()`	http://mng.bz/gBgR

	between the numeric columns of a data frame		
`s.to_frame`	Turns a series into a one-column data frame	`s.to_frame()`	http://mng.bz/5wp1
`s.removesuffix`	Returns a new string with the same contents as `s` but without a specified suffix (if it’s there)	`s.removesuffix('.csv')`	http://mng.bz/6DAD
`s.removeprefix`	Returns a new string with the same contents as `s` but without a specified prefix (if it’s there)	`s.removeprefix('abcd')`	http://mng.bz/o1Rr
`s.title`	Returns a new string based on `s` in which each word starts with a capital letter	`s.title('hello out there')`	http://mng.bz/nWzg
`pd.concat`	Returns one new data frame based on a list of data frames passed to `pd.concat`	`pd.concat([df1, df2, df3])`	http://mng.bz/vn9J
`df.assign`	Adds one or more columns to a data frame	`df.assign(a=df['x']*3)`	http://mng.bz/1J1V
`DataFrameGroupBy.agg`	Applies multiple aggregation methods to a `groupby`	`df.groupby('a')['b'].agg(['mean', 'std'])`	http://mng.bz/v8o1
`DataFrameGroupBy.filter`	Keeps rows whose group results in `True` from an outside function	`df.groupby('a').filter(filter_func)`	http://mng.bz/z0BQ
`DataFrameGroupBy.transform`	Modifies rows based on an outside function	`df.groupby('a').transform(transform_func)`	http://mng.bz/0l26

`df.rename`	Renames columns in a data frame	`df.rename(columns={'a':'b', 'c':'d'})`	http://mng.bz/K9W0
`df.drop_duplicates`	Returns a data frame whose rows contain distinct values	`df.drop_duplicates()`	http://mng.bz/9Qv1
`df.drop`	Removes rows or columns from a data frame, returning a new one	`df.drop('a', axis='columns')`	http://mng.bz/j1eP

Exercise 32 • Multicity temperatures

Grouping is one of the most useful and common functions we use when analyzing data. That's because although it's helpful to get an overall view of a data set, it's even *more* useful to learn about the different pieces of the data set so we can compare them with one another. For example, we may want to know how many people voted in the most recent election. But if we're interested in running a campaign that encourages more people to vote, we'll want to count voters from each age range, location, or ethnicity, to target our campaign more effectively.

In this exercise, we're going to get some additional practice with grouping. But I've added another challenge: creating the data frame on which you'll perform the grouping. That's because I want you to create the data frame based on eight different CSV files, each of which contains weather data from a different city. Moreover, the eight cities come from four different US states—and we want the data frame to contain `city` and `state` columns so we can work with them individually in that way.

Each of the files you'll load has the same column names and format. Take advantage of that when loading the data into a data frame.

Specifically, I'd like you to

1. Take the eight CSV files I've provided, containing weather data from eight different cities (spanning four states), and turn them into a data frame. The files are **san+francisco,ca.csv**, **new+york,ny.csv**, **springfield,ma.csv**, **boston,ma.csv**, **springfield,il.csv**, **albany,ny.csv**, **los+angeles,ca.csv**, and **chicago,il.csv**.
2. We are only interested in the first three columns from each CSV file: the date and time, the max temperature, and the min temperature.
3. Add `city` and `state` columns that contain the city and state from the filename and allow us to distinguish between rows.

Once you've done all that, answer the following questions:

- Does the data for each city and state start and end at (roughly) the same time? How do you know?
- What is the lowest minimum temperature recorded for each city in the data set?
- What is the highest maximum temperature recorded in each *state* in the data set?

Working it out

One of the most important things I tell newcomers to programming is that your choice and design of data structures has a huge effect on the programs you write. When you're working with Python, you should think carefully about whether you'll use a list, a tuple, a dictionary, or some combination of those.

The pandas analog to this advice is that you should design your data frames such that they include all the information you need to simplify your queries. This sometimes means you'll need to do some additional manipulations and calculations when loading data from files—but for the most part, having your data in a clear and organized data frame opens the door to straightforward, easy-to-understand, and efficient queries.

In this exercise, we need columns for `city` and `state` (where the temperature reports were made), the date and time of the reading, the maximum temperature recorded, and the minimum temperature recorded. We can get the city and state from the filename and the final three values from the rows of the CSV files. We'll aim to create a data frame with these columns from each of the cities and then combine them into one large data frame.

Let's first consider how we can create a data frame from the combination of all these CSV files. We already know how to read in a single CSV file using `read_csv`:

```
one_filename = 'new+york,ny.csv'
one_df = pd.read_csv(one_filename)
```

However, we aren't interested in every column in the CSV file. We thus pass a number of key-value pairs:

- `usecols`, specifying which columns we want to read and use from the CSV file. Here, we specify them using integers.
- `names`, indicating what column names we want in the data frame, ensuring that the names are the same across all files.
- `header`, indicating that the first row (i.e., line 0) of the file contains the header information—mostly so we can ignore the names and replace them with our own.

Our call to `read_csv` ends up like this:

```
one_df = (
    pd
    .read_csv(one_filename,
              usecols=[0, 1, 2],
              names=['date_time', 'max_temp',
                     'min_temp'],
              header=0)
)
```

This code gives us the three columns we want from CSV-file data. However, we still need to extract the city and state info from the filename and add that to the data frame.

First, we remove the .csv suffix:

```
base_filename = one_filename.removesuffix('.csv')
```

The filenames, at least as I've defined them for the Jupyter notebooks we're using for this book, are all in a parallel directory called ../data. So the real filename is ../data/new+york,ny.csv, which means we need to remove both the prefix and the suffix. We can do that in one line via method chaining:

```
one_filename.removeprefix('../data/').removesuffix('.csv')
```

This whole expression returns a string that we could assign to a new variable. But we want to get the city and state from the string, so we'll run the Python `str.split` method, which returns a list of strings based on breaking a string into multiple parts. All we have to do is indicate what character serves as a field delimiter in this string—in this case, a comma:

```
one_filename.removeprefix('../data/').removesuffix('.csv').split(',')
```

Given that we know how these files are named, we can be sure that the result of calling `str.split` is a two-element list in which the first element is the city name and the second element is the two-letter state abbreviation. Thanks to Python's "tuple unpacking" feature, we can assign the elements of this list to two variables:

```
city, state = (
    one_filename
    .removeprefix('../data/')
    .removesuffix('.csv')
    .split(',')
)
```

Just like that, the `city` variable contains the city name from the filename, and the `state` variable contains the state abbreviation.

We now have `one_df` (a variable containing a data frame) and both `city` and `state`. How can we put the values from the variables `city` and `state` into columns `city` and `state` in `one_df`?

One way is to assign a scalar value to a new column, which has the effect of assigning that value to every row in the column:

```
one_df['city'] = city
one_df['state'] = state
```

But there's something wrong here: the city names contain `+` signs instead of space characters and are written in lowercase letters. Similarly, the state abbreviations are in lowercase letters. We can fix that, using some additional Python string methods:

```
one_df['city'] = city.replace('+', ' ').title()
one_df['state'] = state.upper()
```

Although this code works, we should use method chaining when importing the files. We can do that by using `assign`, which temporarily adds one or more new columns to a data frame. We can say this, using method-chaining syntax:

```
one_df = (
    pd
    .read_csv(one_filename,
              usecols=[0, 1, 2],
              names=['date_time', 'max_temp',
                     'min_temp'],
              header=0)
    .assign(city=city.replace('+', ' ').title(),
            state=state.upper())
)
```

In other words: `read_csv` returns a new data frame based on the city's CSV file. Before returning that data frame to the caller, we add `city` and `state` columns. The result, assigned to `one_df`, is a data frame with the five columns we want.

How can we use this template to read data from all eight CSV files into a single data frame? We can use `pd.concat`, which takes a list of data frames and returns a single, combined data frame. If we can create a list of data frames, each based on a different CSV file, we'll have the data as we need it.

To do that, we use a `for` loop, iterating over the list of filenames returned by `glob.glob`, a function in Python's standard library. We iterate over each filename we get back from `glob.glob`, create a data frame from its contents, add the city and state, and append that data frame to our list. After all the iterations are done, we can use `pd.concat` to combine them:

```
import glob

all_dfs = []
```

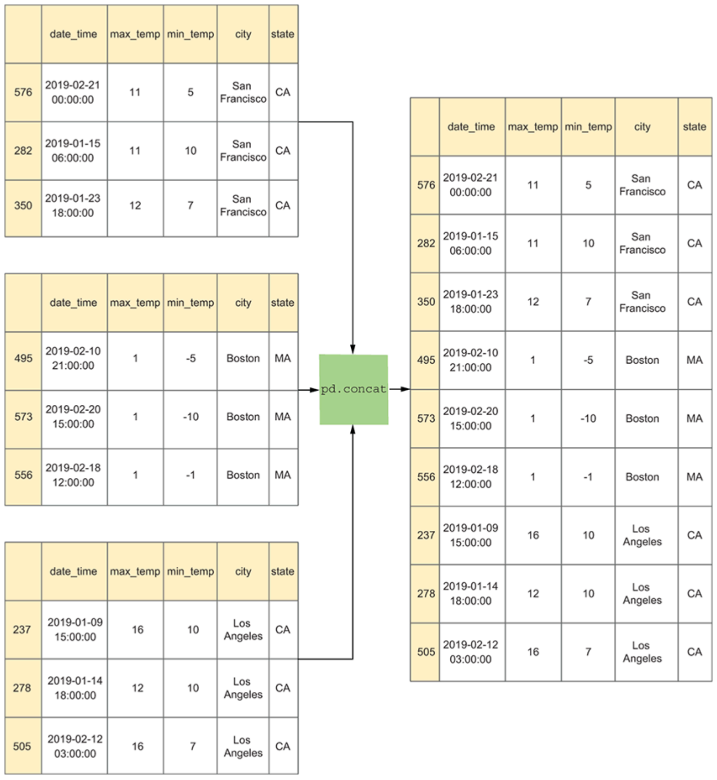
```
for one_filename in glob.glob('../data/*,*.csv'):
    print(f'Loading {one_filename}...')
    city, state = (
        one_filename
        .removeprefix('../data/')
        .removesuffix('.csv')
        .split(',')
    )

    one_df = (
        pd
        .read_csv(one_filename,
                  usecols=[0, 1, 2],
                  names=['date_time', 'max_temp',
                         'min_temp'],
                  header=0)
        .assign(city=city.replace('+', ' ').title(),
                state=state.upper())
    )

    all_dfs.append(one_df)
```

In this code, we iterate over each filename that matches the pattern `*,*.csv`. We create a new data frame from that CSV file and add (with `assign`) a new `city` column (based on the city name, which we get from `one_filename`) and a new `state` column (again, based on the state abbreviation, which we also get from `one_filename`).

We append each data frame to `all_dfs`, a list, such that we'll grow the list with one new element per CSV file. When we're done with all the data frames, we then create `df`, the result of concatenating them (figure 7.1). We can then run `pd.concat` and get a single data frame from the list.



	date_time	max_temp	min_temp	city	state
576	2019-02-21 00:00:00	11	5	San Francisco	CA
282	2019-01-15 06:00:00	11	10	San Francisco	CA
350	2019-01-23 18:00:00	12	7	San Francisco	CA

	date_time	max_temp	min_temp	city	state
495	2019-02-10 21:00:00	1	-5	Boston	MA
573	2019-02-20 15:00:00	1	-10	Boston	MA
556	2019-02-18 12:00:00	1	-1	Boston	MA

	date_time	max_temp	min_temp	city	state
237	2019-01-09 15:00:00	16	10	Los Angeles	CA
278	2019-01-14 18:00:00	12	10	Los Angeles	CA
505	2019-02-12 03:00:00	16	7	Los Angeles	CA

	date_time	max_temp	min_temp	city	state
576	2019-02-21 00:00:00	11	5	San Francisco	CA
282	2019-01-15 06:00:00	11	10	San Francisco	CA
350	2019-01-23 18:00:00	12	7	San Francisco	CA
495	2019-02-10 21:00:00	1	-5	Boston	MA
573	2019-02-20 15:00:00	1	-10	Boston	MA
556	2019-02-18 12:00:00	1	-1	Boston	MA
237	2019-01-09 15:00:00	16	10	Los Angeles	CA
278	2019-01-14 18:00:00	12	10	Los Angeles	CA
505	2019-02-12 03:00:00	16	7	Los Angeles	CA

Figure 7.1 Graphical depiction of using `pd.concat` to join separate data frames into a single one

Put that all together, and we have our loading code:

```
df = pd.concat(all_dfs)
```

Now that we have created our five-column data frame with information from all eight cities, we can start to tackle the questions I raised in the exercise. First, I asked whether the data for each city and state starts at roughly the same time. How can we know such a thing? Well, each row has a `date_time` column indicating when the temperature readings were taken. If we can get the minimum and maximum values for each city's rows, we can do a quick comparison.

This, of course, is precisely what `groupby` was designed to do: take a data frame and run an aggregation method (e.g., `min` or `max`) for each of the distinct values in one column.

However, there's a twist. Although we could group by city alone, we're going to group by two different columns: first `state` and then `city`. Why not just `city`? Because several of the city names appear twice. If we grouped results only by city, the information from Springfield, Illinois would be mixed up with that from Springfield, Massachusetts. Also, grouping by both state and city ensures that we get a nice report of our data. The query looks like this:

```
(
    df.groupby(['state', 'city'])['date_time'].min()
    .sort_values()
)
```

In this code, we tell pandas that we want to get the minimum value of `date_time` for each distinct combination of `state` and `city`. We then want to sort the values so we can easily find the earliest one—as well as find out if they're all from the same period of time. We can similarly run `max` on the values, to find the highest one:

```
(
    df.groupby(['state', 'city'])['date_time'].max()
    .sort_values()
)
```

In running these queries, we see that all the data files are from the same period, starting on December 11, 2018, and going through March 11, 2019. As we'll see in chapter 9, pandas allows us to work with actual dates and times, performing calculations and comparisons on them. Here, the `date_time` column is a string, which makes it possible to do some basic queries, but nothing as sophisticated as what we can do with `timestamp` objects, as you'll see.

I then asked you to find the lowest minimum temperature recorded for each city in our data set. Again, we run a `groupby` query, but this time we're interested in the actual values, not just in comparing them with one another. The minimum temperature is located in the `min_temp` column. So if we want to get the lowest minimum temperature for each city-state combination, we can say

```
df.groupby(['state', 'city'])['min_temp'].min()
```

This returns a series in which the index is the combination of state and city and the values are the minimum temperatures in each city. We can see that the data was taken in the winter, given how many of the temperatures are below 0 Celsius.

Finally, I asked you to find the highest maximum temperature recorded during this period, but on a per-state basis rather than a per-city basis. This means grouping just by `state`:

```
df.groupby('state')['max_temp'].max()
```

Sure enough, we get the maximum temperature for each state. Notice that because we have eight cities but that they're spread across only four states, we get four results rather than eight. The number of results we get from a grouping action reflects the number of unique values in the grouping column (or columns).

Of course, we can also use the `agg` method to ask for both results:

```
(
    df.groupby(['state', 'city'])['date_time']
    .agg(['min', 'max'])
)
```

Solution

```
import glob

all_dfs = []                                                ①

for one_filename in glob.glob('../data/*,*.csv'):           ②
    print(f'Loading {one_filename}...')

    city, state = (
        one_filename
        .removeprefix('../data/')
        .removesuffix('.csv')
        .split(',')                                         ③
    )

    one_df = (
        pd
        .read_csv(one_filename,
                  usecols=[0, 1, 2],                        ④
                  names=['date_time',
                         'max_temp',
                         'min_temp'],                       ⑤
                  header=0)                                 ⑥
        .assign(city=city.replace('+', ' ').title(),        ⑦
                state=state.upper())                        ⑧
    )

    all_dfs.append(one_df)                                  ⑨
df = pd.concat(all_dfs)                                     ⑩

df.groupby(['state', 'city'])[
    'date_time'].min().sort_values()                        ⑪

df.groupby(['state', 'city'])[
    'date_time'].max().sort_values()                        ⑫

df.groupby(['state', 'city'])['min_temp'].min()             ⑬
df.groupby('state')['max_temp'].max()                       ⑭
```

① Creates an empty list

② Uses glob.glob to get all filenames matching this pattern, and iterates over them

③ Uses str.split to get separate variables

④ We only care about the first three columns in each CSV file.

⑤ Assigns names to the three columns we loaded

⑥ The file's first row (index 0) contains headers.

⑦ Adds a city column to the data frame

⑧ Adds a state column to the data frame

⑨ Appends the new data frame to all_dfs

⑩ Creates one data frame from each of the city-specific data frames

⑪ Gets the earliest value of date_time for each city and state

⑫ Gets the latest value of date_time for each city and state

⑬ Gets the minimum temperature for each city

⑭ Gets the maximum temperature for each state

You can explore a version of this in the Pandas Tutor at **http://mng.bz/ddQO**.

Beyond the exercise

- Run `describe` on the minimum and maximum temperature for each state-city combination.
- Running `describe` works, but we only see the first and last few rows from each result. Using `pd.set_option` to change the value of `display_max_rows` makes it possible to see all the results in Jupyter. Then reset the option to 10 rows.
- What is the average difference in temperature (i.e., max – min) for each of the cities in our data set?

Window functions

Let's assume that a data frame contains sales information for last year:

```
df = DataFrame({'sales':[100, 150, 200, 250,
                200, 150, 300, 400,
                500, 100, 300, 200],
               'quarters':'Q1 Q2 Q3 Q4'.split()_ 3})
```

We've already seen how we can evaluate the data here a few different ways:

- We can get the mean (and other aggregate information) for all sales quarters by applying `mean` to the `sales` column.
- We can use `groupby` on the `quarters` column and then run `mean` on the `DataFrameGroupBy` object we get back to find out how well we did, on average, in each quarter.

These are important, common, and useful analyses. But what if we want to determine how much we sold, total, through the current quarter? That is, we want to know how much we sold in Q1, then in Q1+Q2, then Q1+Q2+Q3, and so on, until the final result is `df['sales'].sum()`.

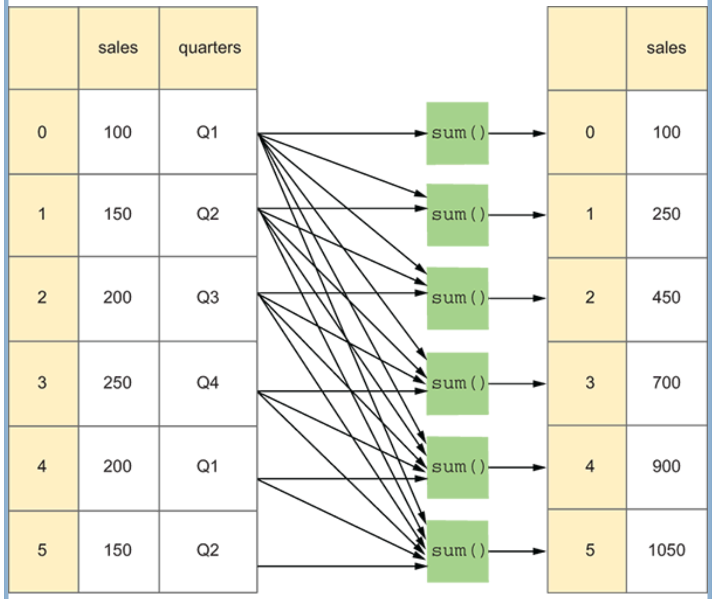
To perform this kind of operation, pandas provides *window functions*. There are several different types of window functions, but the basic idea is that they allow us to run an aggregate function, such as `mean`, on subsections of our data frame.

What I described earlier—that we would like to know, for each quarter, how much revenue we had through that quarter—is a classic example of a window function. This is known as an *expanding window* because we run the function with an ever-expanding number of lines—first one line, then two, then three . . . all the way up to the entire data frame.

For example, we could run

```
df['sales'].expanding().sum()
```

This returns a series whose values are the cumulative sum of values in `sales` up to that point. Because the first four values in the `sales` column are 100, 150, 200, and 250, the output of our call to `expanding` is 100, 250, 450, and 700.



Graphical depiction of an expanding window function with `sum`

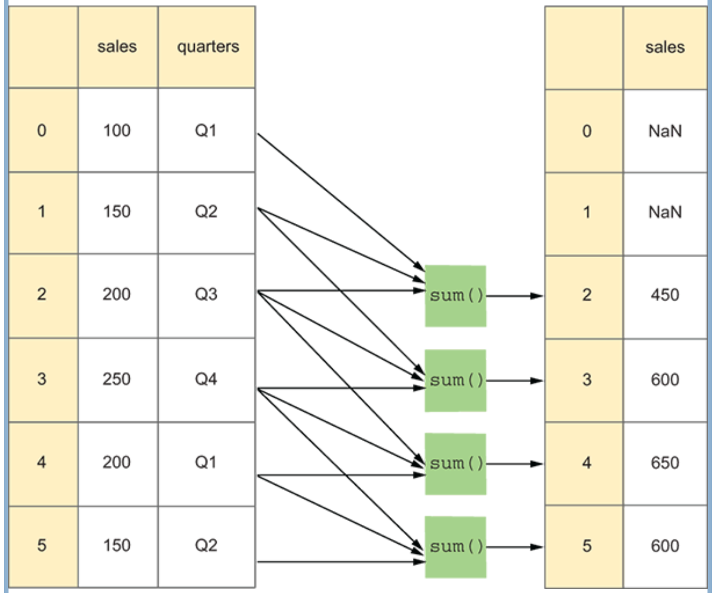
Perhaps we don't want to get a cumulative total, but rather a running average of how much we've sold per quarter. We can run `mean` or any other aggregation method:

```
df['sales'].expanding().mean()
```

In this case, the output from `expanding` is 100, 125, 150, and 175.

We can also use a *rolling* window function. In this case, we determine how many rows are considered part of the window. For example, if the window size is 3, we run the aggregation function on row index 0-2, then 1-3, then 2-4, and so on, until we get to the end of the data frame. For example, if we want to determine the mean of rows that are close to each other, we can do it as follows:

```
df['sales'].rolling(3).mean()
```



Graphical depiction of a rolling window function (looking at three lines) with `sum`

In this code, `rolling` is how we indicate that we want to run a rolling window function, and the argument `3` indicates that we want three rows in each window. We thus invoke `mean` on rows 0-2, then 1-3, then 2-4, then 3-5, and so on. The series we get back from this call puts the result of `mean` in the same location as the third (and final) row in our rolling window. This means row indexes 0 and 1 have `NaN` values.

A third type of window function is `pct_change`. When we run this on a series, we get back a new series with `NaN` at row index 0. The remaining rows indicate the percentage change from the previous row to the current one:

```
df['sales'].pct_change()
```

For example, the output from this code is

```
0          NaN
1     0.500000
2     0.333333
3     0.250000
```

The result is calculated as (`later_row` - `earlier_row`) / `earlier_row`:

- Index 0 is always `NaN`.

- Index 1 is the result of calculating (150 – 100) / 100.
- Index 2 is the result of calculating (200 – 150) / 150.
- Index 3 is the result of calculating (250 – 200) / 200.

`pct_change` is great for finding how much values go up or down from row to row. If we want to get the raw changes across rows, rather than the percentage changes, we can use the `diff` method, instead.

Exercise 33 • SAT scores, revisited

Back in exercise 22, we looked at SAT scores. There have long been accusations that the SAT isn't a fair test for college admissions, because wealthier students generally do better than poorer students. Given the data we have about the SAT, can we conclude that wealthier students do indeed, on average, score better? We will examine the math portion of the SAT, seeing if we can see any such problems in the data.

Here's what I would like you to do:

1. Read in the scores file (**sat-scores.csv**). This time, you want the following columns: `Year`, `State.Code`, `Total.Math`, `Family Income.Less than 20k.Math`, `Family Income.Between 20-40k.Math`, `Family Income.Between 40-60k.Math`, `Family Income.Between 60-80k.Math`, `Family Income.Between 80-100k.Math`, and `Family Income.More than 100k.Math`.
2. Rename the income-related column names to something shorter. I recommend `income<20k`, `20k<income<40k`, `40k<income<60k`, `60k<income<80k`, `80k<income 100k`, and `income>100k`.
3. Find the average SAT math score for each income level, grouped and then sorted by year.
4. For each year in the data set, determine how much better each income group did, on average, than the next-poorer group of students. Do you see (just by looking at the data) any income group that did worse, in any year, than the next-poorer students?
5. Which income bracket, on average, had the greatest advantage over the next-poorer income bracket?
6. Can we find, in a calculated and automated way, which income levels consistently (i.e., across all years) do worse than the next-poorest group?

Working it out

In this exercise, we use data to gain insight into a real-world problem. (What we do with this analysis is another question entirely.) For starters, we need to load data from our CSV file into a data frame. We're only interested in the math scores—but actually, we're more interested in the math scores when broken down by family income. As a result, we load the CSV file as follows:

```
df = pd.read_csv(filename,
      usecols=['Year', 'State.Code', 'Total.Math',
               'Family Income.Less than 20k.Math',
               'Family Income.Between 20-40k.Math',
               'Family Income.Between 40-60k.Math',
               'Family Income.Between 60-80k.Math',
               'Family Income.Between 80-100k.Math',
               'Family Income.More than 100k.Math'])
```

What I find particularly interesting here is what we *don't* include in the call to `pd.read_csv`: first and foremost, we don't assign an index. Although it's often useful to set an index, the analyses we do here all use grouping. And although we can still use `groupby` on a column we've set to be the index, there's no added value. For that reason, we stick with the default numeric index starting at 0.

I also asked you to change the names of the columns from long, unwieldy names to something easier to type and read. In theory, we could do so by giving a value to the `name` parameter. But if we give names to columns, we need to use integers to indicate which columns should be imported from CSV. And to be honest, I always find that hard to read, debug, and understand.

So instead, we load the columns with their full, original names, as per the file. We then change the column names by assigning to `df.columns`:

```
df.columns = ['Year', 'State.Code', 'Total.Math',
              'income<20k',
              '20k<income<40k',
              '40k<income<60k',
              '60k<income<80k',
              '80k<income<100k',
              'income>100k',
              ]
```

However, in some older versions of pandas, assigning to `df.columns` this way runs the risk of getting the order wrong. As a result, it's better to rename columns using `df.rename`, passing the `columns` keyword argument a dict value in which the keys are the old column names and the values are the new ones:

```
df = df.rename(
    columns={
    'Family Income.Less than 20k.Math':'income<20k',
    'Family Income.Between 20-40k.Math':'20k<income<40k',
    'Family Income.Between 40-60k.Math':'40k<income<60k',
    'Family Income.Between 60-80k.Math':'60k<income<80k',
    'Family Income.Between 80-100k.Math':'80k<income<100k',
    'Family Income.More than 100k.Math':'income>100k'
    })
```

Now that our data frame has the rows and columns we want and the columns have easy-to-understand names, we can start to analyze things. First, I asked you to find the average SAT math score for each income level, grouped and then sorted by year:

```
df.groupby('Year').mean(numeric_only=True).sort_index()
```

This query is similar to what we've done before: we want to invoke `mean` on every column in `df`, grouping the results by year. We can thus say, for each income bracket, what the average SAT math score was across the United States in each year.

Because we're grouping by the `Year` column, it isn't included in our output. But why isn't `State.Code` included in the output? Because we pass `numeric_only=True`, thus removing any non-numeric columns. In previous versions of pandas, non-numeric columns were silently ignored. Now, however, we need to either explicitly choose numeric columns or ask `mean` to do it for us with this keyword argument.

Moreover, because we group by `Year`, the index of the resulting data frame has an index of `Year`. Because the data set comes sorted by `Year`, the results appear to be sorted. But just to be on the safe side, we invoke `sort_index` on the data frame, ensuring that the result we get back is sorted from the earliest year in the data set through the final year in the data set.

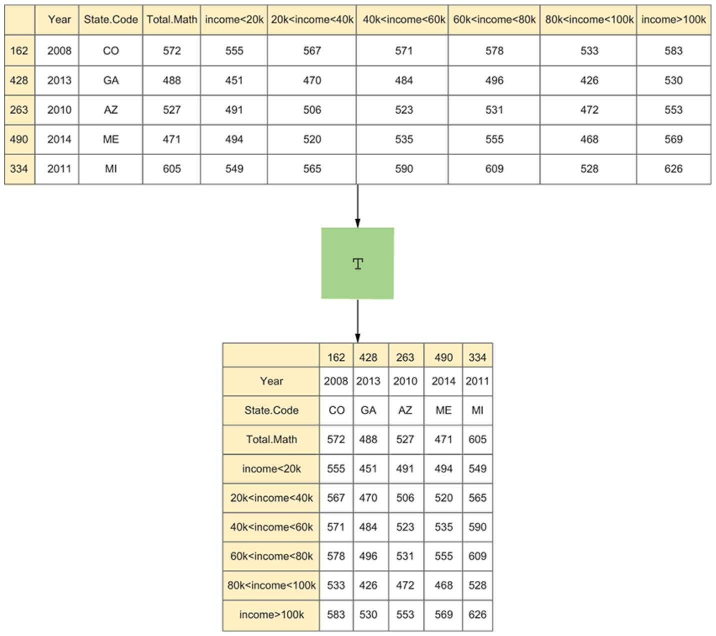
But then I asked you to do something else: to find how much *better* each income bracket did than the next-poorer income bracket. That is, first find the average SAT math score for students in the lowest bracket: `income<20k`. Then determine how much better (or worse) the next bracket (i.e., `20k<income<40k`) did. Perhaps we'll see that there's a negligible difference between them, in which case we can say, to some degree, that SAT scores aren't correlated with student income.

How can we make this comparison? We use `pct_change`, described in the "Window functions" sidebar.

We want to compare the scores by year and income brackets. But `pct_change` works on rows, not columns—and right now, our data frame has the brackets as columns. We thus need to flip the data frame on its side so the years are the columns and the income brackets are the rows.

The solution is to use the `transpose` method, more easily abbreviated as `T`, which returns a new data frame in which the rows and columns have exchanged places (figure 7.2):

```
df.groupby('Year')[['income<20k',
            '20k<income<40k',
            '40k<income<60k',
            '60k<income<80k',
            '80k<income<100k',
            'income>100k']].mean().T
```



	Year	State.Code	Total.Math	income<20k	20k<income<40k	40k<income<60k	60k<income<80k	80k<income<100k	income>100k
162	2008	CO	572	555	567	571	578	533	583
428	2013	GA	488	451	470	484	496	426	530
263	2010	AZ	527	491	506	523	531	472	553
490	2014	ME	471	494	520	535	555	468	569
334	2011	MI	605	549	565	590	609	528	626

	162	428	263	490	334
Year	2008	2013	2010	2014	2011
State.Code	CO	GA	AZ	ME	MI
Total.Math	572	488	527	471	605
income<20k	555	451	491	494	549
20k<income<40k	567	470	506	520	565
40k<income<60k	571	484	523	535	590
60k<income<80k	578	496	531	555	609
80k<income<100k	533	426	472	468	528
income>100k	583	530	553	569	626

Figure 7.2 Example of using `T` to transpose a data frame

T, a shortcut for “transpose”

The `transpose` method is invoked like any other method in pandas, using parentheses:

```
df.transpose()
```

Its convenient alias, `T`, is *not* a method and thus should not be invoked with parentheses:

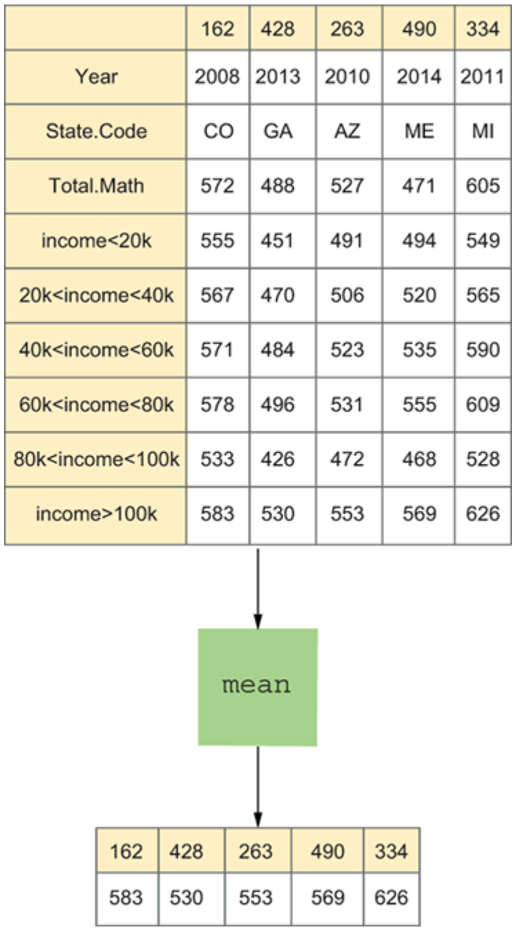
```
df.T
```

In both cases, we get a new data frame back; the original data frame is unmodified.

We can now invoke `pct_change` on this new data frame:

```
(
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
)
```

We get back a data frame in which the columns are years (2005 to 2015) and the rows are income brackets. The values in the data frame are floats, with each number indicating the percentage by which the math scores for that income bracket, in that year, differed from the next-poorer income bracket. The lowest income bracket has `NaN` values, because there is no previous row (figure 7.3).



	162	428	263	490	334
Year	2008	2013	2010	2014	2011
State.Code	CO	GA	AZ	ME	MI
Total.Math	572	488	527	471	605
income<20k	555	451	491	494	549
20k<income<40k	567	470	506	520	565
40k<income<60k	571	484	523	535	590
60k<income<80k	578	496	531	555	609
80k<income<100k	533	426	472	468	528
income>100k	583	530	553	569	626

162	428	263	490	334
583	530	553	569	626

Figure 7.3 Get the mean after transposing.

From a visual scan of the data, we can see that most income brackets did better than the next-lower bracket. Thus, families with an income between $20,000 and $40,000 per year did about 3% to 7% better on their math SAT than people in the lowest bracket. And in families making $40,000 to $60,000 per year, they generally did 2% to 3% better than those in the next-lower bracket.

However, across the years, those earning between $80,000 and $100,000 per year did slightly worse than those than those in the next-lowest income bracket (i.e., between $60,000 and $80,000 per year). What's the reason for this? I'm not sure, but it is consistently true across all the years.

Next, I asked you to determine which income bracket, on average, had the greatest advantage over the next-poorer income bracket. To do this, we start with the result of our call to `pct_change`. But we want to determine how much better, on average, each bracket did than the next-poorer bracket. To do this, we use `mean`—but not on the data frame we get back from `pct_change`. Rather, we retranspose the data frame such that the income brackets are the columns and the years are the rows:

```
(
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
    .T
    .mean()
)
```

Changing the axis

Another option would be to pass `mean` the `axis` keyword argument:

```
df.mean(axis='columns')
```

The default value for `axis` is `'rows'`, giving us a new row with the mean from each column. If we pass `axis='columns'`, we get a new column back with the same index as the data frame.

If the data set isn't too large, I'm fine with transposing twice, which I see as a way to return to the earlier state. But if you feel more comfortable passing the `axis` keyword argument, or if your data set is large enough that transposing will take too much time or memory, you can try that.

We now know how much each income bracket did better, on average, than the next-poorer bracket. Where was the greatest jump in SAT math performance? We can find out by invoking `sort_values` and asking for the values to be in descending order. Then we can invoke `head()` to see the top-ranking income brackets:

```
(
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
    .T
    .mean()
    .sort_values(ascending=False)
    .head()
)
```

All this is fine, but relying on a visual scan of the data is not a good way to go about things. Rather, we'd like an automated way to find which, if any, of the income brackets did worse than the next-lower bracket. How can we do that?

Well, we know that the result of calling `pct_change` is a data frame. As such, we have all our pandas analysis tools at our disposal. We can, for example, assign the result of `pct_change` to a data frame and then look for values that are ≤ 0:

```
change = (
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
)
change <= 0
```

We're applying a comparison operator to a data frame, which means we get back a boolean data frame. Just as applying a boolean series to a series only shows the elements corresponding to `True` values, applying a data frame to a boolean data frame shows the items corresponding to `True` values. The difference is that the data frame has the same shape—and thus any filtered-out values are replaced with `NaN`:

```
change[change <= 0]
```

We can then remove any rows that contain any `NaN` values, showing only rows in which we consistently see a change for the worse as the income level rises:

```
change[change <= 0].dropna()
```

Sure enough, we see that every single income bracket did better, on average, than the income bracket below it.

Solution

```
filename = '../data/sat-scores.csv'

df = pd.read_csv(filename,
 usecols=['Year', 'State.Code', 'Total.Math',
      'Family Income.Less than 20k.Math',
      'Family Income.Between 20-40k.Math',
      'Family Income.Between 40-60k.Math',
      'Family Income.Between 60-80k.Math',
      'Family Income.Between 80-100k.Math',
      'Family Income.More than 100k.Math'])                    ①

df = df.rename(
    columns={
    'Family Income.Less than 20k.Math':'income<20k',
    'Family Income.Between 20-40k.Math':'20k<income<40k',
    'Family Income.Between 40-60k.Math':'40k<income<60k',
    'Family Income.Between 60-80k.Math':'60k<income<80k',
    'Family Income.Between 80-100k.Math':'80k<income<100k',
    'Family Income.More than 100k.Math':'income>100k'
    })                                                           ②

df.groupby('Year').mean(
    numeric_only=True).sort_index()                              ③

(
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
)                                                                ④

(
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
    .T
    .mean()
    .sort_values(ascending=False)
    .head()
)                                                                ⑤

change = (
    df
    .groupby('Year')
    [['income<20k',
      '20k<income<40k',
      '40k<income<60k',
      '60k<income<80k',
      '80k<income<100k',
      'income>100k']]
    .mean()
    .T
    .pct_change()
```

```
)                                    ⑥

change[change <= 0].dropna()         ⑦
```

① Reads data from the CSV file

② Renames the columns as per the dict, with old names as the keys and new names as the values

③ Calculates the mean value of each column for each year and then sorts by year

④ Transposes the result of grouping and getting the mean, and then uses pct_change to check how much better each income group did than the previous one

⑤ Which income bracket had the greatest advantage over the next-highest income bracket?

⑥ Assigns the previous output to a variable, change

⑦ Finds all rows of change in which all columns did worse than the previous value

You can explore a version of this in the Pandas Tutor at http://mng.bz/rj9D.

Beyond the exercise

- Calculate descriptive statistics for all the changes in income brackets. Where do you see the largest difference between income brackets?
- Which five states have the greatest gap in SAT math scores between the richest and poorest students?
- You analyzed math scores. If you perform the same analysis on verbal SAT scores, will you similarly see that wealthier students generally do better than poorer students? Do any income brackets do worse than the next-poorer bracket?

Filtering and transforming

We've already seen how we can use `groupby` to run aggregate methods on each portion of our data to get the average rainfall per city or the total sales figures per quarter. We've also seen, in earlier chapters, how to use a boolean index to filter out rows that fail to match particular criteria.

For example, consider a data frame containing the year-end math scores for each student. The rows of the data frame describe the students. The columns of the data frame, `name`, `year`, and `score`, describe those three student attributes. Here's how we can create a simple form of this data frame:

```
import numpy as np
np.random.seed(0)
```

```
df = DataFrame({'name': list('ABCDEFGHIJ'),
                'year': [2018, 2019, 2020]_ 3 + [2021],
              'score':np.random.randint(80, 100, 10)})
```

Our data frame is

```
name    year  score
0    A  2018     92
1    B  2019     95
2    C  2020     80
3    D  2018     83
4    E  2019     83
5    F  2020     87
6    G  2018     89
7    H  2019     99
8    I  2020     98
9    J  2021     84
```

We can perform a number of calculations:

- We can get the mean score by running `df['score'].mean()`. This returns a single floating-point value, 89.0.
- We can get all the students who scored above 90 with `df.loc[df['score'] > 90]`. This returns the original data frame minus students who got less than 90—in our case, row indexes 0, 1, 7, and 8.
- We can get the mean score per year by running `df.groupby('year') ['score'].mean()`. If the school has eight grades, the result of this query is a series whose index contains the distinct values of `year` from `df` and whose values are the average grades for each year. Here, we get four different results (one for each year).

So far, so good. But consider this: we want to determine which years in our school had an average score of at least 90, and see all the students in those years. We want to filter out specific groups of students based on a per-year aggregate calculation. How can we do that?

The answer, it turns out, is to apply the `filter` method to our `DataFrameGroupBy` object. All we need is to pass `filter` a function that, given a group of rows, returns either `True` or `False`, to indicate whether those rows should be in the result data frame.

In other words,

- We want to decide whether to include or exclude rows based on the `year`, so we run `df.groupby('year')`
- On that `DataFrameGroupBy` object, we run the `filter` method.
- `filter` takes a function as an argument.
- The function we pass is invoked once per group. It receives a data frame—a subset of `df`—as its argument.

- The function must return `True` or `False` to indicate whether rows from that group should be included or excluded in the resulting data frame.
- The function can be a full-fledged Python function (i.e., one defined with `def`), or we can use `lambda` for an inline, anonymous function.

Here's an example of such a function, as well as how we could invoke it:

```
def year_average_is_at_least_90(df):
    return df['score'].mean() > 90

df.groupby('year').filter(year_average_is_at_least_90)
```

The result of running this code is a data frame whose rows all come from `df`, from years in which the average final-exam math score was at least 90. That is only the year 2019, so we get the rows with indexes 1, 4, and 7.

Here are some examples of how to use `filter` in real-world data sets:

- Show all products coming from factories that brought in more than $1 million last year.
- List the staff working for divisions with below-average salaries.
- Find networks whose segments have had more than 10 outages in the last week.

Another, related method we can use on a `GroupBy` object is `transform`. In this case, the point is not to remove rows from the original data frame but rather to transform them in some way. For example, let's say we want to turn the score into a percentage expressed as a float. We can say

```
df.groupby('year')['score'].transform(lambda x: x/100)
```

In this example we're grouping by year, so the function is run once for each year:

- It's invoked with a three-element series with all rows from 2018.
- It's invoked with a three-element series with all rows from 2019.
- It's invoked with a three-element series with all rows from 2020.
- It's invoked with a one-element series with the only row from 2021.

The function is expected to return a series with the same dimensions as the input, which happens naturally in our example because our `lambda` function invokes the division (`/`) operator on the series. Thanks to broadcasting (i.e., that an operation on a series and a scalar is repeated on each element of the series), we're guaranteed to get a result of the correct dimensions. We can then replace the original `score` column with our transformed column:

```
df['score'] = (
    df.groupby('year')['score']
    .transform(lambda x: x/100)
    )
```

But we can do much more than this. After all, our `lambda` function has access to all the rows from each year. This means we can run aggregate functions, such as `sum` or `mean`. For example, let's say that we pass `np.max` as our function:

```
df.groupby('year')['score'].transform(np.max)
```

We want to invoke our function (`np.max`) once for each value of `year` in the data frame. And the input to our function is the column `score`, with the rows for each year. The result is as follows:

```
0    92
1    99
2    98
3    92
4    99
5    98
6    92
7    99
8    98
9    84
Name: score, dtype: int64
```

In the resulting series, the value in each row is the highest value of `score` from that particular year. In other words, we have replaced every score with the maximum score for that year. (This is probably not the best way to evaluate students, I'll admit.)

We can then assign the transformed row back to our data frame:

```
df['score'] = df.groupby('year')['score'].transform(np.max)
```

As you can see, the grouped version of `transform` is useful when we want to transform values in a data frame on a group-by-group basis, much as the grouped version of `filter` is useful when we want to filter values on a group-by-group basis.

Here are some ways to use `transform` with real-world data sets:

- Find the difference between each value in a group and the group's mean.

- Find the proportion that each value in the group has versus the group’s sum.
- Calculate the z-score (i.e., the number of standard deviations) that each value is from its group’s mean.

NOTE In the case of both `filter` and `transform`, an attribute `name` is added to the `df` parameter with the name of the current group.

NOTE The `filter` method for `GroupBy` is very similar to Python’s builtin `filter` function, and the `transform` method for `GroupBy` is very similar to Python’s builtin `map` function. They work differently because they’re acting on data frames rather than simple iterables, but the usage is similar.

Exercise 34 • Snowy, rainy cities

One constant theme, wherever I’ve lived, is that people complain about the weather. In a hot climate, people complain that it’s too hot. In a cold climate, people complain that it’s too cold. In a city with hot summers and cold winters, they complain about both. And, of course, people tell visitors and newcomers that their city’s weather is worse than anywhere else. There isn’t much that we can do about people’s complaints. But maybe we can use data to determine which city does indeed have the most extreme weather. Because, you know, if someone is complaining about the weather, they want nothing more than to be corrected with hard data.

The calculations we’ll make in this exercise all take advantage of the `filter` and `transform` methods on `DataFrameGroupBy` objects. These methods allow us to conditionally keep (`filter`) and modify (`transform`) rows in a data frame while having access to all rows of the group when deciding and calculating.

NOTE The `DataFrameGroupBy` versions of `filter` and `transform` are, in my experience, among the most complex pieces of functionality that pandas provides. It may take you a while to think through what calculation you want to perform and then find the right way to express it in pandas.

In this exercise, I want you to

1. Read in the data frames for the city weather as in exercise 32, reading three columns: `max_temp`, `min_temp`, and `precipMM`.
2. Determine which cities had, on at least three occasions, precipitation of 15 mm or more.
3. Find cities that had at least three measurements of 10 mm of precipitation or more when the temperature was at or below 0° Celsius.
4. For each precipitation measurement, calculate the proportion of that city’s total precipitation.
5. For each city, determine the greatest proportion of that city’s total precipitation to fall in a given period.

Working it out

In this exercise, we use `filter` and `transform` on `DataFrameGroupBy` objects to work with rows according to their aggregate properties. We start by loading the weather data from six different cities, similarly to how we did it in exercise 32. We want to load three columns: `max_temp`, `min_temp`, and `precipMM` (i.e., the amount of precipitation that fell, in millimeters). Because it's so similar to what we did before, I'll show the code here without comment:

```
import glob

all_dfs = []

for one_filename in glob.glob('../data/*,*.csv'):
    print(f'Loading {one_filename}...')

    city, state = (
        one_filename
        .removeprefix('../data/')
        .removesuffix('.csv')
        .split(',')
    )

    one_df = (
        pd
        .read_csv(one_filename,
                  usecols=[0, 1, 2],
                  names=['max_temp',
                         'min_temp',
                         'precipMM'],
                  header=0)
        .assign(city=city.replace('+', ' ').title(),
                state=state.upper())
    )

    all_dfs.append(one_df)

df = pd.concat(all_dfs)
```

Once we have our data frame in place, we can start to analyze it. For starters, we want to find cities that had measured precipitation of 15 mm or more on at least three occasions. This means

- We need to check measurements on a per-city basis (via `groupby`).
- We'll only keep cities that reported 15 mm of precipitation at least three times (via `filter`).

Let's start with our `groupby`. Because we want to find the precipitation on a per-city basis, you may think we should group by city name:

```
df.groupby('city')
```

However, we can’t do this, because there are two different cities with the name “Springfield”—one in Illinois and the other in Massachusetts. For that reason, we need to group not just by city but also by state. We can do so by passing a list of columns to `groupby` rather than just one:

```
df.groupby(['city', 'state'])
```

This gives us our `GroupBy` object, which we’ve previously used to apply aggregate functions on distinct subsets of our data. But here we’ll use the `groupby` object a different way, to include and exclude rows from `df` based on properties of their city and state. That is, we want to filter out rows, but we want to do it by group—such that for each group, all the rows are included or excluded. (You can think of this as the collective punishment feature of pandas.)

We do this by calling `filter` on our `GroupBy` object. `filter` on a `GroupBy` works on a group-by-group basis. The argument to `filter` is a function that expects to get a data frame as its argument. The function is called once for each group in the `groupby`, and the data frame passed to it is a subset of the original data frame, containing only those rows in the current group.

The function passed to `filter` should return `True` or `False`. If the function returns `True`, the rows from this subframe are kept. If the function returns `False`, the rows from this subframe are not included. Because its argument is a data frame with all the rows in the current group, `filter` can perform all sorts of calculations in determining whether to return `True` or `False`.

In this case, we want to preserve rows from cities that had 15 mm of precipitation on at least three occasions. Our function thus needs to determine whether the subframe it is passed contains at least three such rows. Our function can look like this:

```
def has_multiple_readings_at_least(mini_df):
    return mini_df.loc[
        mini_df['precipMM'] >= 15,
        'precipMM'
        ].count() >= 3
```

If we were to invoke this function on a data frame, it would return a single `True` or `False` value indicating whether the complete data frame had recorded at least 15 mm of precipitation on at least three occasions. By running it via `filter`, though, we can determine which cities had such records:

```
(
    df
    .groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least)
)
```

The result of this query is a subset of our original data frame. But my question to you wasn't which rows would pass the filter. Rather, I asked you which cities had such precipitation. One way to do this would be to retrieve just the `city` and `state` columns from the resulting data frame:

```
(
    df.groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least)
    [['city', 'state']]
)
```

However, this gives us the city and state for each row. That's more than we need. We can just run the `drop_duplicates` method on the result, instead:

```
(
    df
    .groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least)
    [['city', 'state']]
    .drop_duplicates()
)
```

This works and gives us the answer we want—namely, that only New York and Los Angeles had three occasions on which at least 15 mm of precipitation fell. However, if you've been programming for any length of time, the `has_multiple_readings_ at_least` function may seem odd. Do we really want to hard-code the values "15 mm" and "3 times" into the function? It may make more sense to write a generic function that can take additional arguments.

But how can we do that? After all, we're not calling `has_multiple_readings_ at_least` directly. Rather, we're passing it to `filter`, which calls the function on our behalf. There isn't an obvious way for us to pass arguments to our function when it's being invoked via `filter`.

Here, pandas does something clever: any additional arguments passed to `filter` are passed along to our function. This is done using the standard Python constructs of `*args` and `**kwargs`, for arbitrary positional and keyword arguments. (For a tutorial on this subject, check out my blog post at **https://lerner.co.il/2021/06/07/python-parameters-primer**.)

We can thus rewrite our function as follows:

```
def has_multiple_readings_at_least(mini_df, min_mm, times):
    return mini_df.loc[
        mini_df['precipMM'] >= min_mm,
        'precipMM'
        ].count() >= times
```

Now it looks more like a regular Python function, taking three arguments. The first is still the subframe that was passed before, containing all the rows in the current group. But the second two arguments are assigned indirectly, via `filter`, when it calls our function. We can then say

```
(
    df
    .groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least,
            min_mm=10,
            times=3)
    [['city', 'state', 'precipMM']]
    .drop_duplicates()
)
```

In this code, we call `filter` and pass it our function, `has_multiple_readings_at_ least`. In theory, we could then pass values for `min_mm` and `times` as positional arguments. But if we do that, we'll also have to pass a second positional argument to `filter`, called `dropna`. Rather than calling `filter(func, True, 10, 3)`, we call `filter (func, min_mm=10, times=3)`. This is an aesthetic choice, rather than a technical one, but I think it makes sense here.

Next, I asked you to find cities that had

- At least three measurements of 10 mm precipitation . . .
- . . . when the temperature was below 0° Celsius

We again use `groupby` and then `filter`, using a slightly modified version of our `has_ multiple_readings_at_least` function from before:

```
def has_multiple_readings_at_least(mini_df, min_mm, times):
    return mini_df.loc[
        ((mini_df['precipMM'] >= min_mm) &
         (mini_df['min_temp'] <= 0)),
        'precipMM'
        ].count() >= times
```

We can then perform our grouping and filtering in the following way:

```
(
    df
    .groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least, min_mm=10, times=3)
    [['city', 'state']]
    .drop_duplicates()
)
```

Next, I asked you to find the proportion of that city's precipitation that fell with each measurement. If our data frame contains two precipitation measurements for a given city, and we see that 3 mm fell on the first day and 7 mm fell on the second day, we want to find that 30% fell in the first measurement and 70% fell in the second.

In other words, we'll calculate one value for each row. But the value we calculate for each row will depend on an aggregate calculation for the row's group. It's precisely for these situations that pandas provides the `transform` method. Similar to what we did with `filter`, we'll pass a function as the first argument to `transform`. This function is invoked once per group, and the function is passed a series: the column we want to transform. The function must then return a series, of the same length and with the same index, as its argument.

Let's assume that we have a series of numbers, each representing one measurement of precipitation. What function can we write that will return a new series with the same length and index as the original, but whose values indicate the proportion of the whole? It may look like this:

```
def proportion_of_city_precip(s):
    return s / s.sum()
```

Our function takes a series `s` as input and then returns the result of dividing each row by the sum total of all rows. This is how we would do it if all the values were from the same city. How can we do it, then, if we have many different cities? That's part of the magic—the `groupby` version of `transform` takes care of it for us. The rows from each group are passed, one at a time, to the function `proportion_of_city_precip`. The return value is then a series in which the parallel rows from the input series have their new values. We can assign the resulting series back to the column from which it was transformed, add a new column to a data frame, or just save the transformed column.

The difference between the standard `transform` method and the `groupby` version of `transform` is that in the latter, we have access to the entire series and can thus make calculations using aggregation functions.

Here's how we can write this:

```
df['precip_pct'] = df.groupby('city')[
    'precipMM'].transform(proportion_of_city_precip)
```

Notice that, in this example, we assign the returned series to the data frame as a new column. With this column in place, we can then answer the final question for this exercise: for each city, what was the greatest proportion of that city's total precipitation to fall in a given period? In other words, which measurement reflected the greatest proportion of precipitation we measured?

To answer this question, we use a simple, classic `groupby`: we apply an aggregate function (`max`) to each city in our system. Of course, because we have a duplicate city name, we group on both city and state. That gives the following:

```
df.groupby(['city', 'state'])['precip_pct'].max()
```

Solution

```
import glob

all_dfs = []

for one_filename in glob.glob('../data/*,*.csv'):
    print(f'Loading {one_filename}...')

    city, state = (
        one_filename
        .removeprefix('../data/')
        .removesuffix('.csv')
        .split(',')
    )

    one_df = (
        pd
        .read_csv(one_filename,
                  usecols=[0, 1, 2],
                  names=['max_temp',
                         'min_temp',
                         'precipMM'],
                  header=0)
        .assign(city=city.replace('+', ' ').title(),
                state=state.upper())
    )

    all_dfs.append(one_df)                                  ①

df = pd.concat(all_dfs)                                     ②

def has_multiple_readings_at_least(mini_df):
    return mini_df.loc[
        mini_df['precipMM'] >= 15,
        'precipMM'
        ].count() >= 3                                      ③

(
    df
    .groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least)
    [['city', 'state']]                                     ④
    .drop_duplicates()                                      ⑤
)

def has_multiple_readings_at_least(mini_df, min_mm, times):
    return mini_df.loc[
        ((mini_df['precipMM'] >= min_mm) &
         (mini_df['min_temp'] <= 0)),
        'precipMM'
        ].count() >= times                                  ⑥
```

```
(
    df
    .groupby(['city', 'state'])
    .filter(has_multiple_readings_at_least, min_mm=10, times=3)
    [['city', 'state']]                                           ⑦
    .drop_duplicates()                                            ⑧
)

def proportion_of_city_precip(s):
    return s / s.sum()                                            ⑨

df['precip_pct'] = df.groupby('city')[
    'precipMM'].transform(proportion_of_city_precip)              ⑩

df.groupby(['city', 'state'])['precip_pct'].max()                 ⑪
```

① Appends, one by one, the data frames we load to a list

② Creates one data frame from all the loaded data frames

③ This function returns True if there are at least three rows with precipMM >= 15.

④ Grouping by city and state, we apply the filter to keep the rainiest cities.

⑤ Gets the unique combinations of city and state

⑥ This function returns True if precipitation of min_mm has fallen at least times times.

⑦ Uses the new version of has_multiple_readings_at_least to find rainiest cities

⑧ Gets the unique combinations of city and state

⑨ This function returns the proportion of a city's precipitation that fell in one reading.

⑩ Adds a new column, precip_pct, showing the proportion for each city

⑪ Finds the reading showing the greatest proportion of precipitation for that city

You can explore a version of this in the Pandas Tutor at **http://mng.bz/VRA0**.

Beyond the exercise

- Implement the first version of `has_multiple_readings_at_least`, which takes a single argument (`df`), but with `lambda`.
- Implement the second version of `has_multiple_readings_at_least`, which takes three arguments (`df`, `min_mm`, and `times`), but with `lambda`.

- Implement our transformation, but replace `proportion_of_city_precip` with a `lambda`. Then find the reading that represented the greatest proportion of rainfall for each city.

Exercise 35 • Wine scores and tourism spending

Earlier in this chapter, we used `join` to combine two data frames into a single one. In this exercise, we go deeper into uses for `join`, exploring how we can join more than two data frames, how we can combine joining with grouping, and the different types of joins we can perform. We'll also look for correlations among our joined data sets.

This time, we'll combine several data sets to answer a question I'm sure you've often thought about: does a country that spends more on tourism also make better wines? Our data will come not only from the OECD tourism data we've previously explored but also from more than 150,000 rankings of wines.

To perform this analysis, I'd like you to do the following:

1. Create a data frame, `oecd_df`, from **oecd_locations.csv**, containing a subset of all OECD countries. The resulting data set should have a single column called `country`. The index should be based on the country's abbreviation.
2. Create a second data frame, `oecd_tourism_df`, from **oecd_tourism.csv**. You're only interested in four columns: `LOCATION` (which will serve as the index), `TIME` (containing the year in which the measure was taken), `SUBJECT` (the type of spending), and `Value` (the amount spent in each year). You're also only interested in rows where `SUBJECT` has the value `'INT-EXP'`, meaning spending. Once you've kept only the rows with `'INT-EXP'`, you can remove the `SUBJECT` column.
3. Create a new series, `tourism_spending`, in which the index reflects the country names (i.e., not abbreviations) and the value contains the average tourism spending for that country.
4. Create a third data frame, `wine_df`, based on **winemag-150k-reviews.csv**. You only need two columns: `country` and `points`.
5. Get the mean wine score for each country, across all wine reviews, sorted in descending order.
6. Perform a standard join between the average wine scores per country and the average tourism spending per country. Where do you see `NaN` values? What do those `NaN` values mean?
7. Perform an outer join between the average wine scores per country and the average tourism spending per country. Where do you see `NaN` values? What do they mean now?
8. Find the correlation between average wine score and average tourism spending. What can you say about these two values? Is there any correlation?

Working it out

This exercise is meant to demonstrate how we can bring together many of the ideas we've seen in this chapter on a grander scale—joining multiple data frames, moving between series and data frames, and even finding correlations across different data sets. The first thing I asked you to do was create `oecd_df`, a data frame with a subset of OECD members. The input CSV file, as we saw in exercise 31, contains just two columns and doesn't have any headers, which means we need to set the column names to `abbrev` and `country`. I asked you to set the input data frame's index column to be `abbrev`. To do all this, we can use the following code:

```
oecd_df = pd.read_csv('../data/oecd_locations.csv',
                      header=None,
                      names=['abbrev', 'country'],
                      index_col='abbrev')
```

Let's take a look at `oecd_df.head()`:

```
abbrev  country
AUS     Australia
AUT     Austria
BEL     Belgium
CAN     Canada
DNK     Denmark
```

This data frame isn't that useful on its own. The point of loading this is to get a translation table between the country names (the `country` column) and the country abbreviations (the `abbrev` column). We will need the country names to work with the wine ratings, but we will need the country abbreviations to work with the tourism spending data. It's not uncommon to have such data frames when working with data from different sources.

With this data frame created and in place, we can create the second one, which we call `oecd_tourism_df`. This data frame comes from a CSV file that does have headers, so we don't need to name them. However, we are only interested in four of the input columns, so we need to select them using `usecols`. Then we use one column (`SUBJECT`) to keep only those rows that have to do with tourist expenses; once we're done with it, we drop it. Finally, I asked that you set the `LOCATION` column (i.e., the country abbreviation) as the index.

We can do all this with the following code:

```
oecd_tourism_df = (
    pd
    .read_csv('../data/oecd_tourism.csv',
              usecols=['LOCATION', 'TIME',
                       'Value', 'SUBJECT'],
              index_col='LOCATION')
    .loc[lambda df_: df_['SUBJECT'] == 'INT-EXP']   ①
    .drop('SUBJECT', axis='columns')          ②
)
```

① Keeps rows where the subject is 'INT-EXP'

② Removes the SUBJECT column

Notice that, in this code, we use `lambda` as an argument to `.loc`. Wherever the `lambda` expression returns `True`, the row from the original data frame is kept. We use `df_` as the parameter in the `lambda` expression to indicate that it's a temporary value and to ensure that we don't confuse it with `df`, which is often used for other data frames. Besides, when the `lambda` is being run, the data frame created by `read_csv` hasn't yet been assigned to a variable, so we need to give it a temporary name.

Once we're done using the `SUBJECT` column to keep only tourist expenses, we can remove it with `drop`. Don't forget that `drop` defaults to using the index; to drop one or more columns, we need to specify that with `axis='columns'`.

Here's the result of invoking `oecd_tourism_df.head()`:

```
LOCATION  TIME    Value
AUS       2008  27620.0
AUS       2009  25629.6
AUS       2010  31916.5
AUS       2011  39381.5
AUS       2012  41632.8
```

We now have two data frames, both of which use the same country abbreviations for their indexes. Never mind that in `oecd_tourism_df`, the index contains repeat values, whereas in `oecd_df`, the index contains unique values; the join system knows what to do in such cases and will handle things just fine. The key (no pun intended) thing here is that the two data frames' indexes contain the same elements. (What happens if one or both of them contains values that aren't in the other? We'll deal with that later in this exercise.)

I next asked you to find the mean tourist spending per country in the OECD subset. That is, we have tourist spending figures from a number of different OECD countries across several years. We want to determine how much each country spent on tourism, on average, across all years in the data set. Moreover, we want the results to show the countries' names, not their abbreviations.

Finding the mean tourist spending per country across all years is a classic use of grouping. We could, for example, do it as follows:

```
oecd_tourism_df.groupby('LOCATION')['Value'].mean()
```

This code says that we want to get the mean of the `Value` column for each distinct `LOCATION`. (Notice that even though `LOCATION` is now the index of this data frame, we can still use it for grouping.) However, we don't want `LOCATION`, containing the country abbreviations. Rather, we want to use the country names, which are in `oecd_df`.

We thus need to join these two data frames. Both use the abbreviations as an index, which makes this possible. (It doesn't matter that the columns have different names; joining typically works on the data frames' indexes.) When we join, we basically say that we want to create a new, wider data frame containing all the columns from the first and all the columns of the second, with the indexes overlapping. So the resulting data frame has a total of four columns: an index containing the location abbreviations, as before, a `country` column (from `oecd_df`), and `TIME` and `Value` columns (from `oecd_tourism_df`). The left and right sides are joined wherever the index of `oecd_df` matches the index of `oecd_tourism_df`, which means it's not a problem to have repeated values in the indexes of one or both data frames.

We can join them this way:

```
oecd_df.join(oecd_tourism_df)
```

We invoke `join` on `oecd_df`, which is seen as the left data frame, and we pass `oecd_ tourism_df` as an argument to `join`. It is, of course, the right data frame in the join. The result is a new data frame. We run `groupby` on this data frame, grouping by `country` —the full names of the countries we're looking at. We then retrieve only the `Value` column and calculate the mean:

```
(
    oecd_df
    .join(oecd_tourism_df)
    .groupby('country')['Value'].mean()
)
```

This way, we've again calculated and retrieved the mean tourism spending, per country, over all years in the data set. But the result we get back uses the full country names, rather than the abbreviations. Moreover, because the result has an index (country names) and a single value column, it's returned as a series, rather than as a data frame. I asked you to assign the resulting series to a variable, `tourism_spending`, for easier manipulation later:

```
tourism_spending = (
    oecd_df
    .join(oecd_tourism_df)
    .groupby('country')['Value'].mean()
)
```

Here is the result of invoking `tourism_spending.head()`:

```
country
Australia    36727.966667
Austria      11934.563636
Belgium      20859.883455
Brazil       21564.351833
Canada       40984.633333
Name: Value, dtype: float64
```

Now it's time to load our third CSV file into a data frame. In this case, we're only interested in two columns from the CSV file, `country` and `points`:

```
wine_df = pd.read_csv(
    '../data/winemag-150k-reviews.csv',
    usecols=['country', 'points'])
```

Here's the result of running `wine_df.head()`:

```
country   points
0   US         96
1   Spain      96
2   US         96
3   US         96
4   France     95
```

As soon as we've created this data frame, we want to calculate the mean score (`points`) that each country received. Once again, we can perform a grouping operation:

```
country_points = (
    wine_df
    .groupby('country')['points'].mean()
)
```

Here's the result of running `country_points.head()`:

```
country
Albania                         88.000000
Argentina                       85.996093
Australia                       87.892475
Austria                         89.276742
Bosnia and Herzegovina          84.750000
Name: points, dtype: float64
```

This returns a series in which the index contains the country names and the values are the mean points per country. We assign this to a variable, `country_points`, so we can use it in additional tasks.

The first task we want to do with it is to sort the average scores from highest to lowest. This can be done with a call to `sort_values`, passing `ascending=False` to ensure that we sort the values in descending order:

```
country_points.sort_values(ascending=False)
```

We get back a new series showing which countries had the highest average wine scores and which had the lowest. Here are the first five rows from my result:

```
country
England    92.888889
Austria    89.276742
France     88.925870
Germany    88.626427
Italy      88.413664
```

Now we come to the climax of this exercise: joining the wine scores and the tourism spending. How can we do that?

Well, it makes sense that we want to use `join` again, with `country_points` on the left (i.e., as the data frame on which we invoke `join`) and `tourism_spending` on the right (i.e., as the data frame passed as an argument to `join`). There's just one problem: `country_points` is a series, and we can only invoke `join` on a data frame. (We can pass a series as the argument to `join`, though—so a series can be the right side, but not the left side, of a pandas join.)

Fortunately, we can call the `to_frame` method on our series and get back a single-column data frame with the same index we had in the series:

```
country_points.to_frame()
```

With our new data frame in place, we can invoke `join`, passing `tourism_spending` as the argument:

```
country_points.to_frame().join(tourism_spending)
```

Again, it's important to remember that a join links the left data frame with the right one, connecting them along their indexes. In this case, we end up with three columns: `country`, the index column that is shared by the left and right, `points` from the left, and `Value` from the right.

Here's what the first five rows look like after performing this join:

```
country         points                  Value
Albania         88.0                    NaN
Argentina       85.9960930562955        NaN
Australia       87.89247528747227       37634.433333333334
Austria         89.27674190382729       16673.886363636364
Bos and Herz    84.75                   NaN
```

Avoiding duplicate column names

What happens if the left and right data frames have identically named columns? After all, although pandas indexes don't need to have unique elements, column names must be unique. If you try to join frames such that you'll end up with more than one column with the same name, you'll get a `ValueError` exception saying "columns overlap but no suffix specified." And indeed, pandas allows you to specify what the suffixes should be for the left side (`lsuffix`) and right side (`rsuffix`) when you invoke `join`. For example, we can join `oecd_df` with itself (already a wild idea known as a "self join," for which there are practical uses) with

```
oecd_df.join(oecd_df, lsuffix='_l', rsuffix='_r')
```

The data frame we get back has the `abbrev` index and two identical columns named `country_l` and `country_r`.

The good news is that this join worked. But as you look at it, you'll likely notice that there are `NaN` values in many rows of the `Value` column. That's because the index of the left data frame (in this case, `country_points.to_frame()`) dictates the index of the resulting data frame. As a result, this is known as a *left join*. In a left join, columns from the right frame are missing values (and thus have `NaN`) wherever there was no corresponding row for the left's index.

For example, after performing this join, although we have both `points` and `Value` for Australia and Austria, there is a `NaN` in `Value` (i.e., tourism information) for Albania, Bulgaria, and Chile (among others). That's because although we had wine-quality information for these countries (and thus an entry in the left side's index), we didn't have tourism information (in the right side's index).

There are other types of joins, too. If we want to use the right data frame's index in the result, we can use a *right join*. We can accomplish that in pandas by passing `how='right'` to the `join` method. (By default, the method assumes `how='left'`.) In such a case, we get `NaN` values on columns from the left frame wherever it has no index entry corresponding to the right.

We can also be fancy and do an *outer join*, in which case the output frame's index is the combination of the left's index and the right's index. We may thus end up with `NaN` values in columns from both the left and right, depending on which index value was missing. And so, for the final part, I asked you to perform an outer join:

```
country_points.to_frame().join(tourism_spending,
                               how='outer')
```

The resulting data frame has 54 rows rather than 48, reflecting the union of the indexes from the left and right. And we now have `NaN` values from the left, such as for Belgium and Denmark, along with `NaN` values from the right. Outer joins ensure that you don't lose any data when combining data sources, but they don't automatically interpolate values—so you will almost certainly end up with some null values, which (as we've seen in chapter 5) need cleaning in various ways.

Here are the first five rows from this outer join. Notice that Belgium now appears, with a `NaN` for `points`:

```
country                        points          Value
Albania                     88.000000            NaN
Argentina                   85.996093            NaN
Australia                   87.892475   36727.966667
Austria                     89.276742   11934.563636
Belgium                           NaN   20859.883455
```

Finally, I asked you to determine whether there's any correlation between the scores a country received from the wine magazine's judges and the amount its citizens spend on tourism. To find this, we can use the `corr` method:

```
country_points.to_frame().join(
    tourism_spending, how='outer').corr()
```

This finds how highly correlated each column is to the other columns in the data set. A score of 1 indicates that it's 100% positively correlated, meaning when one column goes up, the other column goes up by the same degree. A score of –1 indicates that it's 100% negatively correlated, meaning when one column goes up, the other goes *down* by the same degree. A score of 0 indicates that there is no correlation at all. Generally speaking, the closer to 1 (or –1) the score, the more highly correlated the two columns are. By default, `corr` uses the Pearson correlation, but you can change that by passing another value to the "method" keyword argument.

The output from `corr` is a data frame with an identical index and columns. We can thus see how highly correlated (or not) any two columns are by finding one along the index and the other along the columns. (The data is duplicated; we can do it either way.) Along the diagonal, we always see a correlation of 1, because a column is 100% positively correlated with itself.

Our result? We get 0.288, which points to a weak positive correlation between the two. So yes, countries that spend more on tourism are more likely to have highly rated wines. But the relationship is far from strong, so don't select wine based on tourism expenditures.

Solution

```
oecd_df = pd.read_csv('../data/oecd_locations.csv',
                      header=None,
                      names=['abbrev', 'country'],
                      index_col='abbrev')
```

```
oecd_tourism_df = pd.read_csv(
    '../data/oecd_tourism.csv',
    usecols=['LOCATION', 'TIME', 'Value'],
    index_col='LOCATION')

tourism_spending = (
    oecd_df
    .join(oecd_tourism_df)
    .groupby('country')['Value'].mean()
)

wine_df = pd.read_csv(
    '../data/winemag-150k-reviews.csv',
    usecols=['country', 'points'])

country_points = (
    wine_df
    .groupby('country')['points'].mean()
)

country_points.sort_values(ascending=False)
country_points.to_frame().join(tourism_spending)
country_points.to_frame().join(tourism_spending,
    how='outer')
country_points.to_frame().join(tourism_spending,
    how='outer').corr()
```

You can explore a version of this in the Pandas Tutor at http://mng.bz/A8eK.

Beyond the exercise

- Read in the three data frames, but without setting an index. Ensure that the column names in `oecd_tourism_df` are `abbrev`, `TIME`, and `Value` and that the `dtype` of the `Value` column is `np.int64`.
- Perform the same joins as before, but using `merge` rather than `join`.
- How is the default `merge` different from the default `join` when it comes to `NaN` values?

Summary

In this chapter, we dove even further into the world of split-apply-combine, looking at grouping, joining, and sorting from a variety of new perspectives. It's a rare project that doesn't use these techniques at least a little, so I hope you took the time to review these exercises and compare your answers with mine.