

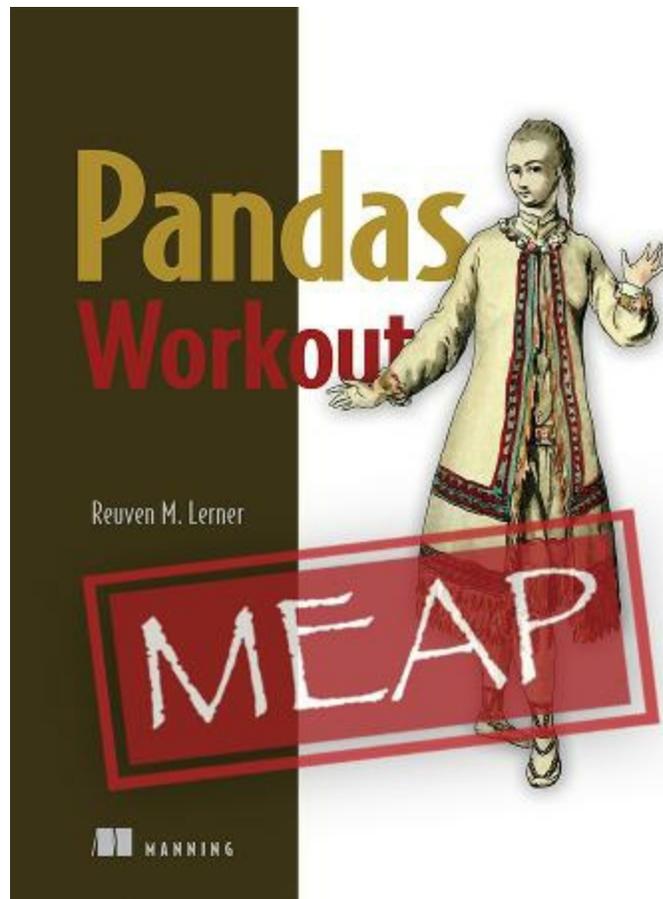
# Pandas Workout

Reuven M. Lerner



# Pandas Workout V13

1. [MEAP VERSION 13](#)
2. [Welcome](#)
3. [1 Series](#)
4. [2 Data frames](#)
5. [3 Importing and exporting data](#)
6. [4 Indexes](#)
7. [5 Cleaning data](#)
8. [6 Grouping, joining, and sorting](#)
9. [7 Midway project](#)
10. [8 Strings](#)
11. [9 Dates and times](#)
12. [10 Visualization](#)
13. [11 Performance](#)
14. [12 Final project](#)



**MEAP VERSION 13**

 **MANNING PUBLICATIONS**

# 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

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

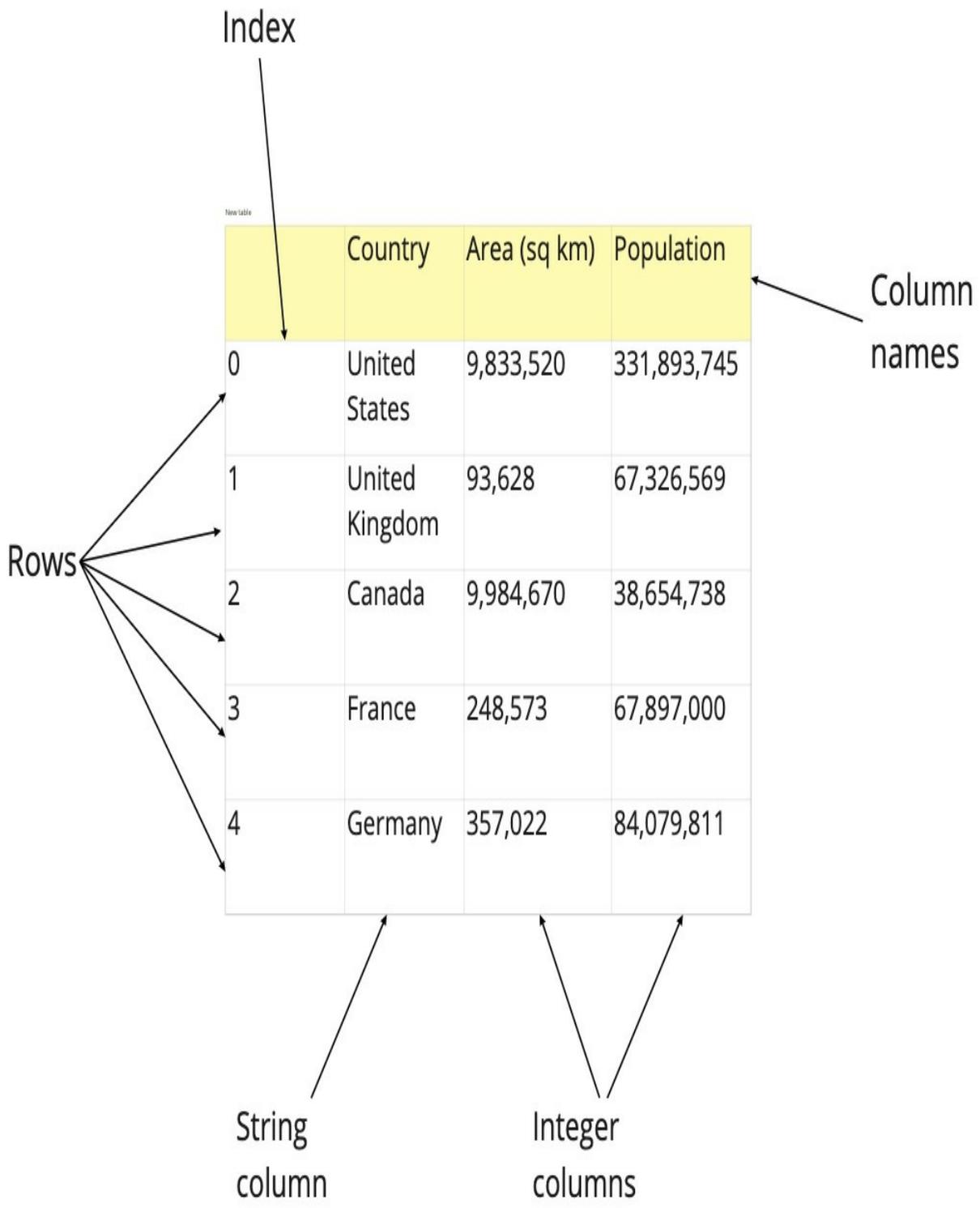
**In this book**

[MEAP VERSION 13](#) [About this MEAP](#) [Welcome](#) [Brief Table of Contents](#) [1 Series](#) [2 Data frames](#) [3 Importing and exporting data](#) [4 Indexes](#) [5 Cleaning data](#) [6 Grouping, joining, and sorting](#) [7 Midway project](#) [8 Strings](#) [9 Dates and times](#) [10 Visualization](#) [11 Performance](#) [12 Final project](#)

# 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.

**Figure 1.1. Each of a data frame's columns is a 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.

**Figure 1.2. Invoking a series method (such as mean) on a data frame often returns one value for each column**

	c1	c2	c3
r1			
r2			
r3			

`df['c1'].mean()`  
returns a float

`df.mean()`  
returns a series, each  
column's  
mean

One  
series

Another  
series

A third  
series

Deep understanding of series can be useful in other ways, too. In particular, with a "boolean index" (also known as a "mask index"), we can retrieve selected rows and columns of a data frame. (If you aren't familiar with boolean indexes, see the sidebar, "Selecting values with booleans," below.)

One of the most important and powerful tools we have as Pandas users is the index, used to retrieve values from both series and data frames. 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.



#### Note

I use a number of variable names throughout this book:

- `s` refers to a series
- `df` refers to a data frame
- `pd` is an alias to the Pandas library, loaded with `import pandas as pd`

While I'm a big fan of using semantically powerful variable names, I tend to use `s` and `df` quite a bit when teaching Pandas. Given that we're normally working with only one series or data frame at a time, I'll assume that its meaning will be clear. In the rare cases when I use more than one series or data frame, I'll normally add numbers to `s` and `df`.

## 1.1 Useful references

Table 1.1. What you need to know

Concept	What is it?	Example	To
Jupyter	Web-based system for programming	jupyter notebook	<a href="http://jupyter.org">http</a>

	in Python and data science		
f-strings	Strings into which expressions can be interpolated	<code>f'It is currently {datetime.datetime.now()}'</code>	<a href="#">http</a> and <a href="#">http</a>
data types (aka dtype)	Data types allowed in series	<code>np.int64</code>	<a href="#">http</a>
<code>pd.Series.mean</code>	returns the arithmetic mean of the series contents	<code>s.mean()</code>	<a href="#">http</a>
<code>np.random.randint</code>	returns a NumPy array of randomly selected integers	<code>np.random.randint(0, 10, 100)</code>	<a href="#">http</a>
<code>np.random.rand</code>	returns a NumPy array of randomly selected floats between 0	<code>np.random.rand(10)</code>	<a href="#">http</a>

	and 1		
<code>np.random.normal</code>	returns a NumPy array of random floats in a normal distribution	<code>np.random.normal(100, 10, 5)</code>	<a href="#">http</a>
<code>s.std()</code>	returns the standard deviation of a series	<code>s.std()</code>	<a href="#">http</a>
<code>s.loc</code>	access elements of a series by labels or a boolean array	<code>s.loc['a']</code>	<a href="#">http</a>
<code>s.iloc</code>	access elements of a series by position	<code>s.iloc[0]</code>	<a href="#">http</a>
<code>s.value_counts</code>	returns a sorted (descending frequency) series counting	<code>s.value_counts()</code>	<a href="#">http</a>

	how many times each value appears in s		
s.round	returns a new series based on s, in which the values are rounded to the specified number of decimals.	s.round(2)	<a href="http://">http</a>
s.diff	returns a new series based on s, whose values contain the differences between each value in s and a previous row.	s.diff(1)	<a href="http://">http</a>
s.describe	returns a series summarizing all major descriptive statistics in s	s.describe()	<a href="http://">http</a>

<code>pd.cut</code>	returns a series with the same index as <code>s</code> , but with categorized values based on cut points	<code>pd.cut(s, bins=[0, 10, 20], labels=['a', 'b', 'c'])</code>	<a href="#">http</a>
<code>pd.read_csv</code> with <code>squeeze</code>	returns a new series based on a single-column file	<code>s = pd.read_csv('filename.csv').squeeze()</code>	<a href="#">http</a>
<code>str.split</code>	Breaks strings apart, returning a list	<code>'abc def ghi'.split() # returns ['abc', 'def', 'ghi']</code>	<a href="#">http</a>

## 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:

- What is the student's average test score for the entire year?
- What is the student's average test score during the first half of the year (i.e., the first five months)?
- What is the student's average test score during the second half of the year?

- Did the student improve their performance in the second half? If so, then by how much?

## 1.2.1 Discussion

In this first exercise, I asked you to create a series of 10 elements, with random integers from 70-100.

This raises several questions:

- How do we define a series?
- How can we create 10 random integers from 70-100?
- How can we set the index of the series to month names?

To define a Pandas series, we call `Series`, passing it an iterable—typically, a Python list or NumPy array. For example:

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

Here, I asked you to define the series such that it contains 10 random integers. There are certain areas in which Pandas defers to NumPy, including when generating random numbers. We can get a NumPy array of random integers by calling `np.random.randint`; its three arguments indicate the range (minimum and maximum) of the random numbers, as well as how many we want.



### Note

The Python standard library's `random` module has a `randint` method, which returns a random integer:

```
random.randint(0, 100)
```

In the case of `random.randint`, the returned values range from 0 to 100, **including** 100.

It's easy to confuse this behavior with that of NumPy's `np.random.randint` method:

```
np.random.randint(0, 100, 10)
```

The NumPy method differs from `random.randint` in two ways: First, it takes a third argument, indicating the length of the returned series. Second, the second argument is one more than the highest value we can get back. That is, while `random.randint(0, 100)` could potentially return a value of 100, `np.random.randint(0, 100, 10)` cannot.

I can thus get 10 random integers between 70 and 100 with:

```
np.random.randint(70, 101, 10) #1
```

I can use them to create a series:

```
s = Series(np.random.randint(70, 101, 10))
```



#### Note

Since this is a book of exercises, you will likely want to compare your solutions with mine. How can we do that, though, if we're both generating random numbers? We can set the "random seed," the number which kicks off NumPy's random number generator, to an agreed-upon number, such as 0.

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, just like me. All of my exercises that use random numbers will thus start with `np.random.seed(0)`, and I suggest you do that, too.

That said, the NumPy documentation explicitly says that `np.random.seed` is a convenience, legacy function, and that it doesn't represent the best practice for setting the random seed in a production program. You can see a demonstration of their suggested replacement here: <http://mng.bz/QPxxw>.

We now have a series of random integers between 70 and 100. But the index contains integers from 0 through 9—much as would be the case in a NumPy array, or a Python list. There's nothing inherently wrong with a numeric index, but Pandas gives us much more power and flexibility, letting us use a wide variety of data types, including strings.

We can change the index by assigning to the `index` attribute:

```
np.random.randint(0)
s = Series(np.random.randint(70, 101, 10))
s.index = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()
```

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.

If we know what index we'll want when we create the series, we can assign it to the `index` keyword parameter:

```
np.random.randint(0)
months = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()
s = Series(np.random.randint(70, 101, 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 a new value to `s.index`.

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

We first want to find the student's average test score for the entire year. We can calculate that with the `mean` method, which runs on any numeric series. (Even if the series only contains integers, `mean` will always return 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," although I like to call them "fancy strings") allow 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 five elements in the series, and then the second five elements. There are 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:

```
s[:5].mean() #1
```

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:

```
s[5:].mean() #1
```

**Figure 1.3. Retrieving slices from our slice**

New table

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

`s[:5]`      `s[5:]`

miro

I would argue that it's even better to use the `.loc` and `.iloc` accessors. Whereas `.loc` retrieves one or more elements based on the index, `.iloc` retrieves based on the numeric position—the default index. Let's start with `.iloc`, because its usage is quite similar to what we've already written:

```
s.iloc[:6].mean()
```

"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. It's not mandatory with a slice, but is definitely a good idea, to keep your code more readable.

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

The endpoint of a slice is normally "up to and not including," but 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 (with `.loc`) as the number-based slice (with `.iloc`). If your Pandas analysis takes a long time, consider trying positional indexes with `.iloc`.

**Figure 1.4.** Retrieve via the index using `.loc`, and the position using `.iloc`.

New table

.loc → Index	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun
.iloc → Default (numeric) index	0	1	2	3	4	5	6	7	8	9
Values	82	85	91	70	73	97	73	77	79	89

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 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()
```

Finally, we can check the improvement by subtracting the first half's average from the second half. I decided to assign each half's mean to a variable, and then calculated the difference in an f-string:

```
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 Solution

```
np.random.seed(0)

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

s = Series(np.random.randint(70, 101, 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}')
```

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
20pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame
0Anp.random.seed%280%29%0A%0Amonths%20%3D%20'Sep%20Oct%20Nov%20De
20Feb%20Mar%20Apr%20May%20Jun'.split%28%29%0A%0As%20%3D%20Series%
random.randint%2870,%20101,%2010%29,%0A%20%20%20%20%20%20%20%20%2
index%3Dmonths%29%0A%0Aprint%28f'Yearly%20average%3A%20%7Bs.mean%
7D'%29%0A%0Afirst_half_average%20%3D%20s%5B'Sep'%3A'Jan'%5D.mean%
second_half_average%20%3D%20s%5B'Feb'%3A'Jun'%5D.mean%28%29%0A&d=
&lang=py&v=v1
```

## 1.2.3 Beyond the exercise

Here are three additional exercises to help you better understand using `.loc` and `.iloc` to retrieve data from `s`, the series used in this exercise:

- 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. (If you aren't yet familiar with boolean indexing, see the "Selecting values with booleans" section, below.)
- What were this student's five highest scores?
- 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. Be sure to read the documentation for the round method (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.round.html>), to understand its arguments, as well as how it handles numbers like 15 and 75.

### Understanding 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.

**Figure 1.5. To calculate the mean, we first sort the values, then take the middle one**

New table

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

New table

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



Median  
is 80.5

Mean  
is 81.6

**Figure 1.6. By changing one value, we can see how the mean is more easily affected by outliers than the median**

New table

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

New table

Dec	Jan	Mar	Apr	May	Sep	Oct	Jun	Nov	Feb
3	4	6	7	8	0	1	9	2	5
70	73	73	77	79	82	85	89	91	1000

Median  
is 80.5

Mean  
is 171.9

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. So the higher the standard deviation, the more the values in the data set vary from the mean.

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, and the Pandas authors decided to default to this calculation. (NumPy's `std` calculation doesn't do this.)

If you really want to get the same result as we calculated, and as NumPy provides, 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 deviations, and 99.7 will be within three standard deviations.

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`. This method 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.

In this section, we used a number of so-called "aggregation methods," which run on a series and return a single number—for example, `sum`, `mean`, `median`, and `std`. We'll use these quite a bit throughout the book, and in any data analysis projects you work on.



#### 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() #1
```

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

```
s = Series('1234 5678 9012'.split())
s.mean() #1
```

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.

### Understanding dtype

In Python, we make constant use of our built-in core data types: `int`, `float`, `str`, `list`, `tuple`, and `dict`. Pandas is a bit different, in that we don't use those types much. Rather, we use the types that we get from NumPy, which provide us with a thin, Python-compatible layer over values 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. When the `dtype` is `object`, we can usually assume that the series contains Python strings; more on that in chapter 8. Storing non-string objects is rare, and should generally be avoided, but there are sometimes good reasons for it. You'll also have a `dtype` of `object` if there are multiple, different types in the series.

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`, and `np.float64`. (On some computers, you also have `np.float128`.)
- Python objects: `object`

When you create a series, Pandas normally assigns the `dtype` based on the argument you pass to `Series`:

- 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 #1
```

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.

Given this issue, why not just go for broke, and use 64-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 64 bits is 8 bytes, which doesn't sound like very much for a modern computer. But if you're dealing with 1 billion numbers, using 64 bits means that the data will consume 8 gigabytes of memory—without taking into consideration 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; each case must be evaluated on its own merits.

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 #1
```

```
s = s.astype(np.int64)
s.dtype #2
```

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 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 "scale" the test scores, known in some places as "grading on a curve." That is: They would assume that the average test score should be 80, calculate the difference between our actual mean and 80, then add that difference 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 80 to each of the scores.

### 1.3.1 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
```

**Figure 1.7. When we add two series together, the result is a new series—the result of adding elements at the same index.**

s1

New table

0	1	2	3
10	20	30	40

+

New table

s2

0	1	2	3
100	200	300	400

=

New table

0	1	2	3
110	220	330	440

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.

**Figure 1.8. Vectorized operations work use the index, not the position.**

s1

New table

'a'	'b'	'c'	'd'
10	20	30	40

+

New table

s2

'd'	'c'	'b'	'a'
100	200	300	400

=

New table

'a'	'b'	'c'	'd'
410	320	230	140

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.)

**Figure 1.9. Operations involving a series and a scalar value result in "broadcasting" the operation, resulting in a new series.**

s1

New table

'a'	'b'	'c'	'd'
10	20	30	40

+

s2

New table  
10

=

New table

'a'	'b'	'c'	'd'
410	320	230	140

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. 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.

### 1.3.2 Solution

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

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
random.seed%280%29%0A%0Amonths%20%3D%20'Sep%20Oct%20Nov%20Dec%20J
Feb%20Mar%20Apr%20May%20Jun'.split%28%29%0A%0As%20%3D%20Series%28
random.randint%2870,%20101,%2010%29,%0A%20%20%20%20%20%20%20%20%2
index%3Dmonths%29%0A%0Aprint%28f'Yearly%20average%3A%20%7Bs.mean%
7D'%29%0A%0Afirst_half_average%20%3D%20s%5B'Sep'%3A'Jan'%5D.mean%
second_half_average%20%3D%20s%5B'Feb'%3A'Jun'%5D.mean%28%29%0A%0A
20%2880%20-%20s.mean%28%29%29&d=2022-11-08&lang=py&v=v1
```

### 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. It dictates the way in which vectorized operations will be performed, as well as the index of the new series created by the operation. 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, 25, 32, we want the series 1, 2, 3.

### 1.4.1 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). We know that each number will contain either 1 or 2 digits.

To get the 10s digit, we can:

- 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`, 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:

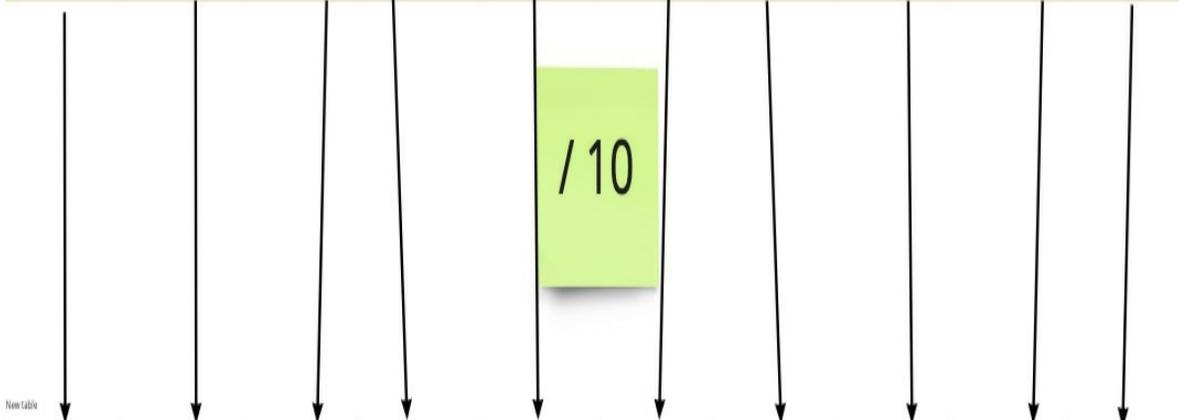
```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: int8
```

Notice that the dtype here is `int8`.

**Figure 1.10.** Graphical depiction of dividing the series by 10, then converting to `np.int8`

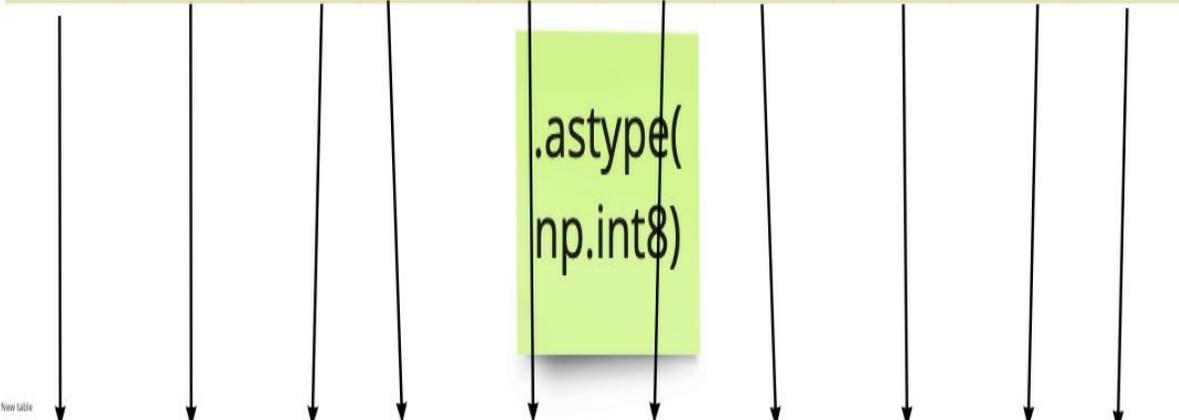
New table

0	1	2	3	4	5	6	7	8	9
44	47	64	67	67	9	83	21	36	87



New table

0	1	2	3	4	5	6	7	8	9
4.4	4.7	6.4	6.7	6.7	0.9	8.3	2.1	3.6	8.7



New table

0	1	2	3	4	5	6	7	8	9
4	4	6	6	6	0	8	2	3	8

There is another way to do this, which involves some more type conversions. This time, won't convert our series into floats, but rather to **strings**. Why? Because when we turn our integers into strings, we can 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 on that accessor 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(-2)`.

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 result, as you can see from the dtype, is object, which typically means Python strings. Can we turn it back into a series of integers? Yes, calling `astype` with an integer argument. 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
```

**Figure 1.11. Graphical depiction of turning the series into strings, retrieving the item at index -2, and replacing NaN with 0**

New table

0	1	2	3	4	5	6	7	8	9
44	47	64	67	67	9	83	21	36	87

`.astype(str)`

New table

0	1	2	3	4	5	6	7	8	9
'44'	'47'	'64'	'67'	'67'	'09'	'83'	'21'	'36'	'87'

`.str.get(-2)`

New table

0	1	2	3	4	5	6	7	8	9
'4'	'4'	'6'	'6'	'6'	NaN	'8'	'2'	'3'	'8'

`.fillna(0)`

New table

0	1	2	3	4	5	6	7	8	9
'4'	'4'	'6'	'6'	'6'	0	'8'	'2'	'3'	'8'

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`, which 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.

### 1.4.2 Solution

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

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
random.seed%280%29%0As%20%3D%20Series%28np.random.randint%280,%20
%29%29%0A%28s%20/%2010%29.astype%28np.int8%29%0A&d=2022-11-08&lan
```

### 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.

## 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.loc[3] #1
```

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.loc[[True, True, False, False, True]] #1
```

**Figure 1.12. Choosing items via a mask index**

New table

0	1	2	3	4
10	20	30	40	50

New table

0	1	2	3	4
True	True	False	False	True

New table

0	1	4
10	20	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.loc[s < 30] #1
```

**Figure 1.13. Generating a boolean series by broadcasting an operation**

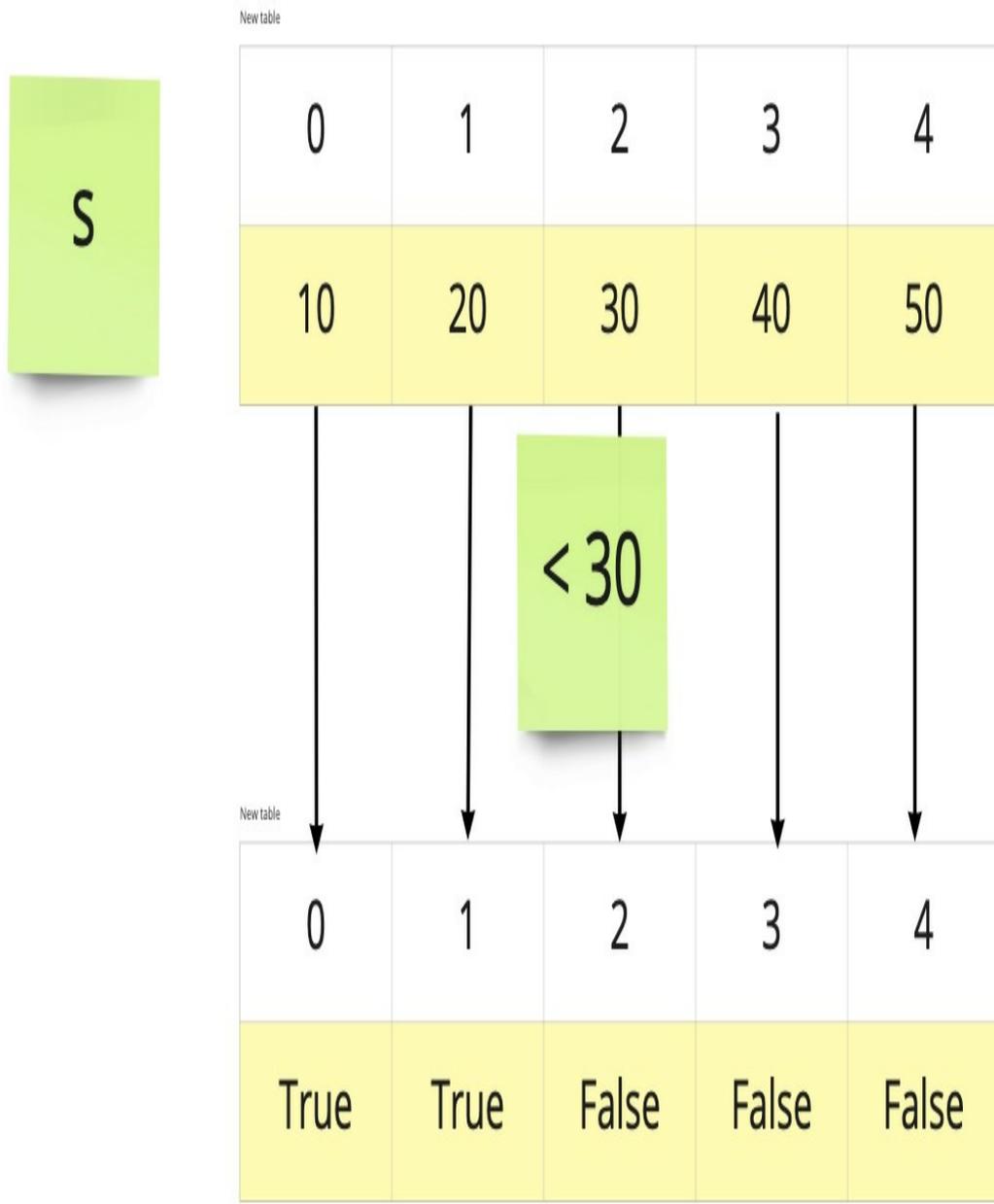
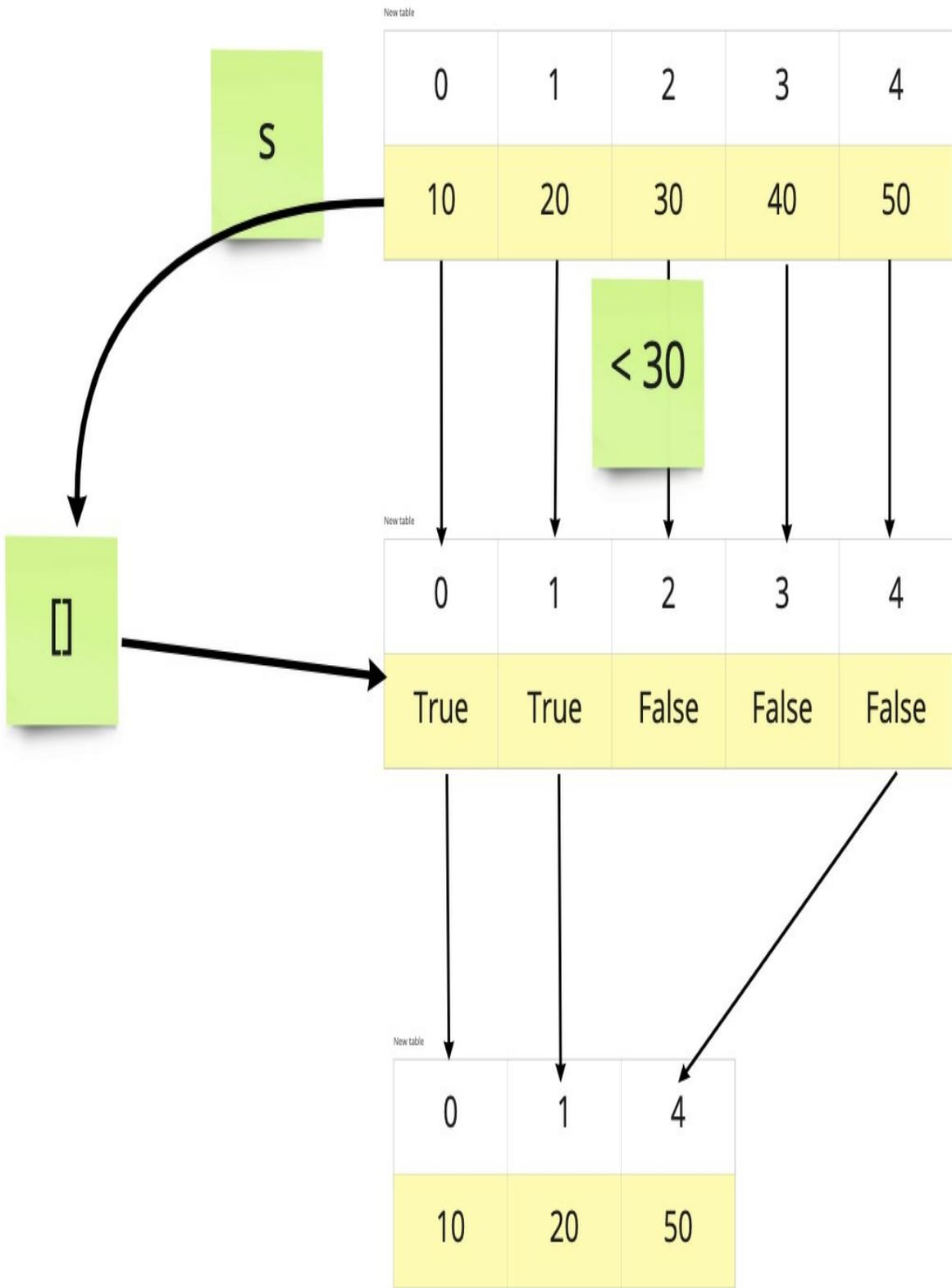


Figure 1.14. Using the boolean series as a mask index



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.loc[s <= s.mean()] #1
```

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.loc[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.

One final note: Given a series `s`, you can retrieve multiple items, from different indexes, using "fancy indexing"—passing a list, series, or similar iterable inside of the square brackets. For example:

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

The above code returns a series containing two values—the elements at `s.loc[2]` and `s.loc[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 original indexes and values.

Don't confuse fancy indexing with the application of a mask index; in the former case, the inner square brackets contain a list of values from the index. In the case of a mask index, the inner square brackets contain boolean (`True` and `False`) values.

## 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 "descriptive statistics," as they're called in statistics and data analytics, typically include the mean, standard deviation, minimum value, 25% quantile, median, 50% quantile, and maximum value. Understanding and using descriptive statistics is a key skill for anyone working with data, and in this exercise, you'll practice doing so, with the following:

- 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? (You don't need to calculate the difference, but rather consider why they aren't the same.)
- Replace the minimum value with 5 times the maximum value.
- Get the descriptive statistics again. Did the mean, median, and standard

deviations change from their previous values? (Again, it's enough to see the difference without calculating it.) If so, why?

## 1.5.1 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        0.157670
std         99.734467
min        -485.211765
25%        -66.864170
50%         0.172022
75%         67.343870
max         424.177191
dtype: float64
```

The mean, as we can see, is 0.157670. 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 0.172022, 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.loc[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.loc[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?

```
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
```

First, the mean value has gone up by a bit—which makes sense, given that we took the smallest value and made it larger than the previously defined largest value. 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 median, 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.

## 1.5.2 Solution

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

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A  
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0  
random.seed%280%29%0A%0As%20%3D%20Series%28np.random.normal%280,%  
%20100_000%29%29%0A%0Aprint%28s.describe%28%29%29%0A%0As.loc%5Bs%  
%20s.min%28%29%5D%20%3D%205*s.max%28%29%0As.describe%28%29%0A&d=2  
&lang=py&v=v1
```

### 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

Newcomers to Pandas often assume that a series index must be unique. After all, the index in a Python string, list, or tuple is unique, as are the keys in a Python dictionary. But the values in a Pandas index can repeat, making it easier to retrieve values with the same index. If an index contains user IDs, country codes, or e-mail addresses, you can then use it to retrieve data associated with specific index values that would otherwise require a messier and longer mask index.

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 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

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.

## 1.6.2 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()
```

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
%20%3D%20'Sun%20Mon%20Tue%20Wed%20Thu%20Fri%20Sat'.split%28%29%0A
random.seed%280%29%0As%20%3D%20Series%28np.random.normal%2820,%20
29,%0A%20%20%20%20%20%20%20%20%20index%3Ddays*4%29.round%28%29
28np.int8%29%0A%0As.loc%5B'Mon'%5D.mean%28%29%0A&d=2022-11-08&lan
```

## 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?

## 1.7 Exercise 6: Passenger frequency

In this exercise, we're going to start to look at some real-world data, loaded from a one-column CSV file. We'll take a deeper look at reading from and writing to files in Chapter 3, but we'll start here by invoking `pd.read_csv` and then calling `squeeze` on the one-column data frame it returns.



#### Note

Pandas previously supported a `squeeze` parameter to `read_csv`. Passing `squeeze=True` would accomplish the same thing as the two-step call to `read_csv` followed by a call to `squeeze`. The `squeeze` parameter in `read_csv` has been deprecated, which we reflect here in the code.

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 passenger, vs. the (theoretical) maximum of 6 passengers.

### 1.7.1 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` returns a data frame—but if we read a file containing only one column, we'll get a data frame with a single column. We can then invoke `squeeze` on that single-column data frame, getting a back a series. 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:

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

The resulting series will have 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.loc[s==1].count() #1
s.loc[s==6].count() #2
```

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.loc[s==1].count() / s.count() #1
s.loc[s==6].count() / s.count() #2
```

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
```

## 1.7.2 Solution

```
import pandas as pd
from pandas import Series, DataFrame

s = pd.read_csv('data/taxi-passenger-count.csv', header=None).squ
s.value_counts(normalize=True)[[1,6]]
```

You can explore a version of this in the Pandas Tutor at:

<https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0A>

```
3D%20Series%28%5B1,%201,%206,%201,%205,%201,%201,%204,%201,%201,%201,%202,%201,%202,%201,%201,%201,%201%5D%29%0A%0As.value_counts  
normalize%3DTrue%29%5B%5B1,6%5D%5D%0A&d=2022-11-08&lang=py&v=v1
```

### 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,  $\leq 2$  miles
- medium,  $> 2$  miles, but  $\leq 10$  miles
- long,  $> 10$  miles

### 1.8.1 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) #1
categories.loc[:] = 'medium' #2
categories.loc[s<=2] = 'short' #3
categories.loc[s>10] = 'long' #4
categories.value_counts()
```

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  $< = 10$ .

This means that the first boundary needs to be less than the smallest value in `s`. I often accomplish this by setting it to be `s.min()-1`.

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()-1, 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 #1
Categories (3, object): ['short' < 'medium' < 'long'] #2
```

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()-1, 2, 10, s.max()],
        labels=['short', 'medium', 'long']).value_counts()
```

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

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

## 1.8.2 Solution

```
import pandas as pd
from pandas import Series, DataFrame

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

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

You can explore a version of this in the Pandas Tutor at:

<https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0A>

```
3D%20Series%28%5B1.63,%200.46,%200.87,%202.13,%201.4,%201.4,%201.
%201.27,%0A%200.6,%200.01,%201.65,%201.14,%200.5,%201.28,%200.6,%
%200.7,%201.9,%201.1%5D%29%0A%0Apd.cut%28s,%20bins%3D%5Bs.min%28%
%2010,%20s.max%28%29%5D,%0A%20%20%20%20%20%20%20labels%3D%5B'shor
%20'medium',%20'long'%5D%29.value_counts%28%29%0A&d=2022-11-08&la
```

### 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 structures that most people think of when they work with Pandas.

## 2 Data frames

Starting long before the invention of computers, people have used tables to present data. That's because tables make it easy to enter, display, understand, and analyze data. Each row in a table represents a single record or data point, and every column describes an attribute associated with each point. For example, consider this table of country names, sizes (in square km) and population, with data taken from Wikipedia toward the end of 2022:

**Table 2.1. Country data**

Country	Area (sq km)	Population
United States	9,833,520	331,893,745
United Kingdom	93,628	67,326,569
Canada	9,984,670	38,654,738
France	248,573	67,897,000
Germany	357,022	84,079,811

If it seems obvious to arrange data in this way, that's just because we've seen them for so long, and in so many contexts. Indeed, here are some examples of tables I've seen in just the last few days:

- Stock-market updates—the rows are some stocks and popular indexes, and the columns are the current value, the absolute change since

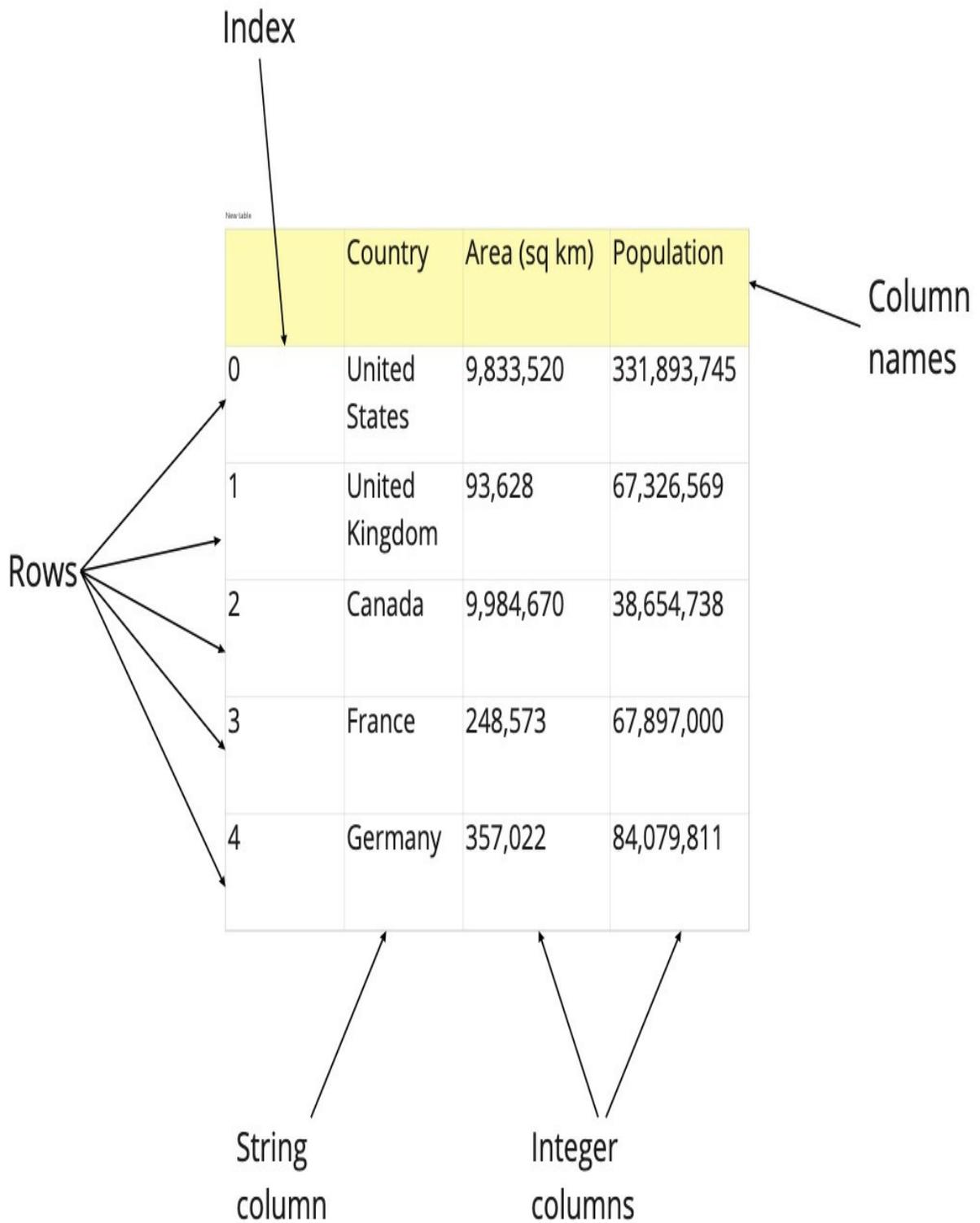
- yesterday, and the percentage change since yesterday
- Luggage allowances on international flights—the rows describe different types of tickets, and the columns indicate how large or heavy your carry-on and checked bags can be
  - Nutrition information on packaged food—the rows are different items we want to know about (e.g., calories, fat, and sugar), and the columns describe the quantity per 100 grams or for an entire package

Because each column contains one attribute, or category, it will typically contain one type of data. Each row, however, might well contain several different types of data, because it cuts across several columns. Adding a new column means that we're adding a new dimension, or aspect, to each record. Adding a new row means that we're adding a new record, with a value for each column.

Computers have been used to store tabular information for decades, most famously in spreadsheet software such as Excel. Pandas continues this tradition, organizing tables in "data frames." Each column in a data frame is a Pandas series object. The data frame has a single index, shared by all of its columns. In many ways, a data frame is a collection of series with a common index.

Because each column in a data frame is its own series, each can have a distinct dtype. For example, you could have a data frame with one integer column, one float column, and one string column.

**Figure 2.1. The table defined above, as a Pandas data frame**



A data frame typically contains more information than we need. Before we can answer any questions, we'll first need to pare our data down to a subset of its original rows and columns. In this chapter, we'll practice doing just that—retrieving just the rows and columns that we want, based on criteria appropriate for our query. We'll see how the `.loc` accessor, boolean indexes, and various Pandas methods allow us to work on just the data that we want and need. (In Chapter 3, we'll look at how to import data from external sources. And in Chapter 5, we'll look at how to clean real-world data so that we can use it reliably.)

We'll also practice creating, modifying, and updating data frames. Sometimes we'll do that because we have new information, and want the data frame to reflect that change. And sometimes we'll do it because we need to clean our data, removing or modifying bad values.

After this chapter, you'll be comfortable doing the most common tasks associated with data frames. We'll build on these basics in later chapters, so that you can organize your data in more sophisticated and interesting ways.

## 2.1 Useful references

Table 2.2. What you need to know

Concept	What is it?	Example	To learn mo
DataFrame	returns a new data frame, based on 2-dimensional data	<code>DataFrame([[10, 20], [30, 40], [50, 60]])</code>	<a href="http://mng.b">http://mng.b</a>
	access elements of a series by		

s.loc	labels or a boolean array	s.loc['a']	<a href="http://mng.b">http://mng.b</a>
df.loc	access one or more rows of a data frame via the index	df.loc[5]	<a href="http://mng.b">http://mng.b</a>
s.iloc	access elements of a series by position	s.iloc[0]	<a href="http://mng.b">http://mng.b</a>
df.iloc	access one or more rows of a data frame by position	df.iloc[5]	<a href="http://mng.b">http://mng.b</a>
[]	access one or more columns in a data frame	df['a']	<a href="http://mng.b">http://mng.b</a>
s.quantile	Get the value at a particular percentage	s.quantile(0.25)	<a href="http://mng.b">http://mng.b</a>

	of the values		
<code>pd.concat</code>	join together two data frames	<code>df = pd.concat([df, new_products])</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.query</code>	Write an SQL-like query	<code>df.query('v &gt; 300')</code>	<a href="http://mng.b">http://mng.b</a>
<code>pd.read_csv</code>	returns a new series based on a single-column file	<code>s = pd.read_csv('filename.csv').squeeze()</code>	<a href="http://mng.b">http://mng.b</a>
<code>interpolate</code>	returns a new data frame with NaN values interpolated	<code>df = df.interpolate()</code>	<a href="http://mng.b">http://mng.b</a>

### 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 old square brackets, which is essentially equivalent to `loc`.

When we work with data frames, we must use `loc` or `iloc` to retrieve rows via the index. That's because square brackets refer to the columns.

For example, let's create a data frame:

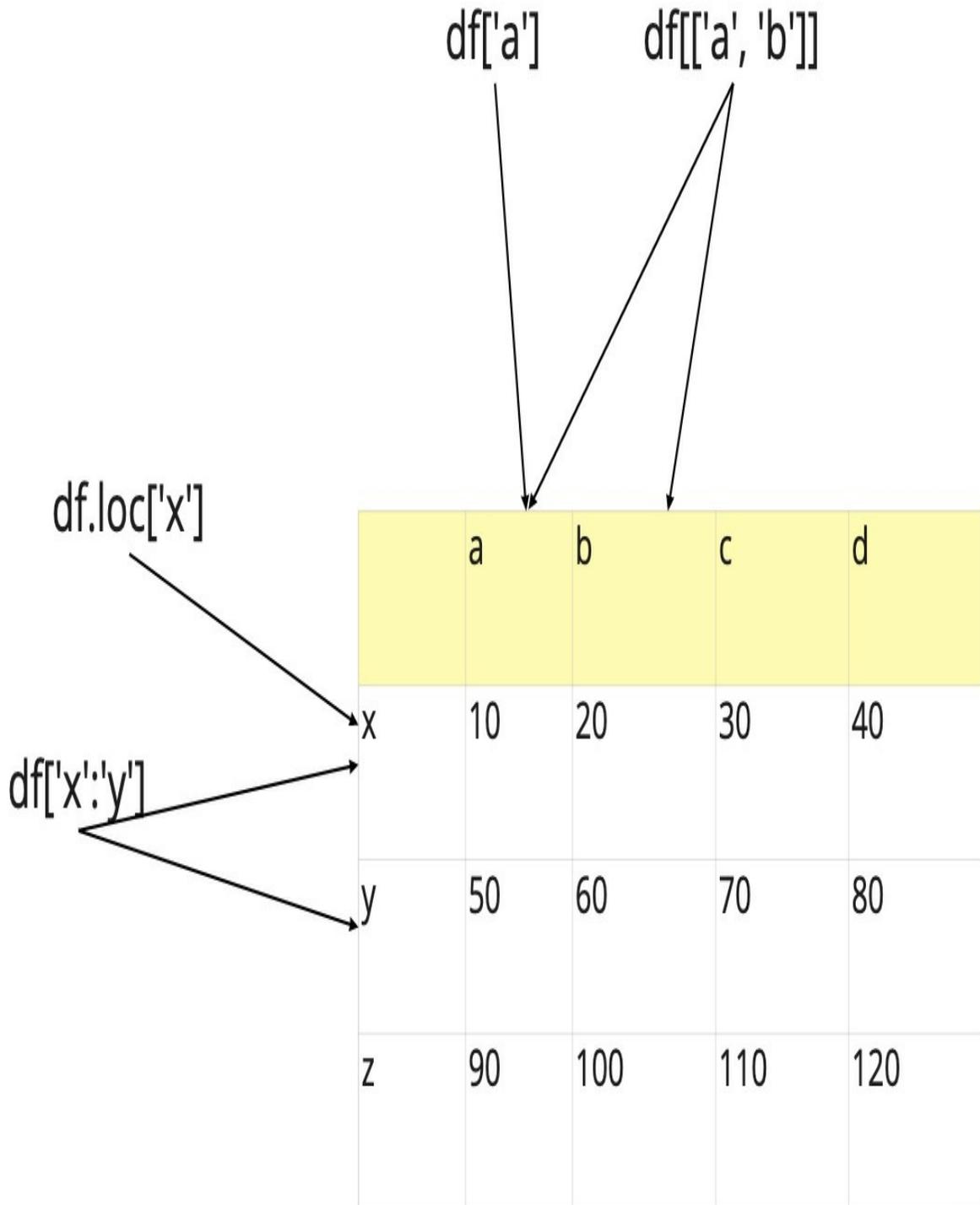
```
df = DataFrame([[10, 20, 30, 40],
                [50, 60, 70, 80],
                [90, 100, 110, 120]],
               index=list('xyz'),
               columns=list('abcd'))
```

Given this data frame, and the fact that square brackets refer to columns, we can understand how `df['a']` returns the a column, and `df[['a', 'b']]`, passing a list of columns inside of the square brackets (i.e., double square brackets) will return a new, two-column data frame based on `df`. If we ask for `df['x']`, Pandas will look for a column `x`, not see one, and raise a `KeyError` exception.

If we want to retrieve the row at index `x`, we must say `df.loc['x']`. Or if we prefer to retrieve it positionally, `df.iloc[0]`.

There is, however, an exception to the "square brackets mean columns" rule: If we use a slice, then Pandas will look at the data frame's rows, rather than its columns. So we can retrieve the rows from `x` through `y` with `df['x':'y']`. The slice tells Pandas to use the rows, rather than the columns. Moreover, the slice will return rows up to **and including** the endpoint, which is unusual for Python (but typical in Pandas).

**Figure 2.2. Our data frame**



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-identifier 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 Pandas users, it's rare to create a new data frame from scratch. You'll create it after importing 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—for example, when assembling data from non-standard sources, or just experimenting with new Pandas techniques—and knowing how to do it can be quite useful.

For this exercise, I want you to create a data frame that represents a company's inventory of five products. Each product will have a unique ID number (a two-digit integer will do), name, wholesale price, retail price, and the number of sales 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!

Once you have created this data frame, calculate the total net revenue from all of your products.

## 2.2.1 Discussion

The first part of this task involved creating a new data frame, by passing values to the `DataFrame` class. There are basically four ways to do this:

- Pass a list of lists. Each inner list represents one row. The inner lists must all be the same length, and will fill the columns positionally.
- Pass a list of dicts. Each dict represents one row, and the keys indicate which columns should be filled.
- Pass a dict of lists. Each key represents one column, and the values (lists) are each column's values.
- Pass a 2-dimensional NumPy array.

**Figure 2.3. Creating a data frame from a list of lists. Each inner list represents one row. Column names are taken positionally.**

Creating a data frame  
from a list of lists

```
df = DataFrame([  
    [10, 20, 30, 40],  
    [50, 60, 70, 80],  
    [90, 100, 110, 120]],  
    index = list('xyz'),  
    columns = list('abcd'))
```

	a	b	c	d
x	10	20	30	40
y	50	60	70	80
z	90	100	110	120

miro

**Figure 2.4. Creating a data frame from a list of dicts. Each dict is a row, and the keys indicate columns values.**

Creating a data frame  
from a list of dicts

```
df = DataFrame([  
    {'a': 10, 'b': 20, 'c': 30, 'd': 40},  
    {'a': 50, 'b': 60, 'c': 70, 'd': 80},  
    {'a': 90, 'b': 100, 'c': 110, 'd': 120}],  
    index = list('xyz'))
```

	a	b	c	d
x	10	20	30	40
y	50	60	70	80
z	90	100	110	120

miro

**Figure 2.5. Creating a data frame from a dict of lists. Each dict key is a column name, and the list contains values for that column.**

Creating a data frame  
from a dict of lists

```
df = DataFrame([  
    'a': [10, 50, 90],  
    'b': [20, 60, 100],  
    'c': [30, 70, 110],  
    'd': [40, 80, 120]),  
    index = list('xyz'))
```

	a	b	c	d
x	10	20	30	40
y	50	60	70	80
z	90	100	110	120

**Figure 2.6. Creating a data frame from a two-dimensional NumPy array**

Creating a data frame  
from a 2D NumPy array

```
df = DataFrame(  
    np.random.randint(0, 10, [3, 4]),  
    columns = list('abcd'),  
    index = list('xyz'))
```

	a	b	c	d
x	10	20	30	40
y	50	60	70	80
z	90	100	110	120

miro

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.

One advantage of a list of dicts is that you don't need to pass column names; Pandas can infer their names from the dict keys. 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 my products' 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'] #1
```

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

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

**Figure 2.7. Graphical depiction of the solution for Exercise 8**

	product_id	name	wholesale_price	retail_price	sales
0	23	computer	500	1000	100
1	96	Python Workout	35	75	1000
2	97	Pandas Workout	35	75	500
3	15	banana	0.5	1	200
4	87	sandwich	3.0	5	300



110700.0<sub>miro</sub>



- 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. It's pretty common to add new columns to an existing data frame, either to add new information you've acquired, or to store the results of per-row calculations—which is what we'll do in this exercise. It's sometimes a good idea to add a new column to hold intermediate values, as a convenience.

The backstory for this exercise is as follows: Our local government is thinking about imposing a sales tax, and is considering 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 our net income under each of the proposed rates, as well as our current net income.

### 2.3.1 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 Exercise 8, 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. Column names are unique—so just as with a dictionary, assigning to an existing column will replace it with the new one.

In the previous exercise, we calculated the total sales 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'])
```



#### Note

There is another way to add a column to a Pandas data frame, namely the `assign` method. I generally prefer to add a new column with assignment, as you've seen throughout this book. `assign` returns a new data frame, rather than modifying an existing one, which can come in handy. For example,

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

we could instead use:

```
df.assign(current_net = (df['retail_price'] - df['wholesale_price
```

Notice that any keyword arguments we pass to `df.assign` result in a new column (with the same name as the keyword argument), whose values are the keyword argument's values. Some people prefer this style, saying that they find it more readable and reproducible than assignment. I suggest that you try solving some of the exercises in this book using `assign`; you might turn out to prefer it.

What happens if we will be taxed at 15 percent? This reduces our net 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:





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, created on the fly from the values in that row. By contrast, if we retrieve a column, nothing new needs to be created, because each column is stored as a series in memory.

What if want to retrieve only part of a row? More significantly, how could we set values on only part of a row?

We can do this in several different ways, but my preference is `loc`, with two arguments in the square brackets. The first argument describes the row(s) that we want to retrieve ("row selector"), while the second describes the column(s) we want to retrieve ("column selector"). You can do something similar with `iloc`, specifying the numeric position, but I

Let's assume that we have a 5x5 data frame, with index a-e, columns v-z, and values from 10 through 250.

**Figure 2.8. Our sample data frame**

New table

	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

miro

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']
```

Especially as the arguments get longer and more complex, It can be easier to put them on separate lines:

```
df.loc['a', #1  
       'x'] #2
```

**Figure 2.9. Graphical depiction of df.loc['a', 'x']**

Column selector

Result

New table

	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

Row selector

miro

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'], #1  
       'x'] #2
```

**Figure 2.10.** Graphical depiction of `df.loc[['a', 'c'], 'x']`

Column selector

Result

New table

	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

Row selector

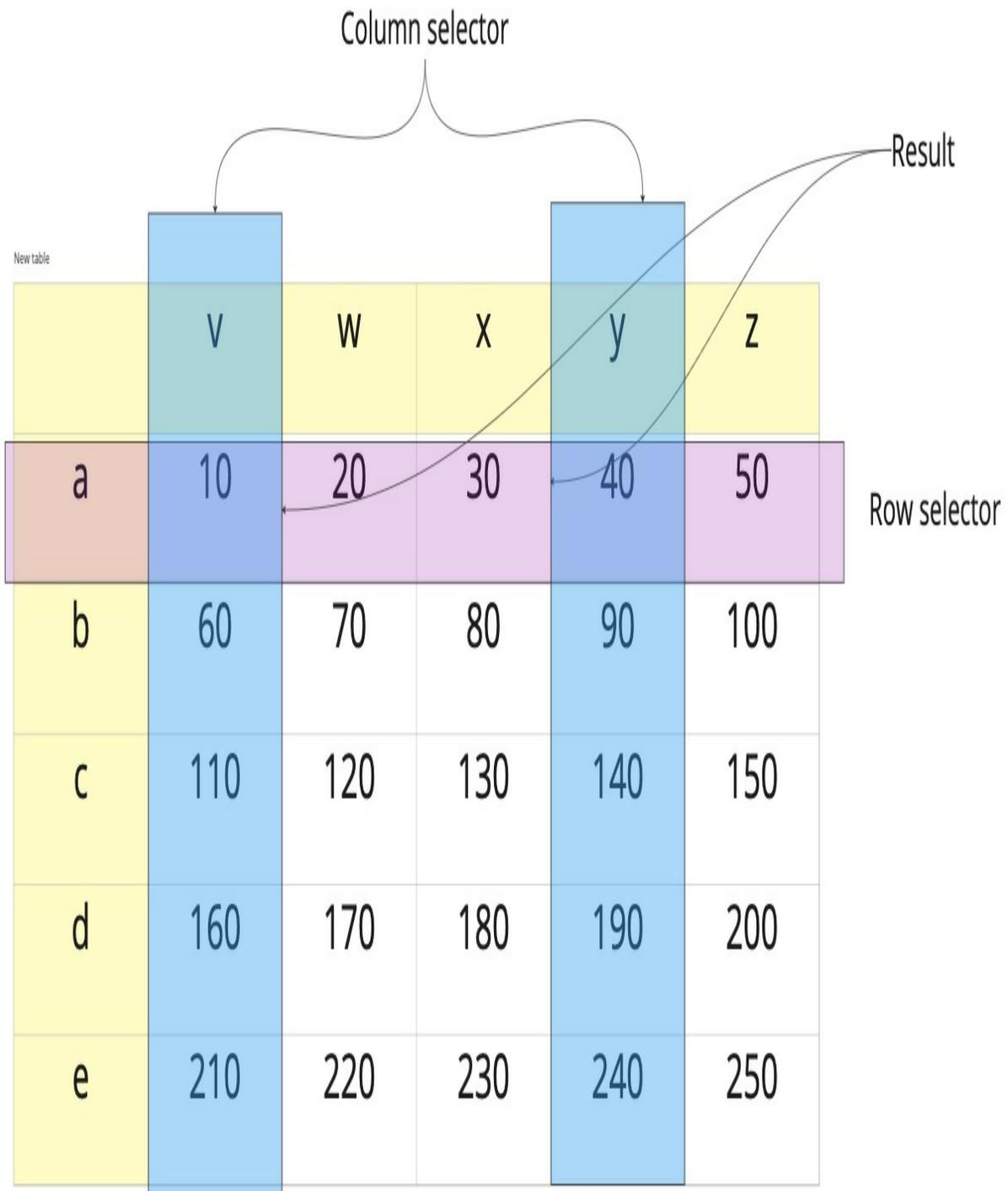
miro

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',#1  
      ['v','y']]#2
```

**Figure 2.11. Graphical depiction of `df.loc['a', ['v', 'y']]`**

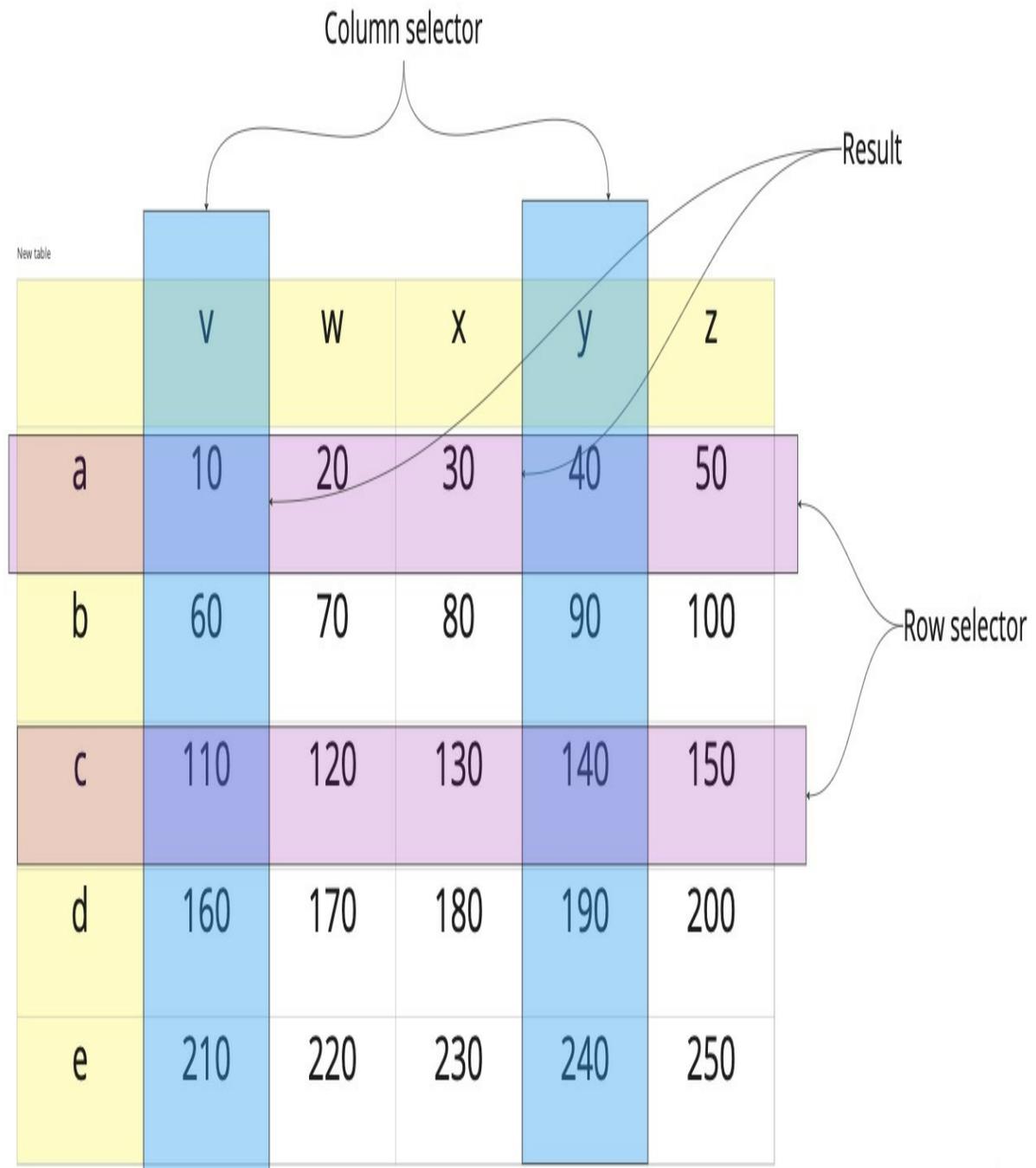


miro

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

```
df.loc[['a', 'c'], #1  
       ['v', 'y']] #2
```

**Figure 2.12. Graphical depiction of df.loc[['a', 'c'], ['v', 'y']]**



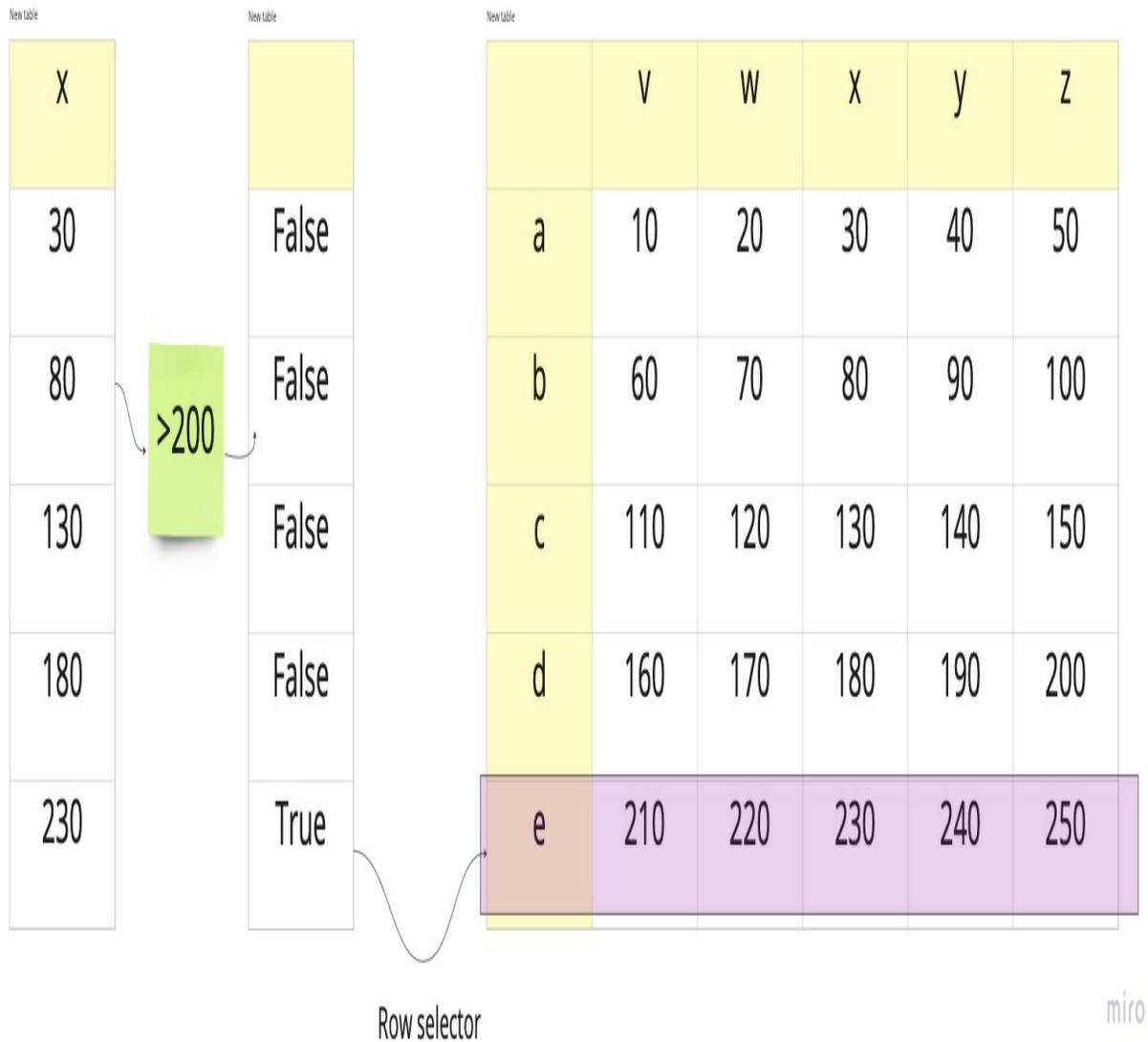
miro

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]#1
```

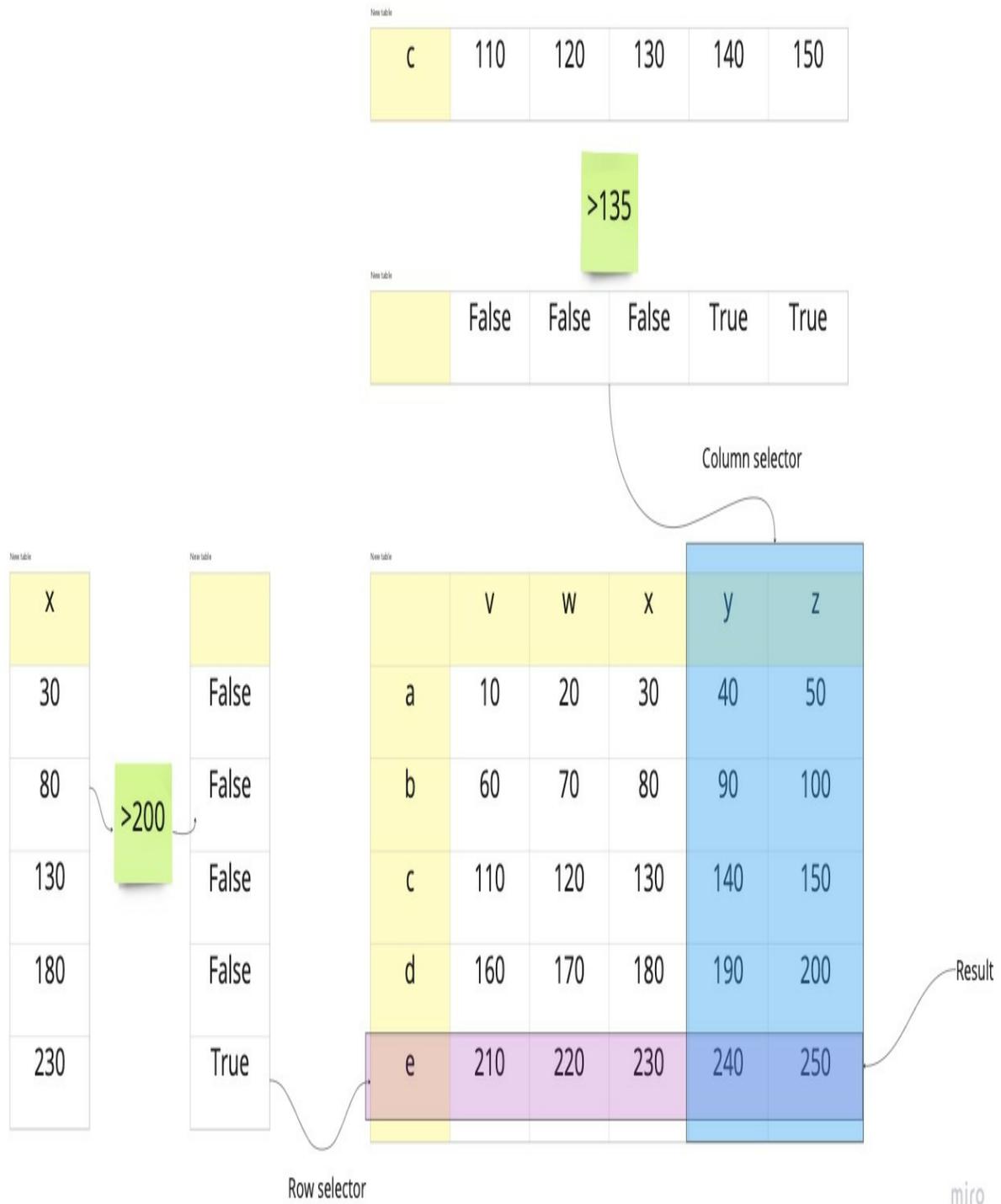
**Figure 2.13. Graphical depiction of df.loc[df['x']>200]**



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

```
df.loc[df['x']>200,#1  
       df.loc['c'] > 135]#2
```

**Figure 2.14. Graphical depiction of `df.loc[df['x']>200, df.loc['c'] > 135]`**



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 135.

I can also dial it back, saying that I'm interested in the row b, but only where c is greater than 135:

```
df.loc['b', #1  
      df.loc['c']>135]#2
```

**Figure 2.15. Graphical depiction of `df.loc['b', df.loc['c'] > 135]`**

New table

c	110	120	130	140	150
---	-----	-----	-----	-----	-----

>135

New table

	False	False	False	True	True
--	-------	-------	-------	------	------

Column selector

New table

	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

Row selector

Result

miro

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 y, to 123. I can do that with:

```
df.loc['b', #1  
      'y' #2  
      ] = 123
```

**Figure 2.16. Graphical depiction of `df.loc['b', 'y'] = 123`**

Column selector

Row selector

Replace the current value with 123

New table

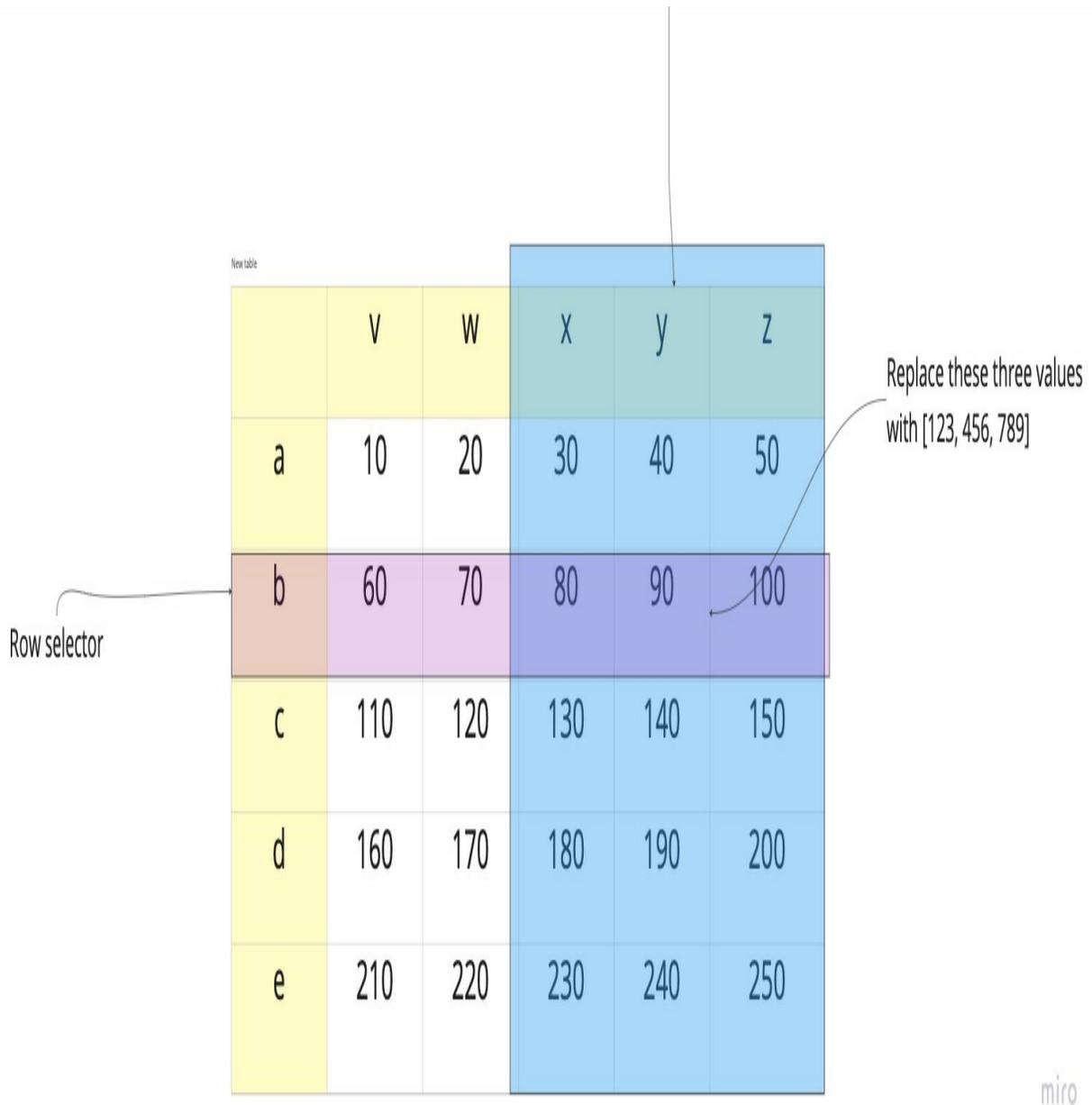
	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

miro

What if I want to set new values in row b, where row c is greater than 125? I can assign a list (or NumPy array, or Pandas series) of three items, matching the three elements my query matched:

```
df.loc['b', #1
      df.loc['c'] > 125#2
      ] = [123, 456, 789]
```

**Figure 2.17. Graphical depiction of `df.loc['b', df.loc['c'] > 125] = [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 divisible by 3:

```
df.loc['b', #1  
        df.loc['c'] % 3 == 0 #2  
        ] *= 2 #3
```

**Figure 2.18. Graphical depiction of `df.loc['b', df.loc['c'] % 3 == 0] *= 2<3>`**

New table

c	110	120	130	140	150
---	-----	-----	-----	-----	-----

%3==0

New table

	False	True	False	False	True
--	-------	------	-------	-------	------

Column selector

New table

	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

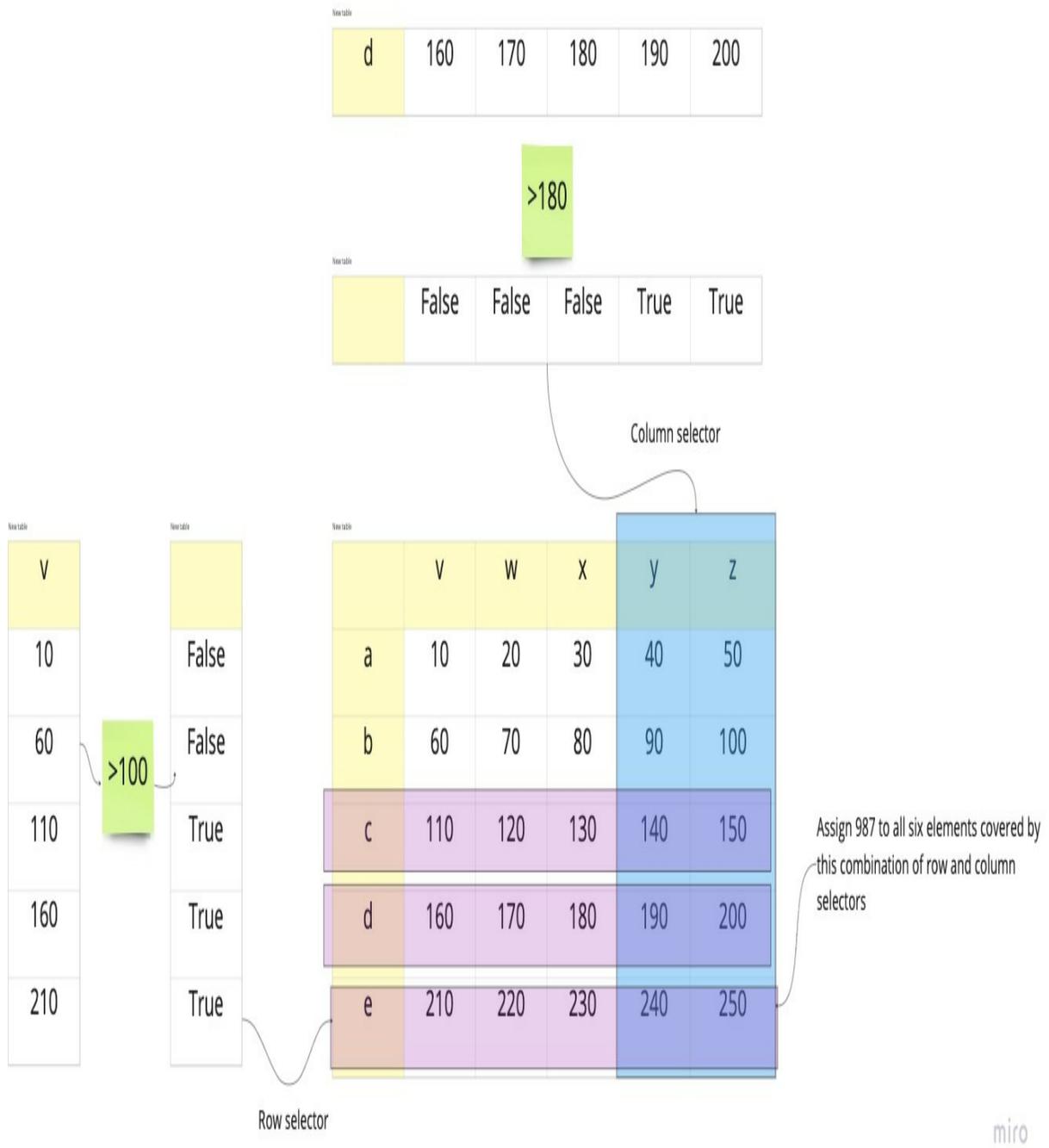
Row selector

Double these values, and replace the existing ones

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

```
df.loc[df['v'] > 100, #1  
       df.loc['d'] > 180 #2  
       ] = 987
```

**Figure 2.19. Graphical depiction of `df.loc[df['v'] > 100, df.loc['d'] > 150] = 987`**



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. I'd like you to do that by creating a new data frame, and adding it to the existing one. This new data frame should contain three products (including product ID, name, wholesale price, and retail price):

- Phone, with an ID of 24, a wholesale price of 200, and a retail price of 500
- Apple, with an ID of 16, a wholesale price of 0.5, and a retail price of 1
- Pear, with an ID of 17, a wholesale price of 0.6, and a retail price of 1.2

Because these are new products, don't 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 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 builtin, 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.

I created a new data frame 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.

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

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

**Figure 2.20. Graphical depiction of `pd.concat([df, new_products])`**

	product_id	name	wholesale_price	retail_price	sales
0	23	computer	500	1000.0	100.0
1	96	Python Workout	35	75.0	1000.0
2	97	Pandas Workout	35	75.0	500.0
3	15	banana	0.5	1.5	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

df

new\_products

miro

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  sa
0           23  computer           500.0       1000.0    10
1           96 Python Workout           35.0         75.0   100
2           97 Pandas Workout           35.0         75.0    50
3           15   banana              0.5          1.0    20

```

4	87	sandwich	3.0	5.0	30
5	24	phone	200.0	500.0	
6	16	apple	0.5	1.0	
7	17	pear	0.6	1.2	

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 a list of rows as the row selector, and the `sales` column's name as the column selector:

```
df.loc[[5,6,7], 'sales'] #1
```

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 change 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.

**Figure 2.21. Graphical depiction of `df.loc[[5,6,7], 'sales'] = [100, 200, 75]`**

	product_id	name	wholesale_price	retail_price	sales
0	23	computer	500	1000.0	100.0
1	96	Python Workout	35	75.0	1000.0
2	97	Pandas Workout	35	75.0	500.0
3	15	banana	0.5	1.5	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

row selector

column selector

Assign [100, 200, 75] to these three cells

miro

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
```





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') #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.

On smaller data frames, query can not only be overkill, but can actually slow your code down. However, when you work on data frames with more than 10,000 rows, query can be significantly faster than the traditional way of writing queries. Moreover, it can use far less memory. We'll look at query in greater depth in Chapter 11.

## 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 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 better than average:

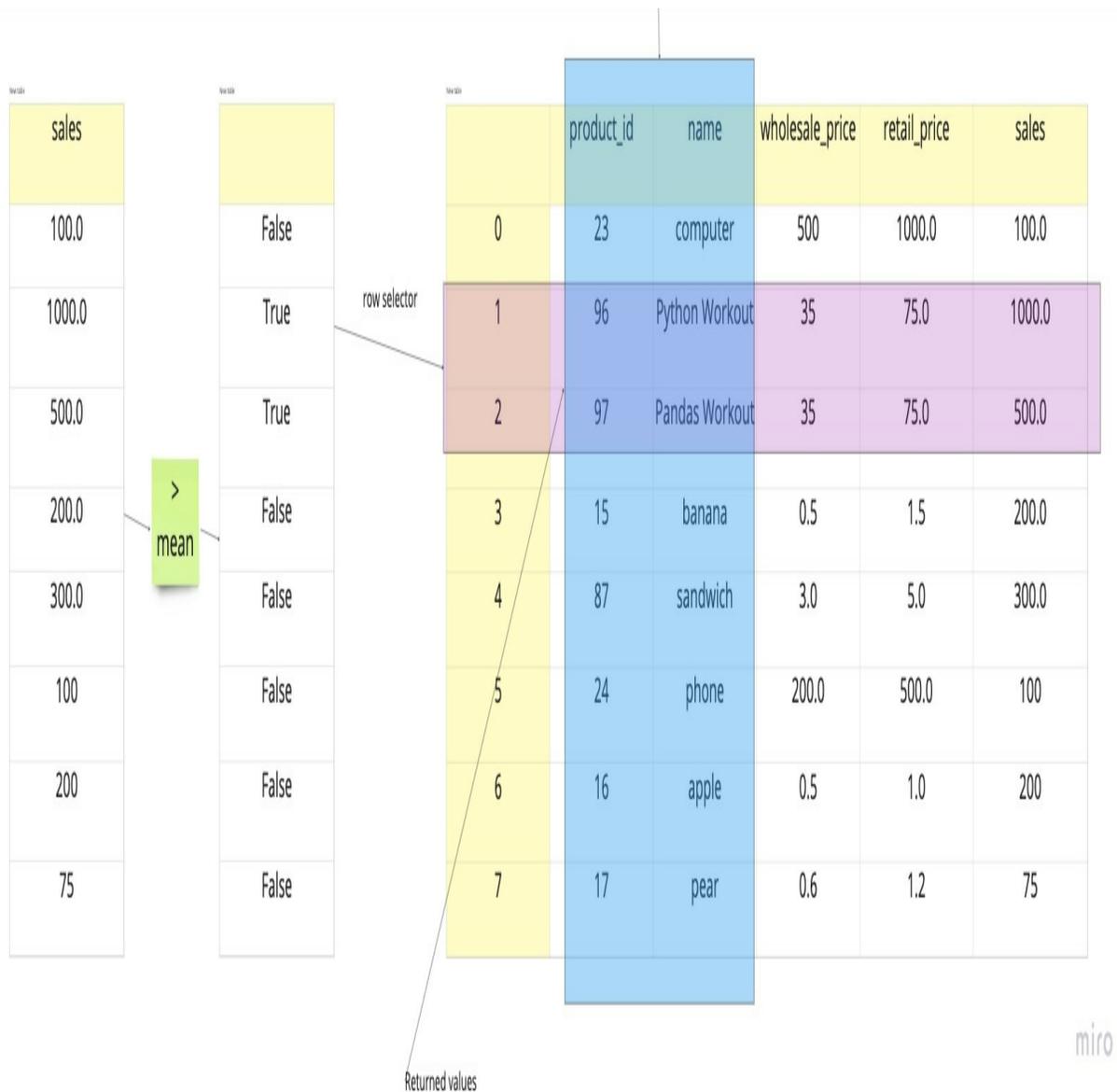
```
df.loc[df['sales'] > df['sales'].mean()] #1
```

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 list the columns we want in the second argument to `loc`, in our column selector:

```
df.loc[df['sales'] > df['sales'].mean(), #1  
       ['product_id', 'name']] #2
```

Sure enough, this produces the desired output.

**Figure 2.22. Graphical depiction of `df.loc[df['sales'] > df['sales'].mean(), ['product_id', 'name']]`**



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()')
```

To get only the `product_id` and `name` columns, we'll need to apply square brackets to the result of `df.query`:

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

## 2.5.2 Solution



- 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 name, 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

We've already seen how the mean, standard deviation, and median can help us to understand our data. But they describe the bulk of our data, trying to summarize where the majority of values lie. But sometimes, it's useful to look at the unusual values:

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

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 at 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 Exercise 6. 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 Discussion

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

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.

**Figure 2.23. Graphical depiction of creating a data frame via a dictionary**

New table

dict keys become  
column names

	trip_distance	passenger_count
0	1.63	1
1	0.46	1
2	0.87	1
3	2.13	1
4	1.40	1
5	1.40	1
6	1.80	1
7	11.90	4

each dict value (a series)  
becomes a column

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'].qu
```

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  $1.5 * \text{the IQR}$ . This is how that looks in Pandas:

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

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
```

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'].q
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.

## 2.6.2 Solution

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

df = DataFrame({'trip_distance': trip_distance,
    'passenger_count': passenger_count}) #1

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] #1
df[df['trip_distance']
    > df['trip_distance'].quantile(0.25) + 1.5*iqr] #2
df['passenger_count'][df['trip_distance']
    > df['trip_distance'].quantile(0.25) + 1.5*iqr].mean() #3
```

You can explore an abridged version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
20%3D%20%7B'trip_distance'%3A%20%7B0%3A%201.63,%201%3A%200.46,%20
87,%203%3A%202.13,%204%3A%201.4%7D,%0A%20'passenger_count'%3A%20%
,%201%3A%201,%202%3A%201,%203%3A%201,%204%3A%201%7D%7D%0A%20%0Adf
DataFrame%28data%29%0A%0Aiqr%20%3D%20df%5B'trip_distance'%5D.quan
75%29%20%20%20%20-%20df%5B'trip_distance'%5D.quantile%280.25%29%0
df%5B'trip_distance'%5D%0A%20%20%20%20%3C%20df%5B'trip_distance'%
quantile%280.25%29%20-%201.5*iqr%5D%20%0Adf%5Bdf%5B'trip_distance
20%20%20%20%3E%20df%5B'trip_distance'%5D.quantile%280.25%29%20%2B
iqr%5D%20%20%0Adf%5B'passenger_count'%5D%5Bdf%5B'trip_distance'%5
20%20%20%3E%20df%5B'trip_distance'%5D.quantile%280.25%29%20%2B%20
5D.mean%28%29%20%0A&d=2022-11-08&lang=py&v=v1
```

## 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?
- 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. 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.

#### **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 find 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.

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.

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. Sometimes, you'll want to remove the NaN values. But other times, such as in Exercise 13 (below), you'll want to assign values based on their neighbors.



### 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. If any of the columns have a lower count result than the others, it's because they 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 original ones. The values might be wrong, 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 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()
```

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
```

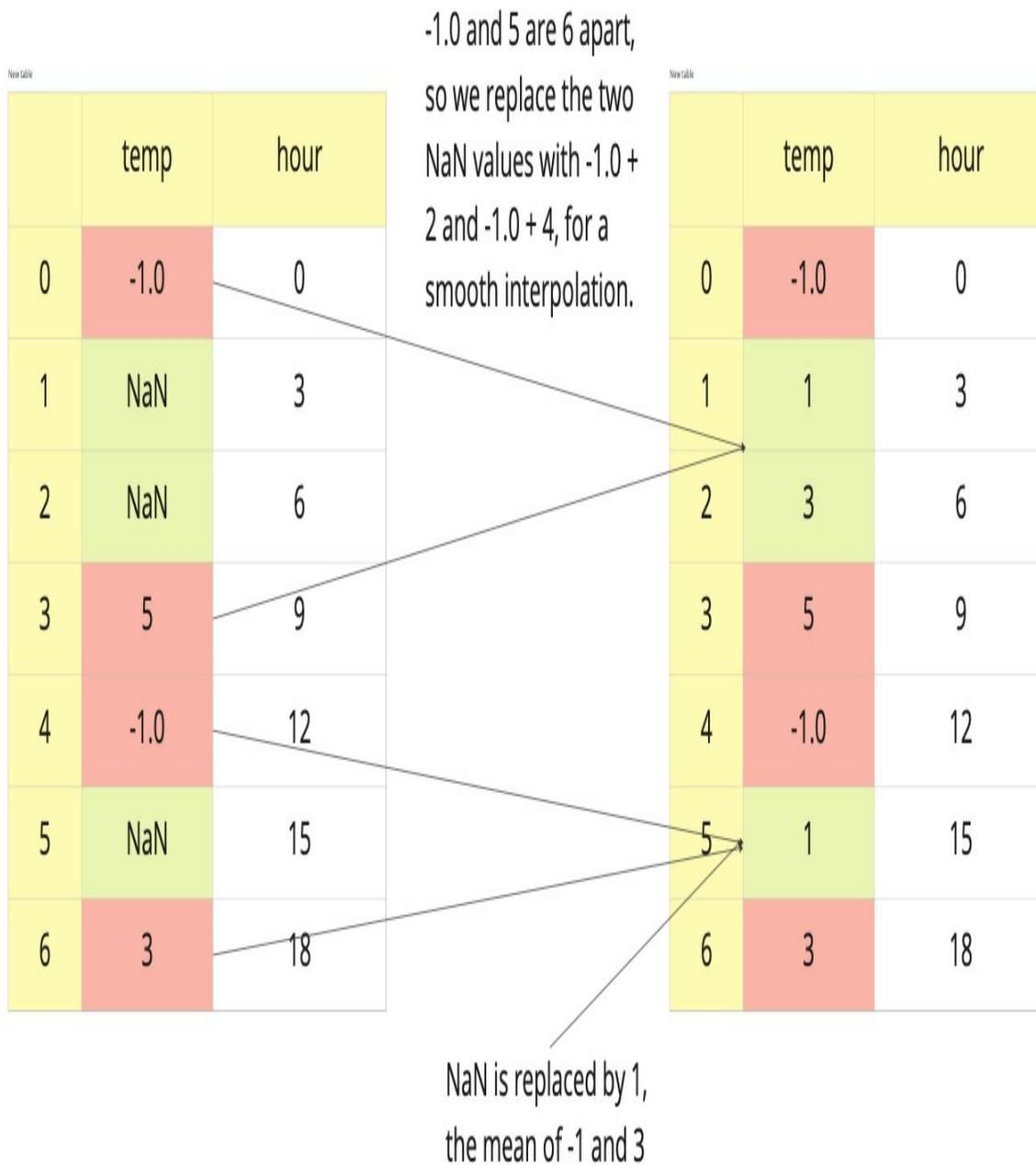
Notice that this query has several pieces:

- 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`.

**Figure 2.24. Graphical depiction of `interpolate`**



miro

By default, `interpolate` fills any `NaN` value with the average of the numbers that come just before and after it. So if row 3 has a temperature of -1, row 4 has is `NaN`, and row 5 has a temperature of 5, `interpolate` will replace `NaN` with a value of 2.0. If you have two `NaN` values in a row, then `interpolate` will replace each `NaN` with half of the distance between the preceding non-`NaN`

value and the succeeding non-NaN value.



#### Note

By passing a value to the method parameter, you can instruct `interpolate` to use a different system for interpolation. For example, if you pass `method='nearest'`, then NaN values will be replaced by the closest non-NaN value. Other methods are discussed in the documentation, at <http://mng.bz/MBo7>.

Since temperature values don't vary all that much from hour to hour, and can be assumed to rise and fall on a continuum, I used the default "linear" method, and felt like I was likely to be close to the actual values. By contrast, hourly temperature readings from the oven in your kitchen cannot be interpolated reliably in such a way. Before you use `interpolate`, consider if it's an appropriate way to fill in NaN values.

## 2.7.2 Solution

```
s = pd.read_csv('data/nyc-temps.txt').squeeze() #1
df = DataFrame({'temp': s,
               'hour': [0, 3, 6, 9, 12, 15, 18, 21] * 91}) #2

df.loc[(df['hour'] == 3) | (df['hour'] == 6), 'temp'] = NaN #3
df = df.interpolate() #4

df['temp'].describe() #5
```

You can explore an abridged version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
20%3D%20%7B'temp'%3A%20%7B0%3A%20-1.0,%201%3A%20np.nan,%202%3A%20
203%3A%205.0,%204%3A%20-1.0,%205%3A%20np.nan,%206%3A%203.0%7D,%0A
%3A%20%7B0%3A%200,%201%3A%203,%202%3A%206,%203%3A%209,%204%3A%201
%2015,%206%3A%2018%7D%7D%0A%20%0Adf%20%3D%20DataFrame%28data%29%0
%5B%28df%5B'hour'%5D%20%3D%203%29%20%7C%20%28df%5B'hour'%5D%20
%29,%20'temp'%5D%20%3D%20np.nan%0Adf%20%3D%20df.interpolate%28%29
f%5B'temp'%5D.describe%28%29%20%0A&d=2022-11-08&lang=py&v=v1
```

## 2.7.3 Beyond the exercise

- How does the behavior of `interpolate` change if we use `method='nearest'`?
- 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 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 perfect 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. Retrieving with this syntax, is also going to be less efficient than using `loc` with the "row selector, column selector" 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, broken up across lines:

```
df.loc[  
    df['temp'] < 0, #1
```

```
    'temp' #2  
] = 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.

## 2.8.2 Solution

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

You can explore an abridged version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A  
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0  
20%3D%20%7B'temp'%3A%20%7B0%3A%20-1.0,%201%3A%20np.nan,%202%3A%20  
203%3A%205.0,%204%3A%20-1.0,%205%3A%20np.nan,%206%3A%203.0%7D,%0A  
%3A%20%7B0%3A%200,%201%3A%203,%202%3A%206,%203%3A%209,%204%3A%201  
%2015,%206%3A%2018%7D%7D%0A%20%0Adf%20%3D%20DataFrame%28data%29%0  
loc%5B%28df%5B'hour'%5D%20%3D%3D%203%29%20%7C%20%28df%5B'hour'%5D  
3D%206%29,%20'temp'%5D%20%3D%20np.nan%0Adf%20%3D%20df.interpolate  
20%0A%0Adf.loc%5Bdf%5B'temp'%5D%20%3C%200,%20'temp'%5D%20%20%3D%2  
&d=2022-11-08&lang=py&v=v1
```

## 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, data frames contain actual, useful 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 a variety of formats, emphasizing CSV files, because they're so common. We'll look at ways in which we can not only read from such 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
<code>pd.read_csv</code>	returns a new data frame based on CSV input	<pre>df = pd.read_csv('myfile.csv')</pre>	<a href="http://mng.bz">http://mng.bz</a>
<code>df.to_csv</code>	Writes a data frame to a CSV-formatted file or string	<pre>df.to_csv('myfile.csv')</pre>	<a href="http://mng.bz">http://mng.bz</a>

<code>pd.read_json</code>	returns a new data frame based on JSON input	<code>df = pd.read_json('myfile.json')</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.corr</code>	Show the correlations among the columns	<code>df.corr()</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.dropna</code>	Return a new data frame, without any NaN values	<code>df.dropna()</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.loc</code>	Retrieve selected rows and columns	<code>df.loc[df['trip_distance'] &gt; 10, 'passenger_count']</code>	<a href="http://mng.b">http://mng.b</a>
<code>pd.read_html</code>	returns a list of data frames based on HTML input	<code>df = df.read_html('@</code>	<a href="http://mng.b">http://mng.b</a>
	returns a sorted (descending		

<code>s.value_counts</code>	frequency) series counting how many times each value appears in s	<code>s.value_counts()</code>	<a href="http://mng.b">http://mng.b</a>
<code>s.round</code>	returns a new series based on s, in which the values are rounded to the specified number of decimals.	<code>s.round(2)</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.memory_usage</code>	Indicates how many bytes are being used by a data frame and its associated data	<code>df.memory_usage()</code>	<a href="http://mng.b">http://mng.b</a>

### 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 <https://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.

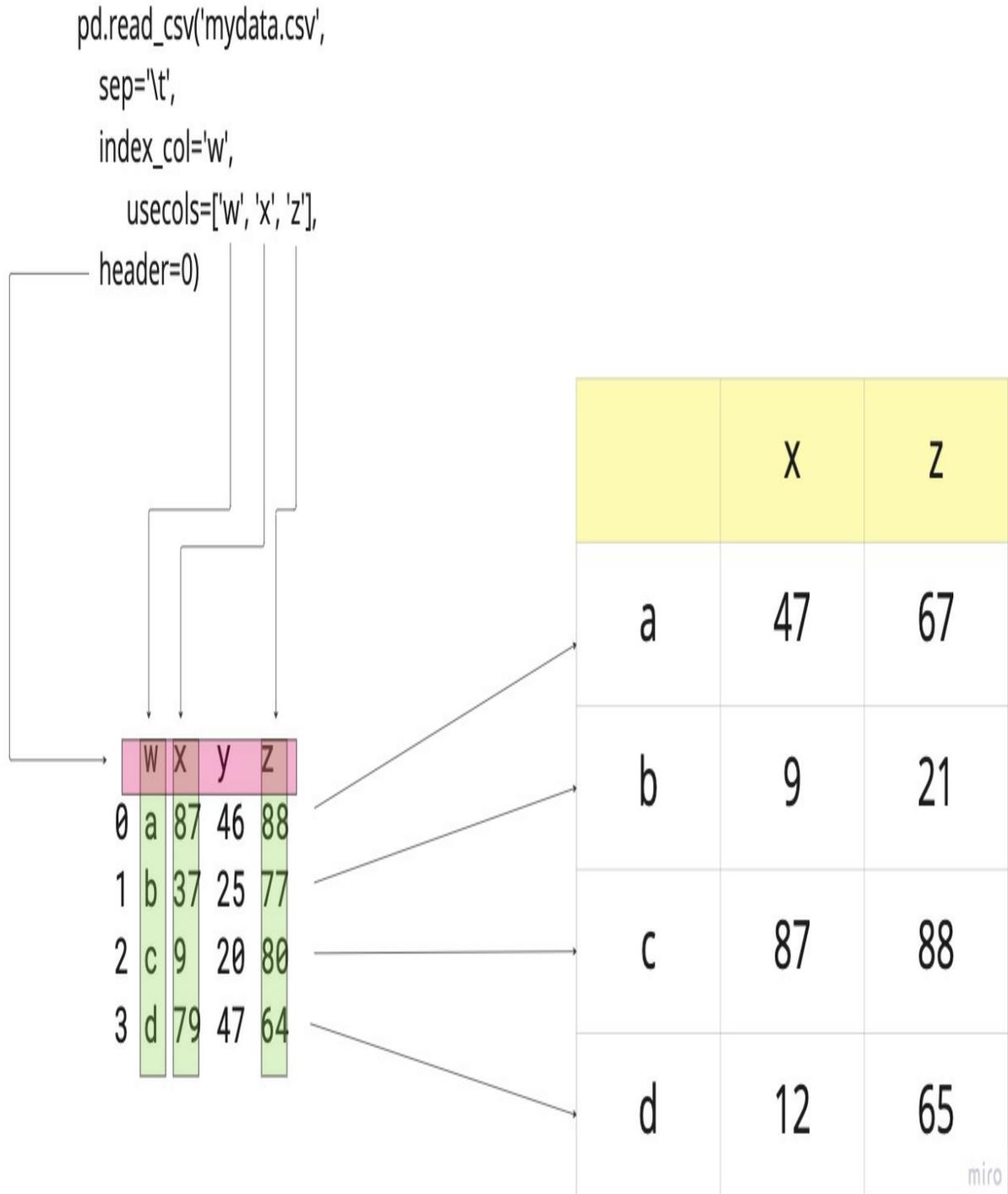
Rather than take a stand on how CSV files should be formatted, Pandas tries to be open and flexible. When we read from a CSV file (with `pd.read_csv`) or write a data frame to CSV (with `df.to_csv`), you can choose from many, **many** parameters, each of which can affect the way in which it is written. 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
- `index_col`, which column, if any, should be set to be the index of our data frame
- `usecols`, which columns from the file should be included in the data frame

For example, I could say:

```
pd.read_csv('mydata.csv', #1
            sep='\t', #2
            index_col='w', #3
            usecols=['w', 'x', 'z'], #4
            header=0) #5
```

**Figure 3.1. Graphical depiction of reading a CSV file with common keyword arguments**



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 explore a number of these parameters in this book, but many that we don't cover might 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 examine a data dictionary when starting to analyze the data. In many cases, the dictionary will give you insights that will help you decide what and how you want to read into Pandas.

## 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. In particular, we'll be looking at several columns from the data set:

- `passenger_count`, the number of passengers who took that taxi ride
- `trip_distance`, the distance traveled, in miles
- `total_amount`, the total owed to the driver, including the fare, tolls, taxes, and tips
- `payment_type`, an integer 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 four columns mentioned above: `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 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, via the `loc` accessor:

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

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

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



Note

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. For example:

```
s = Series([10, 20, np.NaN, 40, 50])
s.count()
```

The result of the above code is 4. However:

```
df = DataFrame([[10, 20, np.NaN, 40],
                [50, np.NaN, np.NaN, np.NaN],
                [np.NaN, 60, 70, 80]],
               index=list('abc'),
               columns=list('wxyz'))
df.count()
```

The result of the above code is a series, showing the number of non-NaN values are in each column:

```
w    2
x    2
y    1
z    2
dtype: int64
```

Right now, 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, #1
        'passenger_count' #2
       ].count() #3
```

Sure enough, this tells us that in January 2019, there were 9 trips with more than 8 people. (I hope that these took place in larger-than-usual taxis.)

**Figure 3.2. Graphical depiction of selecting rows where `passenger_count > 8`, then invoking `sum`.**



Next, how many taxi rides in January 2019 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 more 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, #1  
       'passenger_count' #2  
       ].count() #3
```

It turns out that there 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?

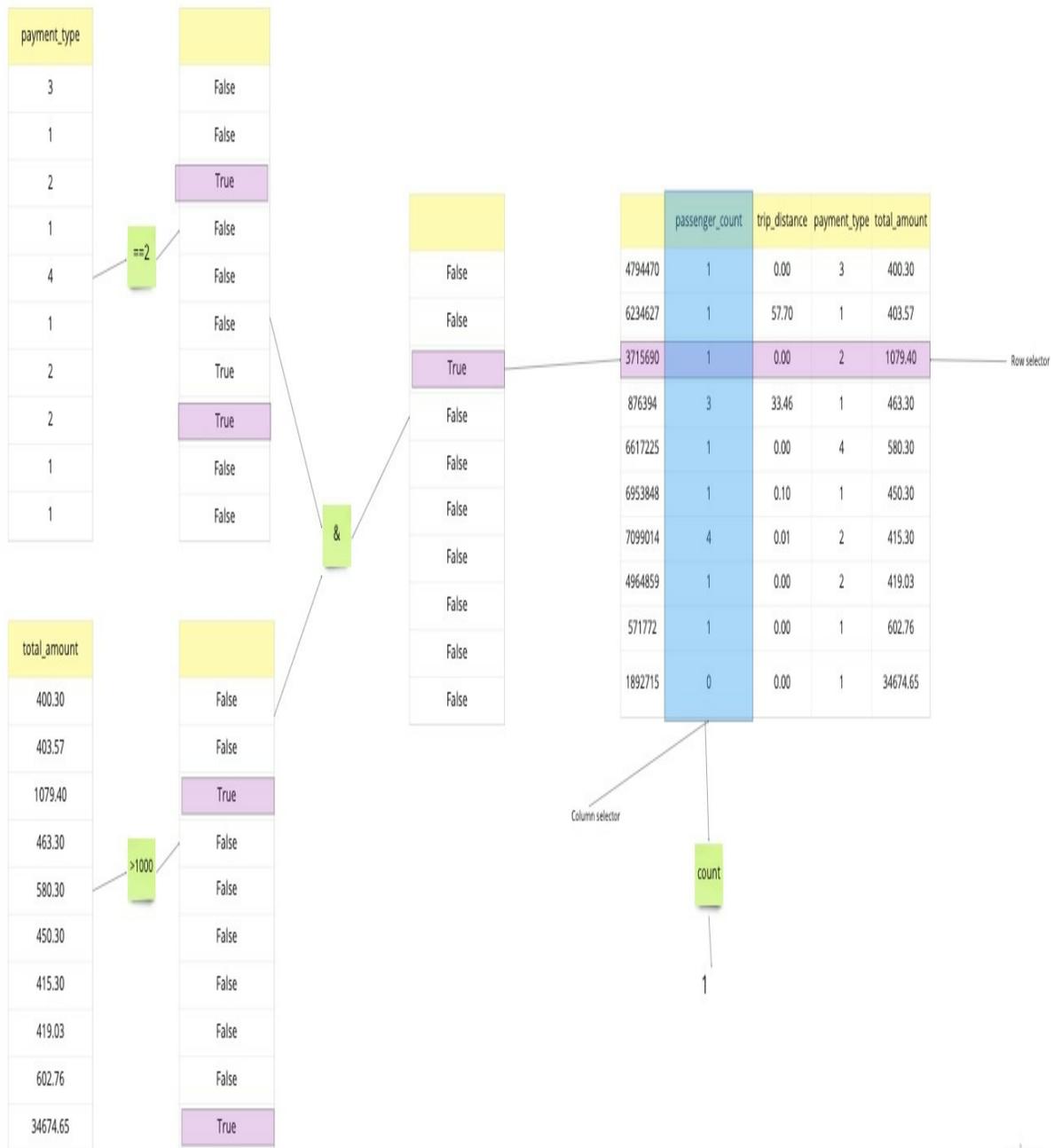
This question is a bit harder to answer, because we're going to need to combine two different boolean series: The first will find rides in which the payment method was cash (i.e., 2), and the second series will find where `total_amount` is greater than 1000. We can then join the two together using `&`, 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' #2  
       ].count() #3
```

**Figure 3.3. Graphical depiction of selecting rows where `payment_type == 2` and `total_amount > 1000`, and counting the elements of `passenger_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 2019?

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

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

We'll apply this boolean series as a mask index on the `total_amount` column, and run `count`:

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

The total is 7,131, meaning 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()
```

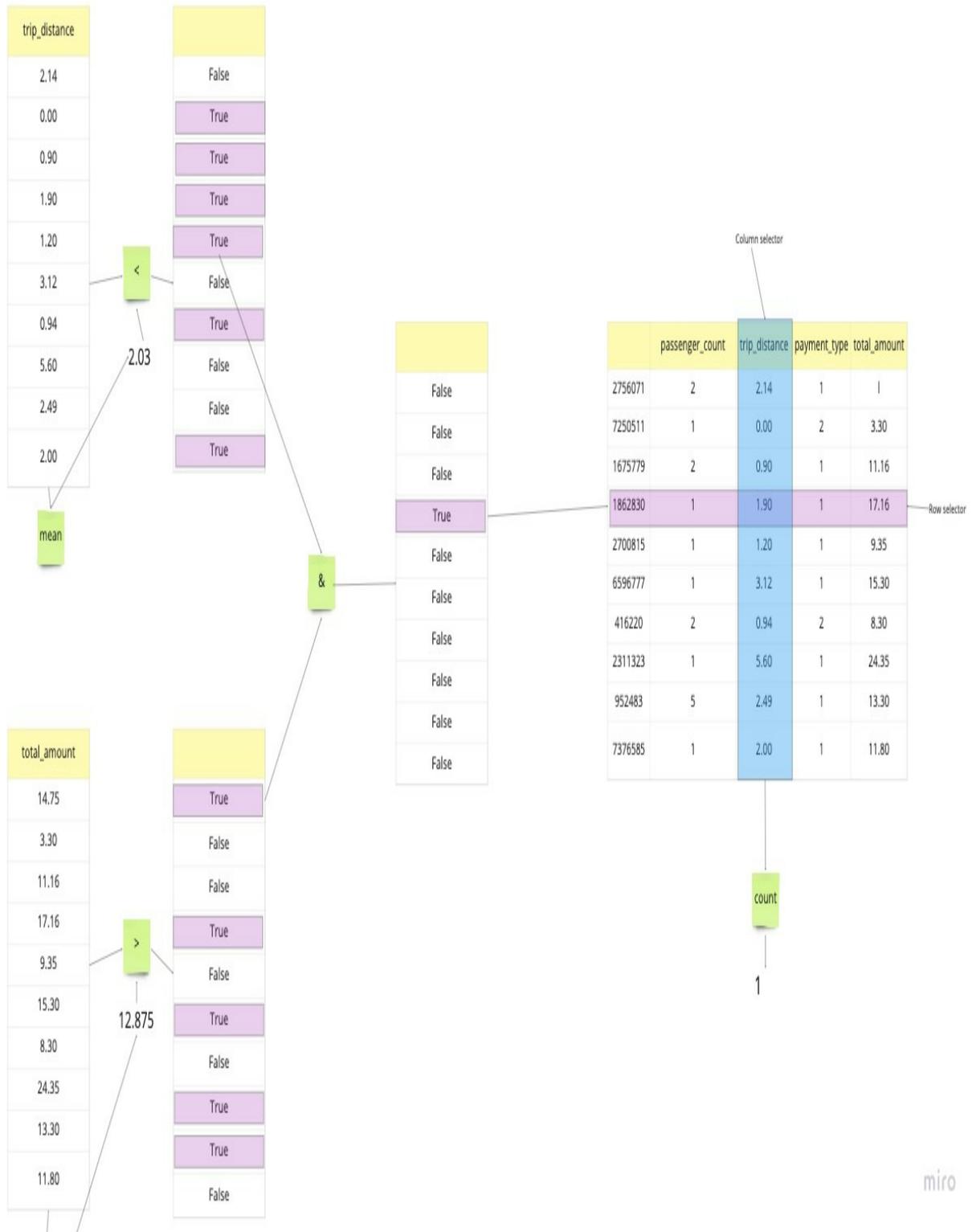
We'll combine them using `&`, to get a new boolean series:

```
(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()) & #1  
       (df['total_amount'] > df['total_amount'].mean()), #2  
       'trip_distance' #3  
       ].count() #4
```

**Figure 3.4. Graphical depiction of counting rows where where trip\_distance is less than the mean, but total\_amount is greater than the mean**



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



### 3.2.3 Beyond the exercise

- Repeat this exercise, but using the query method rather than a boolean index and loc.
- How many of the rides that cost less than 0 were 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 and 2020, and what is the difference between these two figures?
- How much money (in total) was collected in 2019 and 2020, and what was the difference between these two figures?
- Did the proportion of trips with more than passenger change dramatically?
- Did people use cash (i.e., payment\_type of 2) 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 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])
```

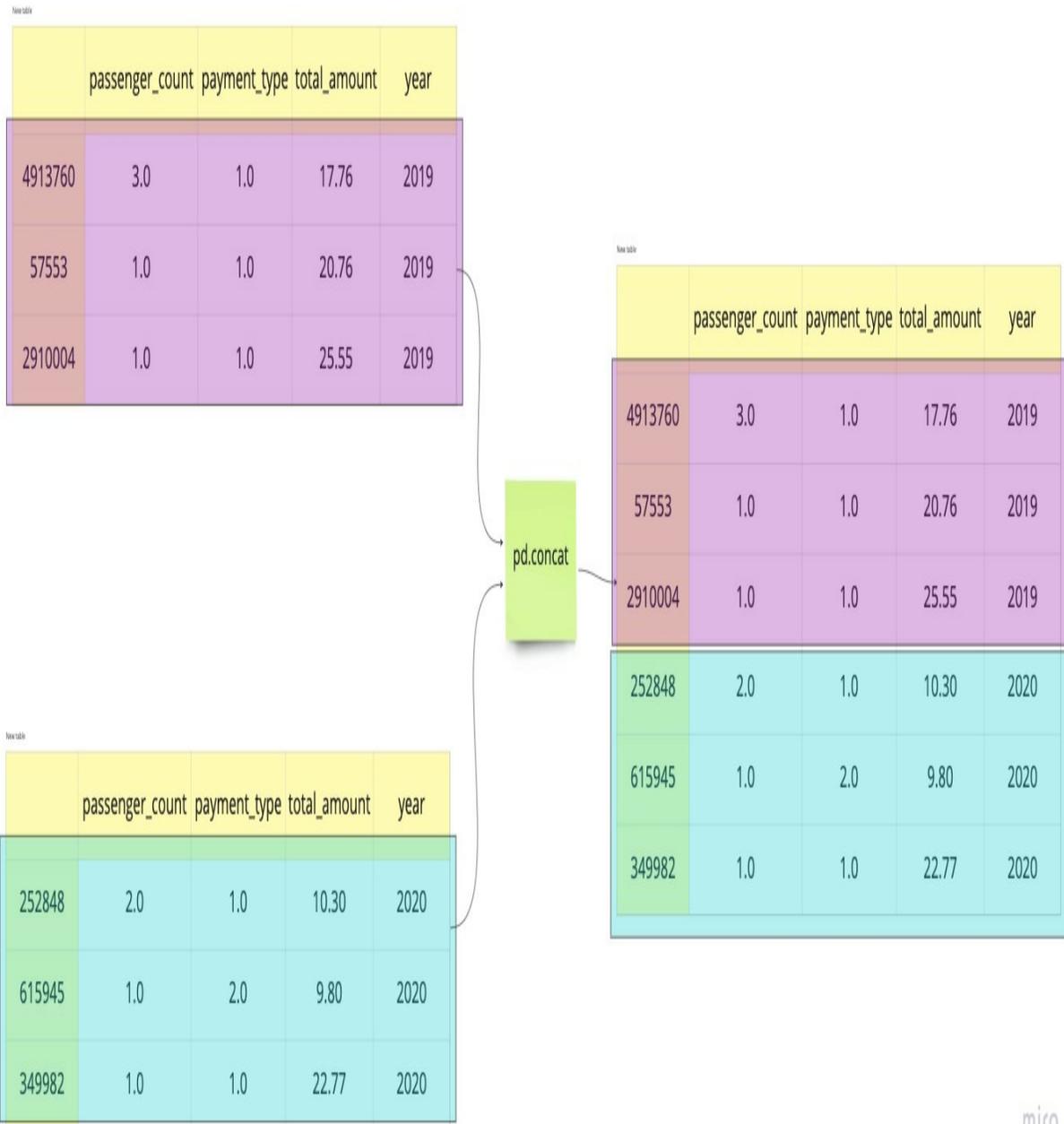
If we were only interested in getting aggregate answers, that would be enough. But we want to separate the answers by year via a year column. My preferred solution to this is to add a new column to each of the file-based data frames, and then concatenate them:

```
df_2019_jul = pd.read_csv('../data/nyc_taxi_2019-07.csv',
                          usecols=['passenger_count',
                                   'total_amount', 'payment_type'])
df_2019_jul['year'] = 2019 #1
```

```
df_2020_jul = pd.read_csv('../data/nyc_taxi_2020-07.csv',
                          usecols=['passenger_count',
                                    'total_amount', 'payment_type'])
df_2020_jul['year'] = 2020 #2

df = pd.concat([df_2019_jul, df_2020_jul]) #3
```

**Figure 3.5. Concatenating two data frames into a single one**



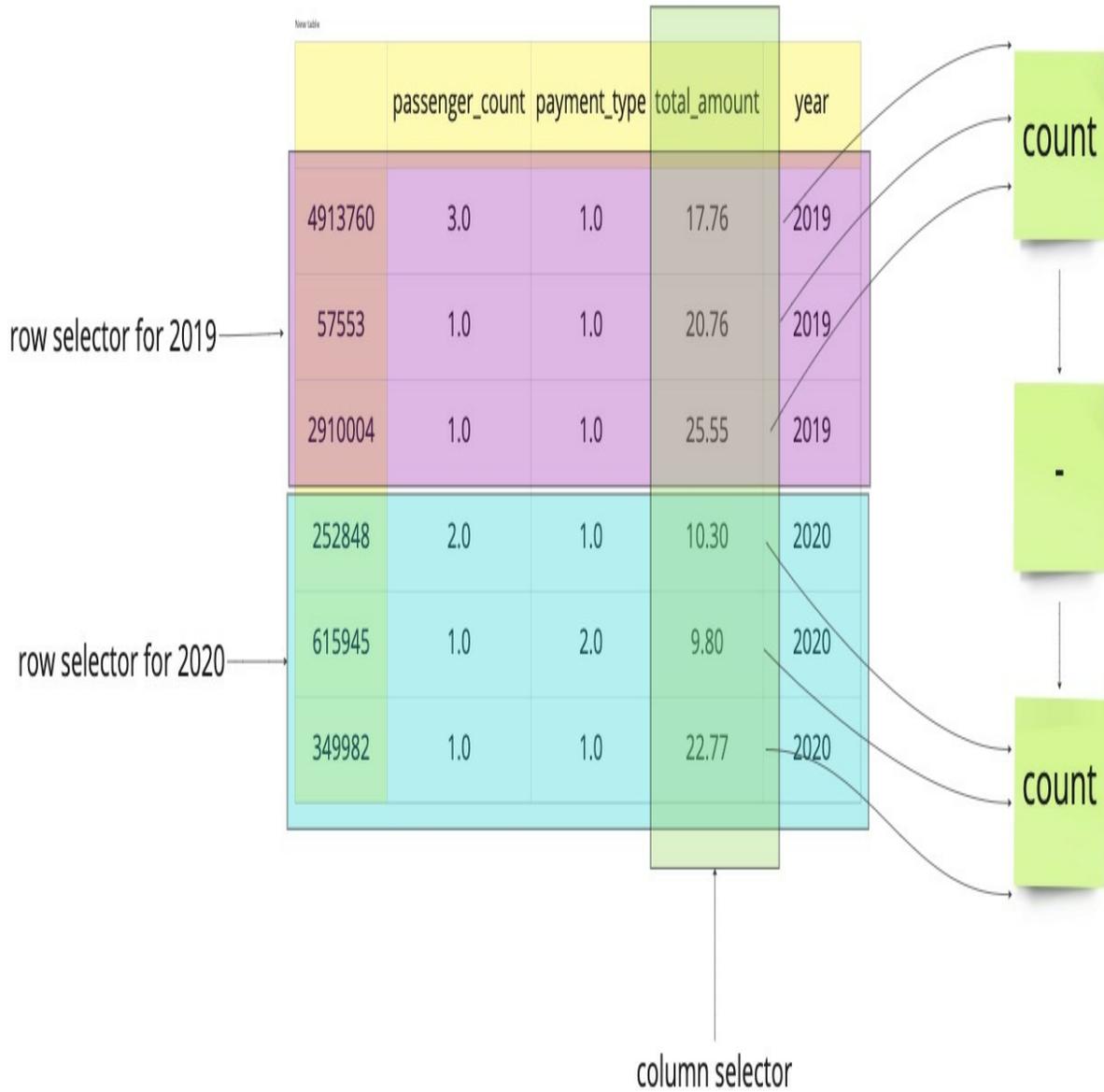
miro

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['y
```

```
== 2020, 'total_amount'].count()
```

Figure 3.6. Comparing number of rides in 2019 with 2020

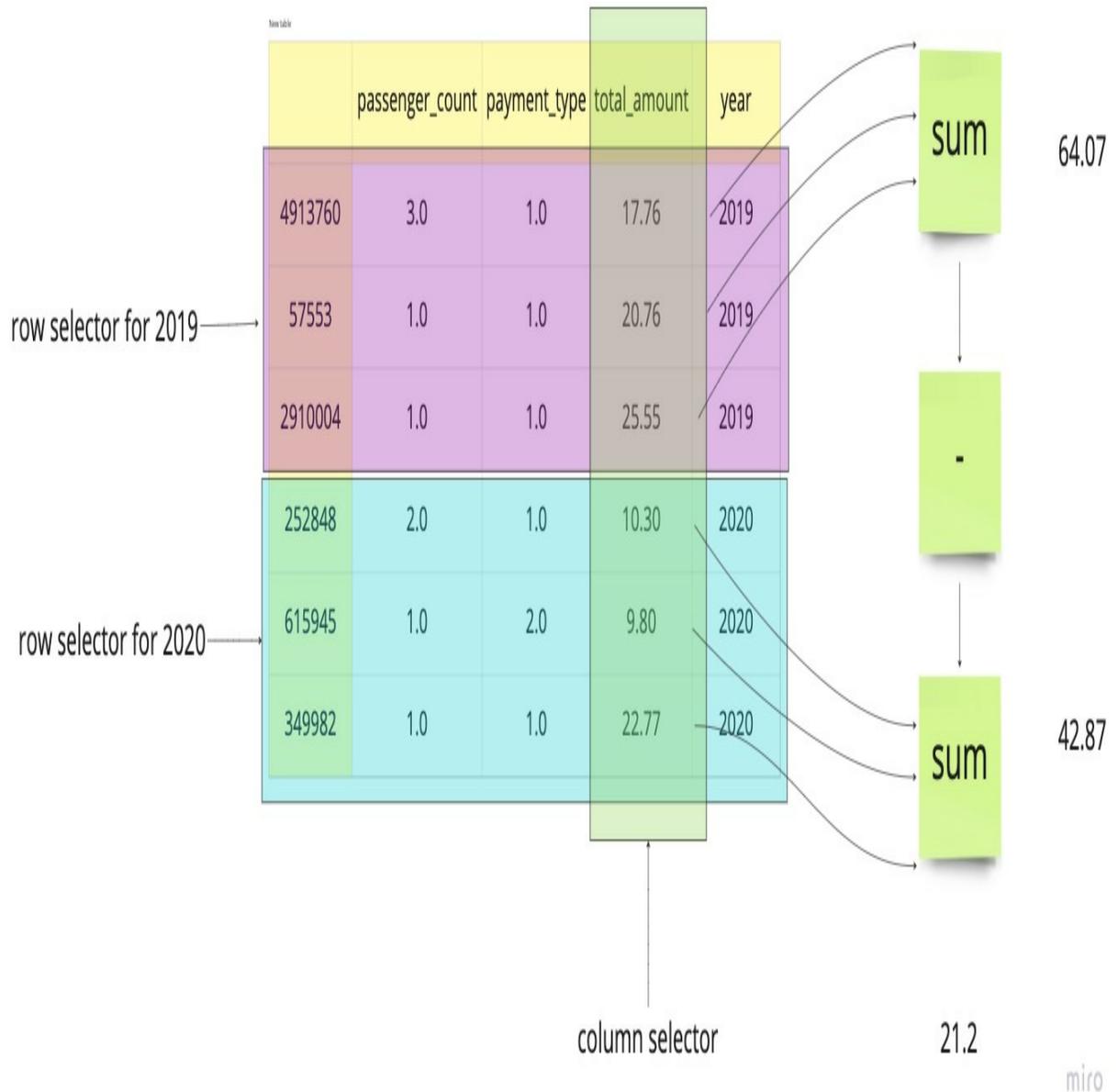


The result is 5,510,007. That's right—in July 2020, New Yorkers took 5.5 million fewer taxi rides than in 2019. Now, that's a lot of taxi rides. But how much less money did they make as a result? Now, instead of using count, we'll use sum to total up the numbers before we subtract them:

```
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 by the huge sums.

**Figure 3.7. Comparing the total amount earned in 2019 with 2020**



**i Note**

If you're bothered by the long number of numbers after the decimal point, you can always use the round method on a series to keep it limited to two

digits.

```
df.loc[df['year'] == 2019, 'total_amount'].sum().round(2) -  
df.loc[df['year'] == 2020, 'total_amount'].sum().round(2)
```

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:

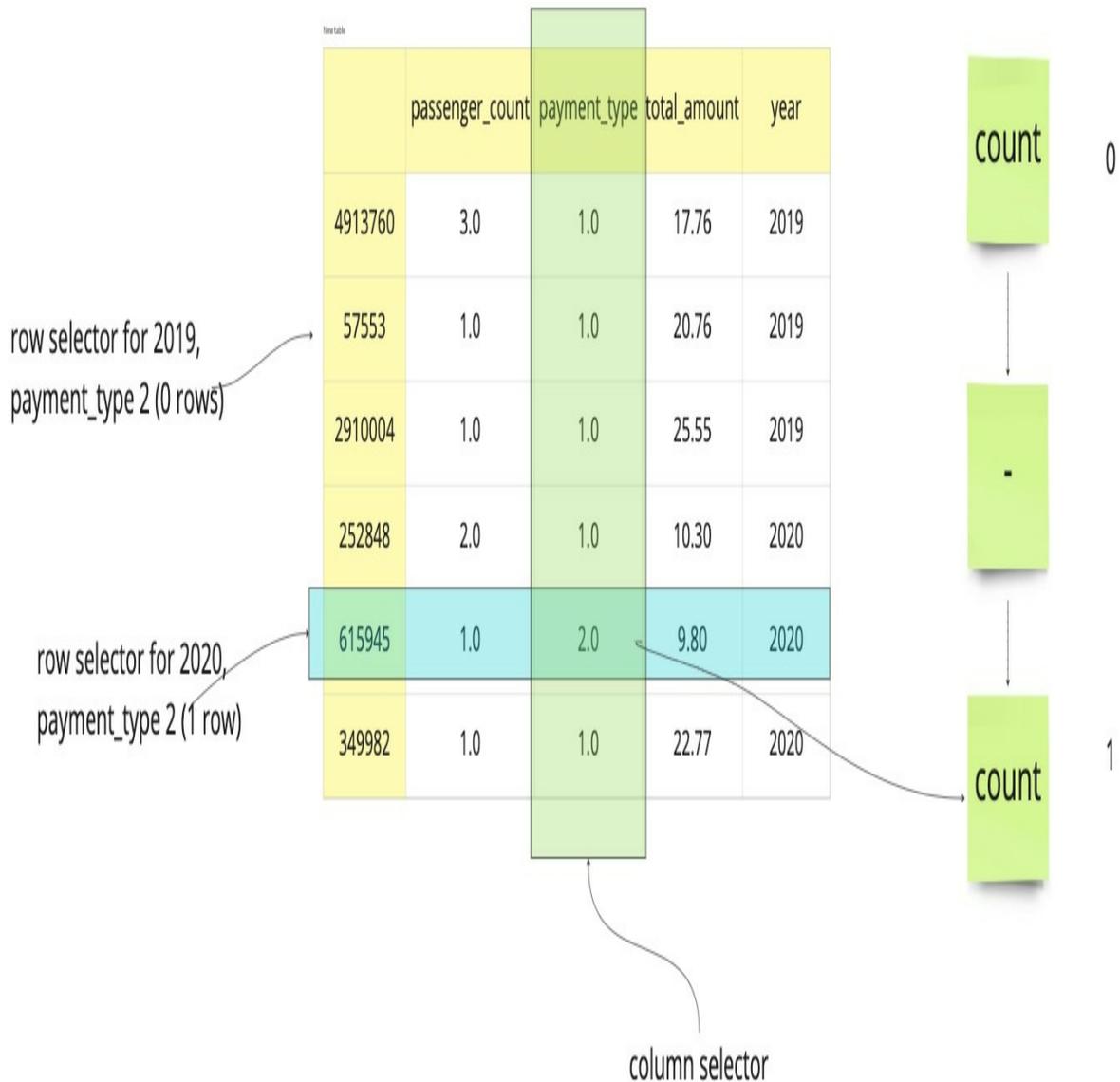
```
df.loc[(df['year'] == 2019) &  
        (df['passenger_count'] > 1), 'passenger_count'].count() /  
df.loc[df['year'] == 2019, 'payment_type'].count()  
  
df.loc[(df['year'] == 2020) &  
        (df['passenger_count'] > 1), 'passenger_count'].count() /  
df.loc[df['year'] == 2020, 'payment_type'].count()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

I get about 28% in 2019, and about 21% in 2020. Meaning that people were less likely to share a taxi during the pandemic. Another interpretation would be that there were fewer family vacations and trips in New York, raising the proportion of single passengers commuting to work.

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()
```

**Figure 3.8. Comparing the number of cash payments 2019 with 2020**



miro

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. One theory, floated by members of my family, is that the only people going to work during the pandemic were those who had to do so, the so-called "essential workers." They tend to earn less money, and use more cash. Regardless of whether

that's the reason, the numbers bear out the increased use of cash.

### 3.3.2 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()
```

You can explore this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
3D%20DataFrame%28%7B'passenger_count'%3A%20%7B3052891%3A%205.0,%0
260%3A%20np.nan,%0A%20%202080921%3A%201.0,%0A%20%201691203%3A%201
203257780%3A%202.0,%0A%20%204454807%3A%201.0,%0A%20%206250323%3A%
20%204105125%3A%205.0,%0A%20%20789467%3A%202.0,%0A%20%204930582%3
%0A%20%204873064%3A%201.0,%0A%20%20502123%3A%201.0,%0A%20%2023209
0,%0A%20%204967113%3A%201.0,%0A%20%20750140%3A%201.0,%0A%20%20409
201.0,%0A%20%201263539%3A%205.0,%0A%20%206049031%3A%201.0,%0A%20%
```

%3A%202.0,%0A%20%201080571%3A%201.0,%0A%20%205743779%3A%201.0,%0A  
3072%3A%201.0,%0A%20%203200119%3A%201.0,%0A%20%204884860%3A%202.0  
206106585%3A%201.0%7D,%0A%20'payment\_type'%3A%20%7B3052891%3A%201  
%20771260%3A%20np.nan,%0A%20%202080921%3A%201.0,%0A%20%201691203%  
%0A%20%203257780%3A%201.0,%0A%20%204454807%3A%201.0,%0A%20%206250  
.0,%0A%20%204105125%3A%201.0,%0A%20%20789467%3A%202.0,%0A%20%2049  
201.0,%0A%20%204873064%3A%202.0,%0A%20%20502123%3A%201.0,%0A%20%2  
3A%202.0,%0A%20%204967113%3A%201.0,%0A%20%20750140%3A%201.0,%0A%2  
6%3A%201.0,%0A%20%201263539%3A%201.0,%0A%20%206049031%3A%201.0,%0  
1774%3A%201.0,%0A%20%201080571%3A%201.0,%0A%20%205743779%3A%201.0  
6153072%3A%201.0,%0A%20%203200119%3A%201.0,%0A%20%204884860%3A%20  
%206106585%3A%201.0%7D,%0A%20'total\_amount'%3A%20%7B3052891%3A%20  
20%20771260%3A%2010.78,%0A%20%202080921%3A%2017.8,%0A%20%20169120  
56,%0A%20%203257780%3A%2017.76,%0A%20%204454807%3A%2011.3,%0A%20%  
3A%2035.8,%0A%20%204105125%3A%2015.36,%0A%20%20789467%3A%2014.3,%  
30582%3A%2010.56,%0A%20%204873064%3A%2044.92,%0A%20%20502123%3A%2  
20%202320995%3A%2010.3,%0A%20%204967113%3A%2021.36,%0A%20%2075014  
16,%0A%20%204097766%3A%2016.55,%0A%20%201263539%3A%2016.64,%0A%20  
%3A%2021.23,%0A%20%202851774%3A%2011.6,%0A%20%201080571%3A%2012.9  
05743779%3A%2011.75,%0A%20%206153072%3A%2020.16,%0A%20%203200119%  
,%0A%20%204884860%3A%2019.8,%0A%20%206106585%3A%2050.75%7D,%0A%20  
20%7B3052891%3A%202019,%0A%20%20771260%3A%202020,%0A%20%202080921  
,%0A%20%201691203%3A%202019,%0A%20%203257780%3A%202019,%0A%20%204  
202019,%0A%20%206250323%3A%202019,%0A%20%204105125%3A%202019,%0A%  
7%3A%202019,%0A%20%204930582%3A%202019,%0A%20%204873064%3A%202019  
0502123%3A%202020,%0A%20%202320995%3A%202019,%0A%20%204967113%3A%  
%20%20750140%3A%202019,%0A%20%204097766%3A%202019,%0A%20%20126353  
9,%0A%20%206049031%3A%202019,%0A%20%202851774%3A%202019,%0A%20%20  
%202019,%0A%20%205743779%3A%202019,%0A%20%206153072%3A%202019,%0A  
119%3A%202019,%0A%20%204884860%3A%202019,%0A%20%206106585%3A%2020  
9%0A%20%20%0Adf.loc%5Bdf%5B'year'%5D%20%3D%3D%202019,%20'total\_am  
count%28%29%20-%20df.loc%5Bdf%5B'year'%5D%20%3D%3D%202020,%20'tot  
%5D.count%28%29%0Adf.loc%5Bdf%5B'year'%5D%20%3D%3D%202019,%20'tot  
%5D.sum%28%29%20-%20df.loc%5Bdf%5B'year'%5D%20%3D%3D%202020,%20't  
'%5D.sum%28%29%0A%0Adf.loc%5B%28df%5B'year'%5D%20%3D%3D%202019%29  
20%20%20%20%20%20%28df%5B'passenger\_count'%5D%20%3E%201%29,%20  
\_count'%5D.count%28%29%20/%20df.loc%5Bdf%5B'year'%5D%20%3D%3D%202  
'payment\_type'%5D.count%28%29%0Adf.loc%5B%28df%5B'year'%5D%20%3D%  
29%20%26%0A%20%20%20%20%20%20%28df%5B'passenger\_count'%5D%20%3  
%20'passenger\_count'%5D.count%28%29%20/%20df.loc%5Bdf%5B'year'%5D  
202020,%20'payment\_type'%5D.count%28%29%0A%0Adf.loc%5B%28df%5B'ye  
3D%3D%202019%29%20%26%0A%20%20%20%20%20%20%28df%5B'payment\_typ  
3D%3D%202%29,%20'payment\_type'%5D.count%28%29%20/%20df.loc%5Bdf%5  
%20%3D%3D%202019,%20'payment\_type'%5D.count%28%29%0Adf.loc%5B%28d  
%5D%20%3D%3D%202020%29%20%26%0A%20%20%20%20%20%20%28df%5B'paym  
%5D%20%3D%3D%202%29,%20'payment\_type'%5D.count%28%29%20/%20df.loc  
year'%5D%20%3D%3D%202020,%20'payment\_type'%5D.count%28%29%0A&d=20  
lang=py&v=1

### 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` between 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.

#### Data frames and dtype

In Chapter 1, we saw that every series has a `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 memory.

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` value 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`.

For example:

```
df_2019_jul = pd.read_csv('../data/nyc_taxi_2019-07.csv',
                          usecols=['passenger_count',
                                    'total_amount', 'payment_type'],
                          dtype={'passenger_count':np.int8,
                                  'total_amount':np.float32,
                                  'payment_type':np.int8})
```

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 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 to a not-quite-correct value. Why not just let Pandas guess, as we have done so far, and then change it afterward? Because in a large data set, 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 choosing a dtype for each chunk. If all of the chunks have the same dtype, then the entire column matches. If not, then the column is set to a dtype of `object`, meaning a collection of Python objects.



#### Note

The chunking I'm describing here is done automatically, as Pandas reads data from the file. Separate functionality allows us to read files in chunks; we'll discuss that in Chapter 11.

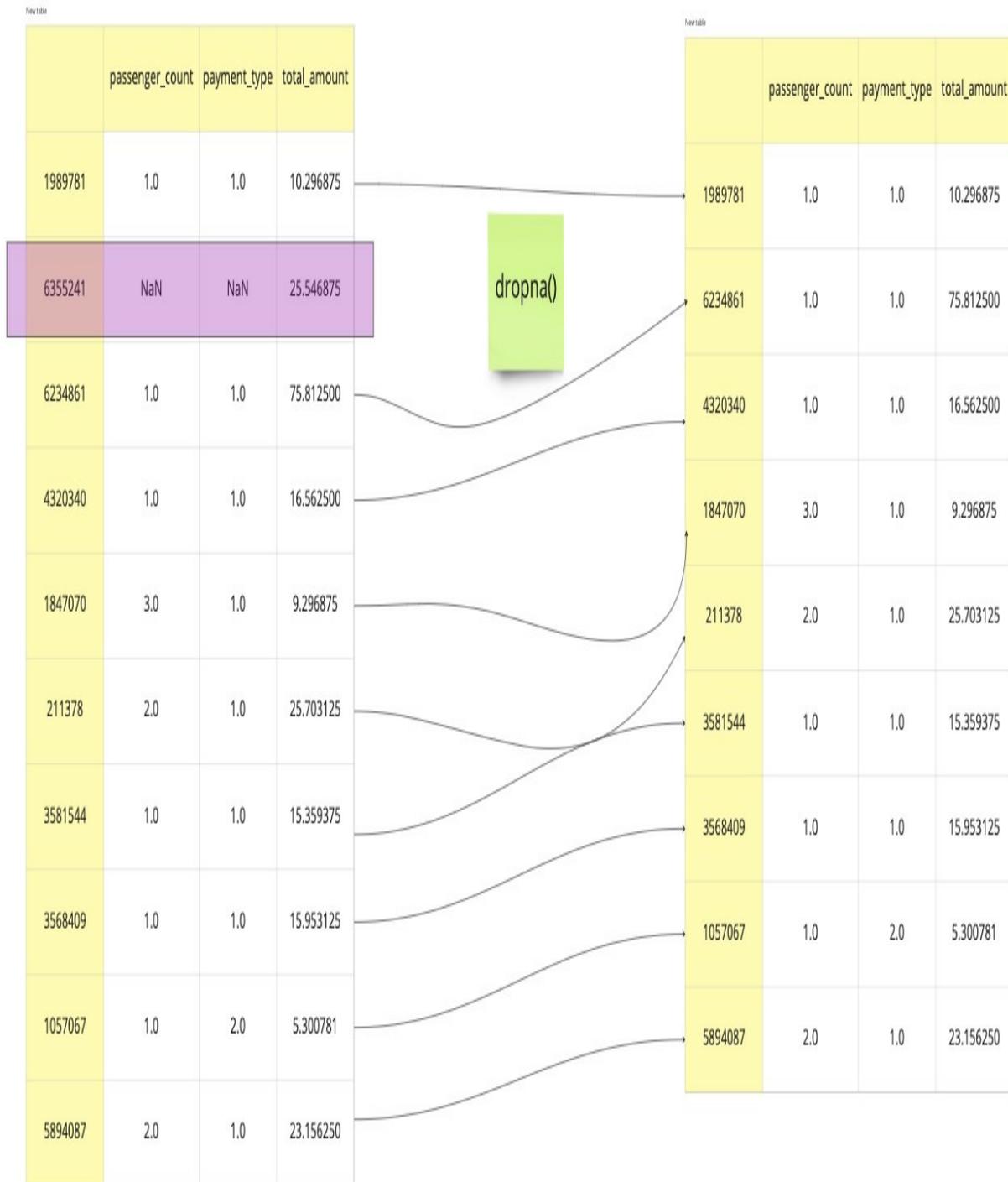
Why would `passenger_count` and `payment_type` contain NaN values? Perhaps because both of them are manually set by the driver. However, it doesn't happen very often: Out of 6.4 million taxi rides in our data set, only 65,441 had NaN values, which works out to about 1 percent. It doesn't seem unreasonable for drivers to neglect to indicate the number of passengers in one out of every 100 rides.

Regardless, to change those two columns' dtype to be `int8`, we need to remove the NaN values. We can do that with `df.dropna()`, which returns a new data frame identical to `df` but without rows containing NaN. We can assign the result of `df.dropna()` back to `df`:

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

Here's a depiction of that:

**Figure 3.9. Removing rows containing NaN with `dropna`**



miro

Even though `df.dropna()` returns a new data frame, it's possible that its data is shared with other data frames, for the sake of efficiency. Modifying our

data frame might thus result in a `SettingWithCopyWarning`. To avoid that, we can use the `copy` method on our data frame, to ensure that there isn't any shared data behind the scenes:

```
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:

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

### 3.4.2 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}) #1
```

```
df.count() #2
```

```
df = df.dropna().copy() #3
```

```
df['passenger_count'] = df['passenger_count'].astype(np.int8) #4
df['payment_type'] = df['payment_type'].astype(np.int8)
```

You can explore this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
3D%20DataFrame%28%7B'passenger_count'%3A%20%7B4283684%3A%201.0,%0
3176%3A%202.0,%0A%20%20573177%3A%201.0,%0A%20%20635083%3A%201.0,%
54203%3A%202.0,%0A%20%202948285%3A%201.0,%0A%20%205908133%3A%201.
05202157%3A%203.0,%0A%20%202491647%3A%201.0,%0A%20%204454508%3A%2
A%20'payment_type'%3A%20%7B4283684%3A%201.0,%0A%20%203503176%3A%2
0%20573177%3A%201.0,%0A%20%20635083%3A%201.0,%0A%20%203754203%3A%
20%202948285%3A%201.0,%0A%20%205908133%3A%201.0,%0A%20%205202157%
0A%20%202491647%3A%201.0,%0A%20%204454508%3A%201.0%7D,%0A%20'tota
3A%20%7B4283684%3A%2012.9609375,%0A%20%203503176%3A%208.796875,%0
177%3A%2012.9609375,%0A%20%20635083%3A%2015.9609375,%0A%20%203754
```

```
10.296875,%0A%20%202948285%3A%2014.0,%0A%20%205908133%3A%2012.359
%205202157%3A%2019.296875,%0A%20%202491647%3A%2048.375,%0A%20%204
%2017.25%7D%7D%29%0A%20%20%0Adf.count%28%29%20%0Adf%20%3D%20df.dr
.copy%28%29%20%0A%0Adf%5B'passenger_count'%5D%20%3D%20df%5B'passe
'%5D.astype%28np.int8%29%20%0Adf%5B'payment_type'%5D%20%3D%20df%5
_type'%5D.astype%28np.int8%29%0A&d=2022-11-15&lang=py&v=v1
```

### 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 Chapter 11, 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 assign.

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 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 certain parameters to `read_csv` are used in nearly every project, making it easier to remember them.

Let's go through each of the keyword arguments that I passed 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`.

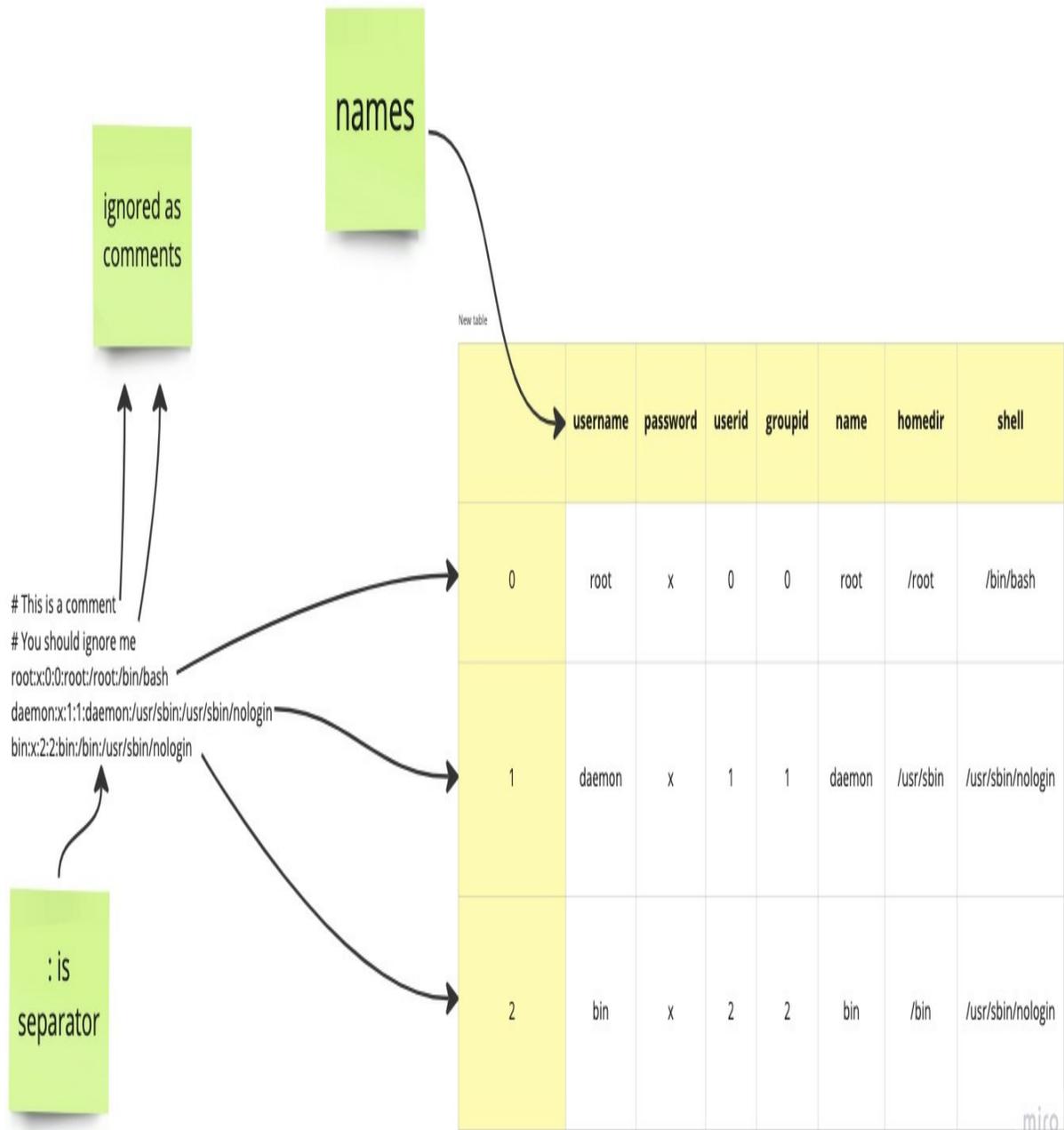
Next, we'll deal with the fact that this `passwd` file contains comments. Comments all start with `#` characters, and extend to the end of the line. Not many companies put comments into their `passwd` files, but given that some do, we should probably handle them. And `read_csv` does this very elegantly,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

**Figure 3.10. Turning the passwd file into a data frame**



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 <https://RegexpCrashCourse.com/>, if you're interested.

Normally, Pandas parses CSV files using a library written in C. If your field separator uses regular expressions, then it needs to use a parser written in Python, which executes more slowly and uses more memory. Consider whether you really need this functionality, and thus the performance hit that the Python-based parser creates.

### 3.5.2 Solution

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

You can explore an abridged version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
3D%20DataFrame%28%7B'username'%3A%20%7B0%3A%20'root',%0A%20%201%3
n',%0A%20%202%3A%20'bin',%0A%20%203%3A%20'sys',%0A%20%204%3A%20's
0%205%3A%20'games',%0A%20%206%3A%20'man',%0A%20%207%3A%20'lp',%0A
%20'mail',%0A%20%209%3A%20'news',%0A%20%2010%3A%20'uucp',%0A%20%2
proxy',%0A%20%2012%3A%20'www-data',%0A%20%2013%3A%20'backup',%0A%
20'list',%0A%20%2015%3A%20'irc',%0A%20%2016%3A%20'gnats',%0A%20%2
nobody',%0A%20%2018%3A%20'syslog',%0A%20%2019%3A%20'messagebus'%7
password'%3A%20%7B0%3A%20'x',%0A%20%201%3A%20'x',%0A%20%202%3A%20
```

0%203%3A%20'x',%0A%20%204%3A%20'x',%0A%20%205%3A%20'x',%0A%20%206  
,%0A%20%207%3A%20'x',%0A%20%208%3A%20'x',%0A%20%209%3A%20'x',%0A%  
A%20'x',%0A%20%2011%3A%20'x',%0A%20%2012%3A%20'x',%0A%20%2013%3A%  
20%2014%3A%20'x',%0A%20%2015%3A%20'x',%0A%20%2016%3A%20'x',%0A%20  
'x',%0A%20%2018%3A%20'x',%0A%20%2019%3A%20'x'%7D,%0A%20'userid'%3  
%200,%0A%20%201%3A%201,%0A%20%202%3A%202,%0A%20%203%3A%203,%0A%20  
,%0A%20%205%3A%205,%0A%20%206%3A%206,%0A%20%207%3A%207,%0A%20%208  
%20%209%3A%209,%0A%20%2010%3A%2010,%0A%20%2011%3A%2013,%0A%20%201  
%0A%20%2013%3A%2034,%0A%20%2014%3A%2038,%0A%20%2015%3A%2039,%0A%2  
041,%0A%20%2017%3A%2065534,%0A%20%2018%3A%20101,%0A%20%2019%3A%20  
20'groupid'%3A%20%7B0%3A%200,%0A%20%201%3A%201,%0A%20%202%3A%202,'  
%3A%203,%0A%20%204%3A%2065534,%0A%20%205%3A%2060,%0A%20%206%3A%20  
207%3A%207,%0A%20%208%3A%208,%0A%20%209%3A%209,%0A%20%2010%3A%201  
011%3A%2013,%0A%20%2012%3A%2033,%0A%20%2013%3A%2034,%0A%20%2014%3  
%20%2015%3A%2039,%0A%20%2016%3A%2041,%0A%20%2017%3A%2065534,%0A%2  
20104,%0A%20%2019%3A%20106%7D,%0A%20'name'%3A%20%7B0%3A%20'root',  
'%3A%20'daemon',%0A%20%202%3A%20'bin',%0A%20%203%3A%20'sys',%0A%20  
'sync',%0A%20%205%3A%20'games',%0A%20%206%3A%20'man',%0A%20%207%3  
%0A%20%208%3A%20'mail',%0A%20%209%3A%20'news',%0A%20%2010%3A%20'u  
%2011%3A%20'proxy',%0A%20%2012%3A%20'www-data',%0A%20%2013%3A%20'  
A%20%2014%3A%20'Mailing%20List%20Manager',%0A%20%2015%3A%20'ircd'  
16%3A%20'Gnats%20Bug-Reporting%20System%20%28admin%29',%0A%20%201  
nobody',%0A%20%2018%3A%20np.nan,%0A%20%2019%3A%20np.nan%7D,%0A%20  
'%3A%20%7B0%3A%20'/root',%0A%20%201%3A%20'/usr/sbin',%0A%20%202%3  
,%0A%20%203%3A%20'/dev',%0A%20%204%3A%20'/bin',%0A%20%205%3A%20'/  
,%0A%20%206%3A%20'/var/cache/man',%0A%20%207%3A%20'/var/spool/lpd  
08%3A%20'/var/mail',%0A%20%209%3A%20'/var/spool/news',%0A%20%2010  
'/var/spool/uucp',%0A%20%2011%3A%20'/bin',%0A%20%2012%3A%20'/var/w  
0%2013%3A%20'/var/backups',%0A%20%2014%3A%20'/var/list',%0A%20%20  
'/var/run/ircd',%0A%20%2016%3A%20'/var/lib/gnats',%0A%20%2017%3A%2  
stent',%0A%20%2018%3A%20'/home/syslog',%0A%20%2019%3A%20'/var/run  
,%0A%20'shell'%3A%20%7B0%3A%20'/bin/bash',%0A%20%201%3A%20'/usr/s  
,%0A%20%202%3A%20'/usr/sbin/nologin',%0A%20%203%3A%20'/usr/sbin/  
%0A%20%204%3A%20'/bin/sync',%0A%20%205%3A%20'/usr/sbin/nologin',%  
3A%20'/usr/sbin/nologin',%0A%20%207%3A%20'/usr/sbin/nologin',%0A%  
20'/usr/sbin/nologin',%0A%20%209%3A%20'/usr/sbin/nologin',%0A%20%  
'/usr/sbin/nologin',%0A%20%2011%3A%20'/usr/sbin/nologin',%0A%20%2  
'/usr/sbin/nologin',%0A%20%2013%3A%20'/usr/sbin/nologin',%0A%20%2  
'/usr/sbin/nologin',%0A%20%2015%3A%20'/usr/sbin/nologin',%0A%20%2  
'/usr/sbin/nologin',%0A%20%2017%3A%20'/usr/sbin/nologin',%0A%20%2  
'/bin/false',%0A%20%2019%3A%20'/bin/false'%7D%7D%29%0A%20%20%0Adf  
-21&lang=py&v=v1

### 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`'s first argument, which we normally think of as a filename, can actually contain several different types of values:

- 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.

For example, you could say:

```
import requests
from io import StringIO

r = requests.get('https://data_for_you.com/data.csv') #1
s = StringIO(r.content.decode()) #2
df = pd.read_csv(s) #3
```

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 look different from mine, even if you use the same code.) 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:

<https://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 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
                 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 the 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`.



#### Note

`df.tail(1)` returns the final row of `df`, which contains both the `date` and `value` columns. If I only want `value`, what can I do?

One option is to think of `df.tail(1)` as a one-row data frame. Each column of a data frame is a series, so we can retrieve the value with:

```
df.tail(1)['value']
```

Sure enough, we'll get a one-element series back. But remember that we can retrieve more than one column from a data frame, by passing a list of columns—that is, in double square brackets. What if we use double square brackets, but only list one column? That is:

```
df.tail(1)[['value']]
```

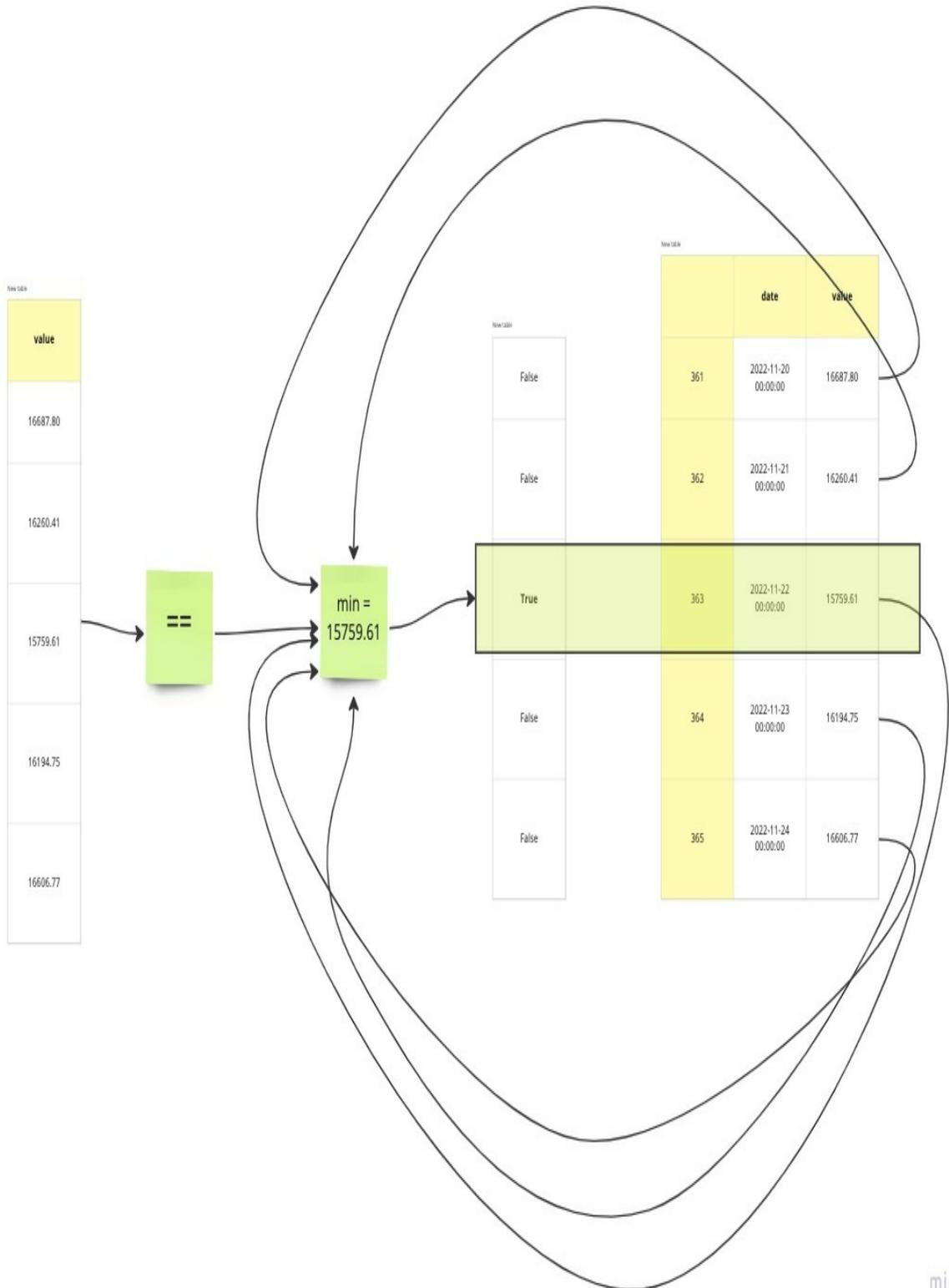
Now the result will be a data frame containing one row (same as `df.tail(1)`) and one column (`value`).

Which syntax you choose depends on what you want to do with the data. In this particular case, though, it doesn't really matter.

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']]
```

**Figure 3.11. Selecting the minimum value from a data frame with a mask index**



## 3.6.2 Solution

```
import pandas as pd
from pandas import Series, DataFrame

df = pd.read_csv('https://api.blockchain.info/charts/market-price
                 header=None,
                 names=['date', 'value']) #1

df.tail(1)[['value']] #2
df.loc[df['value'] == df['value'].min(), ['date', 'value']] #3
df.loc[df['value'] == df['value'].max(), ['date', 'value']] #4
```

You can explore an abridged version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
pandas%20as%20pd%0Afrom%20pandas%20import%20Series,%20DataFrame%0
%3D%20DataFrame%28%7B'date'%3A%20%7B356%3A%20'2022-11-15%2000%3A0
0A%20%20357%3A%20'2022-11-16%2000%3A00%3A00',%0A%20%20358%3A%20'2
2000%3A00%3A00',%0A%20%20359%3A%20'2022-11-18%2000%3A00%3A00',%0A
3A%20'2022-11-19%2000%3A00%3A00',%0A%20%20361%3A%20'2022-11-20%20
A00',%0A%20%20362%3A%20'2022-11-21%2000%3A00%3A00',%0A%20%20363%3
-11-22%2000%3A00%3A00',%0A%20%20364%3A%20'2022-11-23%2000%3A00%3A
%20365%3A%20'2022-11-24%2000%3A00%3A00'%7D,%0A%20'value'%3A%20%7B
16587.96,%0A%20%20357%3A%2016873.56,%0A%20%20358%3A%2016662.24,%0
%3A%2016682.44,%0A%20%20360%3A%2016683.22,%0A%20%20361%3A%2016687
20362%3A%2016260.41,%0A%20%20363%3A%2015759.61,%0A%20%20364%3A%20
%0A%20%20365%3A%2016606.77%7D%7D,%0A%20%20columns%3D%5B'date',%20
D%29%0A%20%20%0Adf.tail%281%29%5B%5B'value'%5D%5D%0Adf.loc%5Bdf%5
5D%20%3D%3D%20df%5B'value'%5D.min%28%29,%20%5B'date',%20'value'%5
Adf.loc%5Bdf%5B'value'%5D%20%3D%3D%20df%5B'value'%5D.max%28%29,%2
,%20'value'%5D%5D%20%0A%0A%0A%0A&d=2022-11-24&lang=py&v=v1
```

## 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. Without assigning the downloaded data to an interim 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 (<https://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 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 as 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 the 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',
```

The result, 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. (So far as I know, there isn't a good way to avoid such problems.)

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)
```

### 3.7.2 Solution

```

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

df['population'].describe()[['mean', '50%']] #1
df.loc[50:, 'population'].describe()[['mean', '50%']]#2
df.loc[df['latitude'] == df['latitude'].max(), ['city', 'state',
df['state'].value_counts().head(1) #4
df['state'].value_counts().tail(5) #5

```

You can explore an abridged version of this in the Pandas Tutor at:

3A8,%228%22%3A9,%229%22%3A10,%2210%22%3A11,%2211%22%3A12,%2212%22%3A13,%2214%22%3A15,%2215%22%3A16,%2216%22%3A17,%2217%22%3A18,%2219%22%3A20%7D,%22state%22%3A%7B%220%22%3A%22New%20York%22%3A%22California%22,%222%22%3A%22Illinois%22,%223%22%3A%22Texas%22%3A%22Pennsylvania%22,%225%22%3A%22Arizona%22,%226%22%3A%22Texas%22%3A%22California%22,%228%22%3A%22Texas%22,%229%22%3A%22California%22%3A%22Texas%22,%2211%22%3A%22Indiana%22,%2212%22%3A%22Florida%22%3A%22California%22,%2214%22%3A%22Ohio%22,%2215%22%3A%22North%22,%2216%22%3A%22Texas%22,%2217%22%3A%22Michigan%22,%2218%22%3A%2219%22%3A%22Tennessee%22%7D%7D'%29%0A%0Adf%5B'population'%5D.describe%28%29%5B%5B'mean',%20'50%25'%5D%5D%20%0Adf.loc%5B50%3A,%20'populati describe%28%29%5B%5B'mean',%20'50%25'%5D%5D%0Adf.loc%5Bdf%5B'lati 20%3D%3D%20df%5B'latitude'%5D.max%28%29,%20%5B'city',%20'state',% %5D%5D%0A&d=2022-11-24&lang=py&v=v1

### 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 city or cities 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

My parents introduced me to the wonders of the public library at a young age. It held an immense number of books, on every subject you could possibly imagine.

But wait: With so many books, on so many subjects, by so many authors, how can you possibly find what you want? Or even know what's available?

The answer was an index. In those days, libraries typically had three different indexes, known as the "card catalog," containing hundreds of drawers full of index cards. These cards allowed you to find books (a) by author, (b) by title, or (c) by subject. Beyond that, the books were shelved according to their subjects, either according to the Dewey Decimal system or by the Library of Congress system.

If you were familiar with all of these systems, you could easily find what you were looking for: A particular book that had been mentioned in the newspaper, other books written by your favorite author, or books on a particular subject you were researching for school.

Nowadays, of course, the indexes are all computerized, allowing you to find books more flexibly and easily than we ever imagined in the days of the card catalog.

Could you have a library without an index? Yes, but it would be much less useful. It would be harder to find what you want, and every search would take significantly longer. How to best catalog information so that it's easily findable is so important that there's an entire branch of academia, "library science," dedicated to it.

Just as an index can help us to find books in a library, it can help us to find data in Pandas. We've already seen that a series has one index (for its elements), and a data frame has two (one for the rows, and a second for the columns). We've seen how `.loc`, along with row selectors and column

selectors, can be quite powerful.

But indexes in Pandas far more flexible than we've seen so far: We can make an existing column into an index, or turn an index back into a regular column. We can combine multiple columns into a hierarchical "multi-index," and then perform searches on specific parts of that hierarchy. Indeed, knowing how to create, query, and manipulate multi-indexed data frames is key to fluent work with Pandas. We can also create "pivot tables," in which the rows and columns reflect not our original data, but rather aggregate summaries of that data.

In this chapter, we'll practice using all of these techniques, to better understand how to create, modify, and manipulate a variety of types of indexes. After working through these exercises, you'll better know how to use Pandas indexes to retrieve data more flexibly and easily.

## 4.1 Useful references

Table 4.1. What you need to know

Concept	What is it?	Example
<code>pd.set_index</code>	returns a new data frame with a new index	<code>df = df.set_index('name')</code>
<code>pd.reset_index</code>	returns a new data frame with a default (numeric, positional) index	<code>df = df.reset_index()</code>

df.loc	Retrieve selected rows and columns	df.loc[:, 'passenger_count'] = df['passenger_count']
s.value_counts	returns a sorted (descending frequency) series counting how many times each value appears in s	s.value_counts()
s.isin	returns a boolean series indicating whether a value in s is an element of the argument	s.isin(['A', 'B', 'C'])
df.pivot	creates a pivot table based on a data frame, without aggregation	df.pivot(index='month', columns='year', values='A')

<code>df.pivot_table</code>	creates a pivot table based on a data frame, <b>with</b> aggregation, if needed	<code>df.pivot_table(index='month', columns='year', values='A')</code>
<code>s.is_monotonic_increasing</code>	Contains True if values in the series are sorted in increasing order.	<code>s.is_monotonic_increasing</code>
<code>df.xs</code>	Returns a cross-section from a data frame	<code>df.xs(2016, level='Year')</code>
<code>IndexSlice</code>	Produce an object for easier querying of data frames using <code>xs</code>	<code>IndexSlice[:, 2016]</code>

## 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` attribute. 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 three makes were most frequently ticketed 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 series as a mask index. Often, 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, reducing our code's complexity and length. 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')
```

**Figure 4.1. Graphical depiction of turning "Issue Date" from a column into the index**

New table

	Plate ID	Registration State	Issue Date	Vehicle Make	Street Name	Date First Observed	Vehicle Color
725518	JFG4137	NY	07/16/2019 12:00:00 AM	DODGE	PACIFIC STREET	0	WHT
247136	DPH1199	NY	07/01/2019 12:00:00 AM	NISSA	160th St	0	BK
1628916	8D45B	NY	08/06/2019 12:00:00 AM	FORD	NB BAYCHESTER AVE @	0	YW
6757299	67974JV	NY	12/11/2019 12:00:00 AM	ISUZU	95th St	0	WHITE
4482906	JBN3055	NY	10/13/2019 12:00:00 AM	DODGE	SWINTON AVE	0	GRY
12331922	CKS1861	GA	06/17/2020 12:00:00 AM	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
1723597	58388MG	NY	08/08/2019 12:00:00 AM	CHEVR	E 38th St	20190808	WH
2474539	AP628T	NJ	08/26/2019 12:00:00 AM	INTER	1st Ave	0	WHITE

set\_index('Issue date')

New table

	Plate ID	Registration State	Vehicle Make	Street Name	Date First Observed	Vehicle Color
Issue date						
07/16/2019 12:00:00 AM	JFG4137	NY	DODGE	PACIFIC STREET	0	WHT
07/01/2019 12:00:00 AM	DPH1199	NY	NISSA	160th St	0	BK
08/06/2019 12:00:00 AM	8D45B	NY	FORD	NB BAYCHESTER AVE @	0	YW
12/11/2019 12:00:00 AM	67974JV	NY	ISUZU	95th St	0	WHITE
10/13/2019 12:00:00 AM	JBN3055	NY	DODGE	SWINTON AVE	0	GRY
06/17/2020 12:00:00 AM	CKS1861	GA	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
08/08/2019 12:00:00 AM	58388MG	NY	CHEVR	E 38th St	20190808	WH
08/26/2019 12:00:00 AM	AP628T	NJ	INTER	1st Ave	0	WHITE

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 data frame. The core Pandas developers have warned that this is a bad idea, because it makes incorrect assumptions about memory and performance. There is, they say, no benefit to using `inplace=True`. As a result, the `inplace` parameter 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 straightforward 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 passing a row selector, then passing a column selector. 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 returns a series in which the index contains the different vehicle makes, and the values are the counts, sorted from highest to lowest. We can limit 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().h
```

Once we have this information, 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().he
```

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')
```

**Figure 4.2. Graphical depiction of returning "Issue Date" from the index to a column, and making "Vehicle Color" the new index**

New table

	Plate ID	Registration State	Vehicle Make	Street Name	Date First Observed	Vehicle Color
Issue date						
07/16/2019 12:00:00 AM	JFG4137	NY	DODGE	PACIFIC STREET	0	WHT
07/01/2019 12:00:00 AM	DPH1199	NY	NISSA	160th St	0	BK
08/06/2019 12:00:00 AM	8D45B	NY	FORD	NB BAYCHESTER AVE @	0	YW
12/11/2019 12:00:00 AM	67974JV	NY	ISUZU	95th St	0	WHITE
10/13/2019 12:00:00 AM	JBN3055	NY	DODGE	SWINTON AVE	0	GRY
06/17/2020 12:00:00 AM	CKS1861	GA	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
08/08/2019 12:00:00 AM	58388MG	NY	CHEVR	E 38th St	20190808	WH
08/26/2019 12:00:00 AM	AP628T	NJ	INTER	1st Ave	0	WHITE

reset\_index()

New table

	Plate ID	Registration State	Issue Date	Vehicle Make	Street Name	Date First Observed	Vehicle Color
725518	JFG4137	NY	07/16/2019 12:00:00 AM	DODGE	PACIFIC STREET	0	WHT
247136	DPH1199	NY	07/01/2019 12:00:00 AM	NISSA	160th St	0	BK
1628916	8D45B	NY	08/06/2019 12:00:00 AM	FORD	NB BAYCHESTER AVE @	0	YW
6757299	67974JV	NY	12/11/2019 12:00:00 AM	ISUZU	95th St	0	WHITE
4482906	JBN3055	NY	10/13/2019 12:00:00 AM	DODGE	SWINTON AVE	0	GRY
12331922	CKS1861	GA	06/17/2020 12:00:00 AM	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
1723597	58388MG	NY	08/08/2019 12:00:00 AM	CHEVR	E 38th St	20190808	WH
2474539	AP628T	NJ	08/26/2019 12:00:00 AM	INTER	1st Ave	0	WHITE

set\_index('Vehicle Color')

New table

	Plate ID	Registration State	Vehicle Make		Street Name	Date First Observed
Vehicle Color				Issue date		
WHT	JFG4137	NY	DODGE	07/16/2019 12:00:00 AM	PACIFIC STREET	0
BK	DPH1199	NY	NISSA	07/01/2019 12:00:00 AM	160th St	0
YW	8D45B	NY	FORD	08/06/2019 12:00:00 AM	NB BAYCHESTER AVE @	0
WHITE	67974JV	NY	ISUZU	12/11/2019 12:00:00 AM	95th St	0
GRY	JBN3055	NY	DODGE	10/13/2019 12:00:00 AM	SWINTON AVE	0
GRAY	CKS1861	GA	Jeep	06/17/2020 12:00:00 AM	NB OCEAN PKWY @ AVE	0
WH	58388MG	NY	CHEVR	08/08/2019 12:00:00 AM	E 38th St	20190808
WHITE	AP628T	NJ	INTER	08/26/2019 12:00:00 AM	1st Ave	0

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
    'Issue Date', 'Vehicle Make', 'Street Name', 'Vehicle Col
df = df.set_index('Issue Date') #1
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make'].value_counts().h
df.loc['06/01/2020 12:00:00 AM', 'Street Name'].value_counts().he
df = df.reset_index().set_index('Vehicle Color') #4
df.loc[['BLUE', 'RED'], 'Vehicle Make'].value_counts().head(1) #5
```

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
numpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%
%20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%2
ingIO%28''%0AIssue%20Date,Vehicle%20Make%5Cn01/02/2020%2012%3A00
```

```
, MAZDA%5Cn01/02/2020%2012%3A00%3A00%20AM, TOYOT%5Cn01/02/2020%2012%20AM, NISSA%5Cn01/02/2020%2012%3A00%3A00%20AM, FORD%5Cn01/02/2020%2012%3A00%20AM, HIN%5Cn01/02/2020%2012%3A00%3A00%20AM, FORD%5Cn01/02/2020%3A00%20AM, FORD%5Cn01/02/2020%2012%3A00%3A00%20AM, KENW0%5Cn01/02/3A00%3A00%20AM, CHRYS%5Cn01/02/2020%2012%3A00%3A00%20AM, TOYOT%5Cn02012%3A00%3A00%20AM, TOYOT%5Cn01/02/2020%2012%3A00%3A00%20AM, GMC%5020%2012%3A00%3A00%20AM, VPG%5Cn01/02/2020%2012%3A00%3A00%20AM, ME/2/2020%2012%3A00%3A00%20AM, INFIN%5Cn01/02/2020%2012%3A00%3A00%20An01/02/2020%2012%3A00%3A00%20AM, KIA%5Cn01/02/2020%2012%3A00%3A00%N%5Cn01/02/2020%2012%3A00%3A00%20AM, NISSA%5Cn01/02/2020%2012%3A00M, HONDA%5Cn01/02/2020%2012%3A00%3A00%20AM, TOYOT%5Cn01/02/2020%20100%20AM, NISSA%5Cn01/02/2020%2012%3A00%3A00%20AM, BUICK%5Cn01/02/2000%3A00%20AM, NISSA%5Cn01/02/2020%2012%3A00%3A00%20AM, NISSA%5Cn01/12%3A00%3A00%20AM, DODGE%5Cn01/02/2020%2012%3A00%3A00%20AM, NISSA%520%2012%3A00%3A00%20AM, HONDA%5Cn01/02/2020%2012%3A00%3A00%20AM, HO/02/2020%2012%3A00%3A00%20AM, MINI%5Cn%0A' ' '%29%0A%0Adf%20%3D%20pd%28data,%20index_col%3D'Issue%20Date'%29%0Adf.loc%5B'01/02/2020%23A00%20AM',%20'Vehicle%20Make'%5D.value_counts%28%29.head%283%29%2022-12-25&lang=py&v=v1
```

### 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 three car makes were most often ticketed from 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?

#### 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 often serves us quite well. But there are many cases in which it's not quite enough. That's because 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 figur
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'.s
```

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'])
```

**Figure 4.3. Graphical depiction of creating a multi-index from the "year" and "month" columns**

New table

	A	B	C	year	month
0	44	47	64	2018	Jan
1	67	67	9	2018	Feb
2	83	21	36	2018	Mar
3	87	70	88	2018	Apr
4	88	12	58	2018	May
5	65	39	87	2018	Jun
6	46	88	81	2018	Jul
7	37	25	77	2018	Aug
8	72	9	20	2018	Sep

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

New table

		A	B	C
year	month			
2018	Jan	44	47	64
2018	Feb	67	67	9
2018	Mar	83	21	36
2018	Apr	87	70	88
2018	May	88	12	58
2018	Jun	65	39	87
2018	Jul	46	88	81
2018	Aug	37	25	77
2018	Sep	72	9	20

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, and city. This allows you to retrieve all rows for a given region, or a given country, or a given city, relatively easily. Usually (but not always), a multi-index will reflect a hierarchy.

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, 2018?

```
df.loc[(2018, '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 2018 and Jun—the outermost level and the inner level—so I use the tuple `(2018, 'Jun')`. I can, of course, retrieve the sales data just for products A and C here, too:

```
df.loc[(2018, 'Jun'), ['A', 'C']]
```

**Figure 4.4. Graphical depiction of retrieving rows from June 2018, columns A and C, from a multi-index**

column selector:  
['A', 'C']

New table

		A	B	C
year	month			
2018	Jan	44	47	64
2018	Feb	67	67	9
2018	Mar	83	21	36
2018	Apr	87	70	88
2018	May	88	12	58
2018	Jun	65	39	87
2018	Jul	46	88	81
2018	Aug	37	25	77
2018	Sep	72	9	20

row selector:  
(2018, 'Jun')

results: [65, 87]

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 2018 and 2020. I can say:

```
df.loc[[2018, 2020]]
```

And if I want to see all data for 2018 and 2020, but only products B and C?

```
df.loc[[2018, 2020], ['B', 'C']]
```

What if I want to get all of the data from June in both 2018 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([(2018, '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. Typically, you won't want all of the columns, because the analysis you'll want to do will involve a subset of the full data frame.

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':'C')
```

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'] #1
```

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 extremely versatile, allowing us to retrieve from a multi-index in a variety of ways.

## 4.3 Exercise 22: State SAT scores

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. (See the sidebar above, "Working with multi-indexes," for a fuller description.)

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. 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 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 exercise, 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`.

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. 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 know how many people took the SAT in 2005. This will mean finding all rows from 2005, and the column `Total.Test-takers`, which tells us how many people took the test in each year, for each state, and summing those values:

```
df.loc[2005, 'Total.Test-takers'].sum()
```

Next, 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']), #1  
       'Total.Math'].mean() #2
```

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.

**Figure 4.5. Graphical breakdown of `.loc` with a multi-index**

row selector; tuple indicates 2 levels

of a multi-index

column selector

Year must == 2010

State.Code can be

any value in the list

```
df.loc[  
    (2010,  
     ['NY', 'NJ', 'MA', 'IL']),  
    'Total.Math']
```

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']], #1
        'Total.Math').mean() #2
```

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']) #1
df.loc[2005, 'Total.Test-takers'].sum() #2

df.loc[(2010, ['NY', 'NJ', 'MA', 'IL']),
        'Total.Math'].mean() #3

df.loc([(2012,2013,2014,2015),
        ['AZ', 'CA', 'TX']],
        'Total.Math').mean()#4
```

You can explore a version of this in the Pandas Tutor at:



## Sorting by index

When we talk about sorting 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 also lets us sort our data frames based on the index. We can do that with the `sort_index` method, which (like so many others) returns a new data frame with the same content as the original, with rows sorted based on index's 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 esthetic benefits, sorting a data frame by index can make certain tasks easier, or even possible. For example, if you try to retrieve a slice, such as `df.loc['a':'c']`, Pandas will insist that the index be sorted, to avoid problems if a and c are interspersed.

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 perfor
```

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
```

Saying that the index is "monotonically increasing," by the way, simply means that it only goes up. Similarly, if the values only go down, then we would say that it's "monotonically decreasing," which we can check with

`is_monotonic_decreasing`. Note that these are **not** methods, but rather boolean attributes. They exist on all series objects, not just on indexes. Some older documentation and blogs mentions the method `is_lexsorted`, which has been deprecated in recent versions of Pandas.

## 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 2016?

### 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'], #1
                 usecols=['Age', 'Height', 'Team',
                          'Year', 'Season', 'City',
                          'Sport', 'Event', 'Medal']) #2

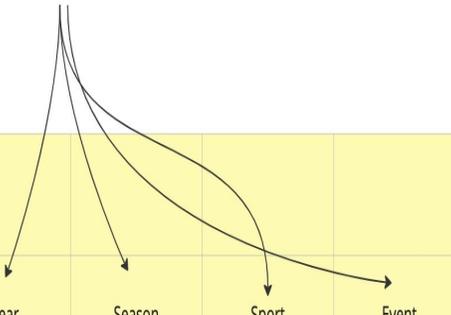
df = df.sort_index() #3
```

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.

**Figure 4.6. Graphical depiction of our data frame with four columns in its multi-index**

## Multi-index

New table



				Age	Height	Team	City	Medal
Year	Season	Sport	Event					
1996	Summer	Athletics	Athletics Men's 10,000 metres	27.0	178.0	United States	Atlanta	NaN
1992	Winter	Biathlon	Biathlon Women's 15 kilometres	22.0	NaN	China	Albertville	NaN
2012	Summer	Fencing	Fencing Men's Foil, Team	29.0	180.0	China	London	NaN
1988	Winter	Cross Country Skiing	Cross Country Skiing Men's 50 kilometres	24.0	174.0	Sweden	Calgary	NaN
1900	Summer	Rowing	Rowing Men's Coxed Eights	21.0	NaN	Germania Ruder Club, Hamburg	Paris	NaN
2006	Winter	Biathlon	Biathlon Men's 4 x 7.5 kilometres Relay	28.0	180.0	Czech Republic	Torino	NaN
2004	Summer	Cycling	Cycling Men's Mountainbike, Cross-Country	22.0	178.0	Spain	Athina	NaN
1912	Summer	Gymnastics	Gymnastics Men's Team All-Around	20.0	NaN	Germany	Stockholm	NaN
1952	Summer	Rowing	Rowing Men's Coxless Fours	26.0	186.0	Norway	Helsinki	NaN
1994	Winter	Ski Jumping	Ski Jumping Men's Large Hill, Team	23.0	175.0	Italy	Lillehammer	NaN

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 held 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

**Figure 4.7. Graphical depiction of applying our multi-index row selector**

Column selector: Age

New table

				Age	Height	Team	City	Medal	
Year	Season	Sport	Event						
1996	Summer	Athletics	Athletics Men's 10,000 metres	27.0	178.0	United States	Atlanta	NaN	
1992	Winter	Biathlon	Biathlon Women's 15 kilometres	22.0	NaN	China	Albertville	NaN	
2012	Summer	Fencing	Fencing Men's Foil, Team	29.0	180.0	China	London	NaN	
1988	Winter	Cross Country Skiing	Cross Country Skiing Men's 50 kilometres	24.0	174.0	Sweden	Calgary	NaN	
1900	Summer	Rowing	Rowing Men's Coxed Eights	21.0	NaN	Germania Ruder Club, Hamburg	Paris	NaN	
2006	Winter	Biathlon	Biathlon Men's 4 x 7.5 kilometres Relay	28.0	180.0	Czech Republic	Torino	NaN	
2004	Summer	Cycling	Cycling Men's Mountainbike, Cross-Country	22.0	178.0	Spain	Athina	NaN	
1912	Summer	Gymnastics	Gymnastics Men's Team All-Around	20.0	NaN	Germany	Stockholm	NaN	
1952	Summer	Rowing	Rowing Men's Coxless Fours	26.0	186.0	Norway	Helsinki	NaN	
1994	Winter	Ski Jumping	Ski Jumping Men's Large Hill, Team	23.0	175.0	Italy	Lillehammer	NaN	

Row selector, Summers 1936 - 2000

Finally, we'll want to find the mean of those ages. We can express this as:

```
df.loc[(slice(1936,2000), 'Summer'), #1
        'Age' #2
       ].mean() #3
```

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'), #1
        'Team' #2
       ].value_counts() #3
```

Here are the first five results:

United States	155
France	151
Great Britain	133
South Korea	102
China	98

Because `value_counts` sorts its values in descending order, we see that the United States has had the greatest number of archery participants, 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), #1
        'Summer', #2
        slice(None), #3
        "Table Tennis Women's Team"), #4
        'Height' #5
        ].mean() #6
```

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), #1
        'Summer', #2
        slice(None), #3
        ["Table Tennis Men's Team",
        "Table Tennis Women's Team"]), #4
        'Height' #5
        ].mean() #6
```

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 2016. This can most easily be handled with `range(1980, 2016)`.
- 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 specify `Tennis` as the sport
- I'll allow any events, so I don't need to pass a fourth element in the tuple

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(1980,2016), #1
        'Summer', #2
        'Tennis'), #3
        'Height' #4
       ].max() #5
```

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']) #1

df = df.sort_index() #2
df.loc[(slice(1936,2000), 'Summer'), 'Age'].mean() #3
df.loc[(slice(None), 'Summer', 'Archery'),
        'Team'].value_counts() #4
df.loc[(slice(None), 'Summer', slice(None),
```

```

    "Table Tennis Women's Team"),
    'Height'].mean() #5
df.loc[(slice(None),
         'Summer', slice(None),
         ["Table Tennis Men's Team",
          "Table Tennis Women's Team"]),
        'Height'].mean() #6
df.loc[(slice(1980, 2016),
         'Summer',
         'Tennis'), 'Height'].max() #7

```

You can explore a version of this in the Pandas Tutor at:

```
%20Architecture,%20Architectural%20Designs%22,%5Cn1936,Summer,Art  
tions,%22Art%20Competitions%20Mixed%20Architecture,%20Architectur  
s%22,48.0%5Cn1936,Summer,Art%20Competitions,%22Art%20Competitions%  
0Architecture,%20Architectural%20Designs%22,51.0%5Cn1936,Summer,A  
tions,%22Art%20Competitions%20Mixed%20Architecture,%20Architect  
igns%22,%5Cn1936,Summer,Art%20Competitions,%22Art%20Competitions%  
Architecture,%20Architectural%20Designs%22,%5Cn1936,Summer,Art%20  
ns,%22Art%20Competitions%20Mixed%20Architecture,%20Architectural%  
22,36.0%5Cn1936,Summer,Art%20Competitions,%22Art%20Competitions%2  
Architecture,%20Architectural%20Designs%22,48.0%5Cn1936,Summer,Ar  
tions,%22Art%20Competitions%20Mixed%20Architecture,%20Architectur  
ns%22,49.0%5Cn1936,Summer,Art%20Competitions,%22Art%20Competitio  
20Architecture,%20Architectural%20Designs%22,37.0%5Cn%0A''%29%0A  
D%20pd.read_csv%28data,%0A%20%20%20%20%20%20%20%20%20%20%20%20  
ndex_col%3D%5B'Year',%20'Season',%0A%20%20%20%20%20%20%20%20%20%2  
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%  
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%  
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%  
'Season',%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%2  
0%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%2  
0%20%20%20%20%20'Sport',%20'Event'%5D,%29%20%0Adf%20%3D%20df.sort_ind  
0%0Adf.loc%5B%28slice%281936,2000%29,%20'Summer'%29,%20'Age'%5D.m  
20%20%0A&d=2022-12-25&lang=py&v=v1
```

## 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), #1
        'Summer', #2
        slice(None), #3
        "Table Tennis Women's Team"), #4
        'Height' #5
        ].mean() #6
```

Using `xs`, we could shorten that query to:

```
df.xs("Table Tennis Women's Team", #1
      level='Event' #2
      ).mean() #3
```

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"), #1
      level=['Season', 'Event']).mean() #2
```

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[1980:2016, :, 'Swimming':'Table tennis'], :] #1
```

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, 2016, 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.

One final note: A data frame can have a multi-index on its rows, its columns, or both. By default, `xs` assumes that the multi-index is on the rows. If and when you want to use it on multi-index columns, pass `axis='columns'` as a keyword argument.

### 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 2016.
- 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.)

#### 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. For example: Years, country names, colors, and divisions of a company.
- 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, consider this simple data frame:

```
np.random.seed(0)
df = DataFrame(np.random.randint(0, 100, [8, 3]),
               columns=list('ABC'))
df['year'] = [2018] * 4 + [2019] * 4
df['quarter'] = 'Q1 Q2 Q3 Q4'.split() * 2
```

This table shows the sales of each product per year and quarter. 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 quarters (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='quarter', columns='year', values='A')
```

The result, on my computer, is a data frame that looks like this:

year	2018	2019
quarter		
Q1	44	88
Q2	67	65
Q3	83	46
Q4	87	37

**Figure 4.8. Graphical depiction of creating a pivot table with index "quarter", columns "year", and values "A"**

New Table

	A	B	C	year	quarter
0	44	47	64	2018	Q1
1	67	67	9	2018	Q2
2	83	21	36	2018	Q3
3	87	70	88	2018	Q4
4	88	12	58	2019	Q1
5	65	39	87	2019	Q2
6	46	88	81	2019	Q3
7	37	25	77	2019	Q4

```
pivot_table(  
  index='quarter',  
  columns='year',  
  values='A')
```

New Table

year	2018	2019
quarter		
Q1	44	88
Q2	67	65
Q3	83	46
Q4	87	37

The quarters are sorted in alphabetical order, which is fine here. In some cases, such as if you use month names for your index, you can pass `sort=False`.

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. (Pandas also offers a `pivot` method, which doesn't do aggregation, which cannot handle duplicate values for index-column combinations.) To use a different aggregation function, pass 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='quarter', columns='year',
               values='A', sort=False, aggfunc=np.size)
```

The result on this data frame isn't very interesting, because there aren't any repeated intersections:

year	2018	2019
quarter		
Q1	1	1
Q2	1	1
Q3	1	1
Q4	1	1

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 athletes? In which country do players appear to consistently be the youngest?
- How tall were the tallest athletes 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.loc[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.loc[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, for all sports and 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:

```
df.pivot_table(index='Year', #1  
                columns='Team', #2  
                values='Age') #3
```

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 at Olympic games. Here was the output from my query:

Team	China	France	Great Britain	India	Switzerland	U S
Year						
1980	21.868421	23.524590	22.882507	25.506667	24.557823	2
1984	22.076336	24.369830	24.445423	24.905660	23.589744	2
1988	22.358447	24.520076	25.439560	24.000000	26.218868	2
1992	21.955752	25.140187	25.584055	24.184615	25.413194	2
1994	20.627907	24.601307	25.282051	NaN	25.500000	2
1996	22.021531	25.296629	26.746032	24.629630	27.122093	2
1998	21.784091	25.462069	27.243902	16.000000	25.641509	2
2000	22.515306	25.982833	26.406948	25.400000	27.376812	2
2002	23.127451	25.737805	26.833333	20.000000	26.238710	2
2004	23.006122	26.139073	26.303977	24.728395	27.343284	2
2006	23.457143	26.303226	26.851852	25.200000	26.284848	2
2008	23.903955	26.285714	25.200969	25.402985	27.312500	2
2010	23.239669	25.911458	26.147059	25.666667	26.548387	2
2012	23.894168	26.606635	25.922619	25.637363	27.172131	2
2014	23.400000	25.708995	25.628571	25.000000	25.855814	2
2016	23.873706	27.095238	26.653191	26.100000	25.891892	2

Next, 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:

```
df.pivot_table(index='Sport', #1
                columns='Year', #2
                values='Height', #3
                aggfunc=np.max) #4
```

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.

Finally, 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:

```
df.pivot_table(index='Year', #1
                columns='Team', #2
                values='Medal', #3
                aggfunc=np.size) #4
```

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.

## 4.5.2 Solution

```
filename = '../data/olympic_athlete_events.csv'

df = pd.read_csv(filename,
                 usecols=['Age', 'Height', 'Team',
                          'Year', 'Season',
                          'Sport', 'Medal'])#1
```





# 5 Cleaning data

In the late 1980s, my employer wanted to know how much rain had fallen in various places. 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?

It's easy to assume that a CSV file from an official-looking Web site will contain good data. But all too often, it'll have 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 misnamed a file, or entered data into the wrong field. 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 accurate as 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
<code>df.shape</code>	A two-element tuple indicating the number of rows and columns in	<code>df.shape</code>	<a href="http://mng.b">http://mng.b</a>

	a data frame		
<code>len(df)</code> or <code>len(df.index)</code>	Get the number of rows in a data frame	<code>len(df)</code> or <code>len(df.index)</code>	<a href="http://mng.b">http://mng.b</a>
<code>s.isnull</code>	Returns a boolean series indicating where there are null (typically NaN) values in the series <code>s</code>	<code>s.isnull()</code>	<a href="http://mng.b">http://mng.b</a>
<code>s.notnull</code>	Returns a boolean series indicating where there are non-null values in the series <code>s</code>	<code>s.notnull()</code>	<a href="http://mng.b">http://mng.b</a>
	Returns a boolean data frame indicating where there		

<code>df.isnull</code>	are null (typically NaN) values in the data frame <code>df</code>	<code>df.isnull()</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.replace</code>	Replace values in one or more columns with other values	<code>df.replace('a':{'b':'c'}, 'd')</code>	<a href="http://mng.b">http://mng.b</a>
<code>s.map</code>	Apply a function to each element of a series, returning the result of that application on each element	<code>s.map(lambda x: x**2)</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.fillna</code>	Replace NaN with other values	<code>df.fillna(10)</code>	<a href="http://mng.b">http://mng.b</a>
<code>df.dropna</code>	Remove rows with NaN values	<code>df = df.dropna()</code>	<a href="http://mng.b">http://mng.b</a>

s.str	Working with textual data	df['colname'].str	<a href="http://mng.b">http://mng.b</a>
str.isdigit	Returns a boolean series, indicating which strings contain only the digits 0-9	df['colname'].str.isdigit()	<a href="http://mng.b">http://mng.b</a>
pd.to_numeric	Returns a series of integers or floats, based on a series of strings	pd.to_numeric(df['colname'])	<a href="http://mng.b">http://mng.b</a>
df.sort_index	Reorder the rows of a data frame based on the values in its index, in ascending order	df = df.sort_index()	<a href="http://mng.b">http://mng.b</a>

<code>pd.read_excel</code>	Create a data frame based on an Excel spreadsheet	<code>df = pd.read_excel('myfile.xlsx')</code>	<a href="http://mng.b">http://mng.b</a>
<code>s.value_counts</code>	returns a sorted (descending frequency) series counting how many times each value appears in s	<code>s.value_counts()</code>	<a href="http://mng.b">http://mng.b</a>
<code>s.unique</code>	returns a series with the unique (i.e., distinct) values in s, including NaN (if it occurs in s)	<code>s.unique()</code>	<a href="http://mng.b">http://mng.b</a>

### 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() #1
```

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() #1
```

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() #1
```

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)#1
```



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 a 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 a result of this missing data?
- Now let's assume that tickets can be dismissed if the license plate, state, and/or street name are missing—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 a result of this missing data?

### 5.2.1 Discussion

When you're first starting off with data analytics, 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 associated 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]
```

It turns out that there's a better way, though: We can invoke the Python builtin `len` on our data frame, thus getting the number of rows:

```
len(df)
```

Not only does this give us the same answer, but in my testing, I found that `len` was twice as fast as `shape[0]`. But it turns out that we can do even better than this, by running `len` on `df.index`:

```
len(df.index)
```

In my tests, I found that `len(df.index)` runs about 45 percent faster than `len(df)`, and about 65 percent faster than `df.shape[0]`.



#### Note

The count method often seems like the most natural, obvious way to count the rows. But it has several issues:

- It ignores NaN values
- On a large data frame, it takes a long time to run

For these reasons, I generally prefer to call `len` on my data frame, or (better yet) on its index, as in `len(df.index)`.

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.

**Figure 5.1. Sample of `df`, including NaN values**

New table

	Plate ID	Registration State	Vehicle Make	Violation Time	Street Name	Vehicle Color
2752511	LHLP99	FL	HYUN	0230P	JACOB RIIS PARK	RED
964568	JXJ1561	PA	TOYOT	0119P	E 58th St	BLUE
5049760	S82HUN	NJ	HONDA	0846A	SB UNIVERSITY AVE @	BK
4248515	HYK8920	NY	FORD	1151A	NB PARK AVE @ E 83RD	GY
353397	KMF8349	PA	NaN	0850P	S/S SEAVIEW AVE	WHITE
2703401	XHXE40	NJ	NaN	1039A	W 43 ST	WH
1434853	TRD7943	OH	NaN	0937A	BASSETT AVE	WH
9585754	76654MK	NY	INTER	NaN	6TH AVE	RED
8915985	HJD9647	NY	ME/BE	NaN	29TH ST	WH
2868914	JHM3686	99	NaN	NaN	NaN	NaN

miro

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()
```

**Figure 5.2. Running `dropna` on a data frame removes all NaN values, and the rows containing them.**

New table

	Plate ID	Registration State	Vehicle Make	Violation Time	Street Name	Vehicle Color
2752511	LHLP99	FL	HYUN	0230P	JACOB RIIS PARK	RED
964568	JXJ1561	PA	TOYOT	0119P	E 58th St	BLUE
5049760	S82HUN	NJ	HONDA	0846A	SB UNIVERSITY AVE @	BK
4248515	HYK8920	NY	FORD	1151A	NB PARK AVE @ E 83RD	GY
353397	KMF8349	PA	NaN	0850P	S/S SEAVIEW AVE	WHITE
2703401	XHXE40	NJ	NaN	1039A	W 43 ST	WH
1434853	TRD7943	OH	NaN	0937A	BASSETT AVE	WH
9585754	76654MK	NY	INTER	NaN	6TH AVE	RED
8915985	HJD9647	NY	ME/BE	NaN	29TH ST	WH
2868914	JHM3686	99	NaN	NaN	NaN	NaN

dropna()

New table

	Plate ID	Registration State	Vehicle Make	Violation Time	Street Name	Vehicle Color
2752511	LHLP99	FL	HYUN	0230P	JACOB RIIS PARK	RED
964568	JXJ1561	PA	TOYOT	0119P	E 58th St	BLUE
5049760	S82HUN	NJ	HONDA	0846A	SB UNIVERSITY AVE @	BK
4248515	HYK8920	NY	FORD	1151A	NB PARK AVE @ E 83RD	GY

miro

Just how many rows did we remove when we used `dropna`? We can calculate that:

```
len(df.index) - len(all_good_df.index)
```

I get quite a large number, 447,359, 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:

```
(len(df.index) - len(all_good_df.index) ) * 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'${(len(df.index) - len(all_good_df.index) ) * 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'])
```

**Figure 5.3. Running `dropna` on a data frame, only looking at a subset of columns**

New table

	Plate ID	Registration State	Vehicle Make	Violation Time	Street Name	Vehicle Color
2752511	LHLP99	FL	HYUN	0230P	JACOB RIIS PARK	RED
964568	JXJ1561	PA	TOYOT	0119P	E 58th St	BLUE
5049760	S82HUN	NJ	HONDA	0846A	SB UNIVERSITY AVE @	BK
4248515	HYK8920	NY	FORD	1151A	NB PARK AVE @ E 83RD	GY
353397	KMF8349	PA	NaN	0850P	S/S SEAVIEW AVE	WHITE
2703401	XHXE40	NJ	NaN	1039A	W 43 ST	WH
1434853	TRD7943	OH	NaN	0937A	BASSETT AVE	WH
9585754	76654MK	NY	INTER	NaN	6TH AVE	RED
8915985	HJD9647	NY	ME/BE	NaN	29TH ST	WH
2868914	JHM3686	99	NaN	NaN	NaN	NaN

```
df[df['Plate ID'].notnull() & df['Registration State'].notnull() & df['Vehicle Make'].notnull() & df['Street Name'].notnull()]
```

New table

	Plate ID	Registration State	Vehicle Make	Violation Time	Street Name	Vehicle Color
2752511	LHLP99	FL	HYUN	0230P	JACOB RIIS PARK	RED
964568	JXJ1561	PA	TOYOT	0119P	E 58th St	BLUE
5049760	S82HUN	NJ	HONDA	0846A	SB UNIVERSITY AVE @	BK
4248515	HYK8920	NY	FORD	1151A	NB PARK AVE @ E 83RD	GY
9585754	76654MK	NY	INTER	NaN	6TH AVE	RED
8915985	HJD9647	NY	ME/BE	NaN	29TH ST	WH

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'${(len(df.index) - len(semi_good_df.index) ) * 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'${(len(df.index) - len(loosest_df.index) ) * 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'])#1
```

```
all_good_df = df.dropna()#2  
len(df.index) - len(all_good_df.index) #3
```



- 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.

### 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 mathematics involves finding ways to rewrite problems so that they're easier to understand, 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



#### 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, in which `True` values correspond with `NaN` in `df['colname']`, and `False` values correspond to non-`NaN` values in

`df['colname']`. This boolean series 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.

**Figure 5.4. Adding a new month column to our data frame, based on the month in dateofdeath**

New table

New table

New table

	dateofdeath	age	month
1277	2016-03-03	82	03
5555	2016-11-02	61	11
1022	2016-02-19	80	02
3302	2016-06-21	87	06
2214	2016-04-19	87	04
4890	2016-09-23	96	09
48	2016-01-03	83	01
751	2016-02-04	94	02
1106	2016-02-24	86	02
3915	2016-07-26	85	07

str.slice(5,7)

dateofdeath
2016-03-03
2016-11-02
2016-02-19
2016-06-21
2016-04-19
2016-09-23
2016-01-03
2016-02-04
2016-02-24
2016-07-26

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() / len(df['age'])
```

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? One way would be to rely on the `str.isdigit` method, which returns `True` if a string contains only digits (and isn't empty). (It'll return `False` if there is a `-` sign or decimal point, so it's not fail-safe for finding numbers, but will work with ages.) 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()]
```

But as is so often the case, Pandas has a more elegant solution, namely the `pd.to_numeric` function. This function—which is defined at the top `pd` level, rather than on a series or data frame—tries to create a new series with numeric values. The function tries to turn the values into integers, but if it cannot, then it returns floats, instead:

```
df['age'] = pd.to_numeric(df['age'])
```

But wait: It turns out that `pd.to_numeric` has some additional functionality, allowing us to skip the step of using `str.isdigit`: By default, `pd.to_numeric` will raise an exception if it encounters a string that cannot be turned into an `int` or `float`. But if we pass the keyword argument `errors='coerce'`, then it'll turn any values it can't convert into `NaN`. We can thus ignore all use of `str.isdigit`, and simply say:

```
df['age'] = pd.to_numeric(df['age'], errors='coerce')
```

Before we go any further, let's check the numbers that we got using `describe`:

```
df['age'].describe()
```

Here's what I got:

```
count      6505.000000
mean       100.960338
std        413.994127
min         7.000000
25%        69.000000
50%        81.000000
75%        89.000000
max       9394.000000
Name: age, dtype: float64
```

I don't know about you, but a mean age of 100 seems pretty suspicious. And a maximum age of 9,394 seems a bit high, as well. This is the result of a string containing the value '9394', which `pd.to_numeric` happily converted into a number.

Let's keep only those people younger than 120 years old:

```
df = df.loc[df['age'] < 120]
```

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'])#1

df['month'] = df['dateofdeath'].str.slice(5,7)#2
df = df.set_index('month')#3
df = df.sort_index()#4

df = df.dropna(subset=['age'])#5
df['age'] = pd.to_numeric(df['age'], errors='coerce')#6
df.loc['02':'07', 'age'].mean()#7
```

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
numpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%
20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%2
ringIO%28''%0Amonth,dateofdeath,age%5Cn08,2016-08-14,89%5Cn02,20
7%5Cn07,2016-07-16,90%5Cn03,2016-03-14,91%5Cn10,2016-10-24,91%5Cn
-11,82%5Cn11,2016-11-05,88%5Cn01,2016-01-17,90%5Cn06,2016-06-16,6
16-11-19,83%5Cn07,2016-07-22,79%5Cn05,2016-05-27,60%5Cn05,2016-05
01,2016-01-07,68%5Cn08,2016-08-27,71%5Cn09,2016-09-06,77%5Cn03,20
4%5Cn05,2016-05-23,90%5Cn02,2016-02-02,59%5Cn12,2016-12-20,28%5Cn
%0Adf%20%3D%20pd.read_csv%28data%29%0Adf%5B'month'%5D%20%3D%20df%
death'%5D.str.slice%285,7%29%0Adf%20%3D%20df.set_index%28'month'%
%3D%20df.sort_index%28%29%0Adf%20%3D%20df.dropna%28subset%3D%5B'a
&d=2022-12-29&lang=py&v=v1
```

### 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 our data contains 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 philosophical. 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 familiar 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.
```

```
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()
```

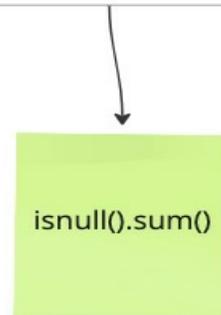
I got the following result:

```
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
```

**Figure 5.5. Finding the number of NaN values in a column by summing the result of `isnull()`**

New table

	<b>name</b>	<b>age</b>
<b>206</b>	Minahan, Dr. William Edward	44.0
<b>945</b>	Lam, Mr. Ali	NaN
<b>1156</b>	Rosblom, Miss. Salli Helena	2.0
<b>1183</b>	Salonen, Mr. Johan Werner	39.0
<b>98</b>	Douglas, Mrs. Walter Donald (Mahala Dutton)	48.0

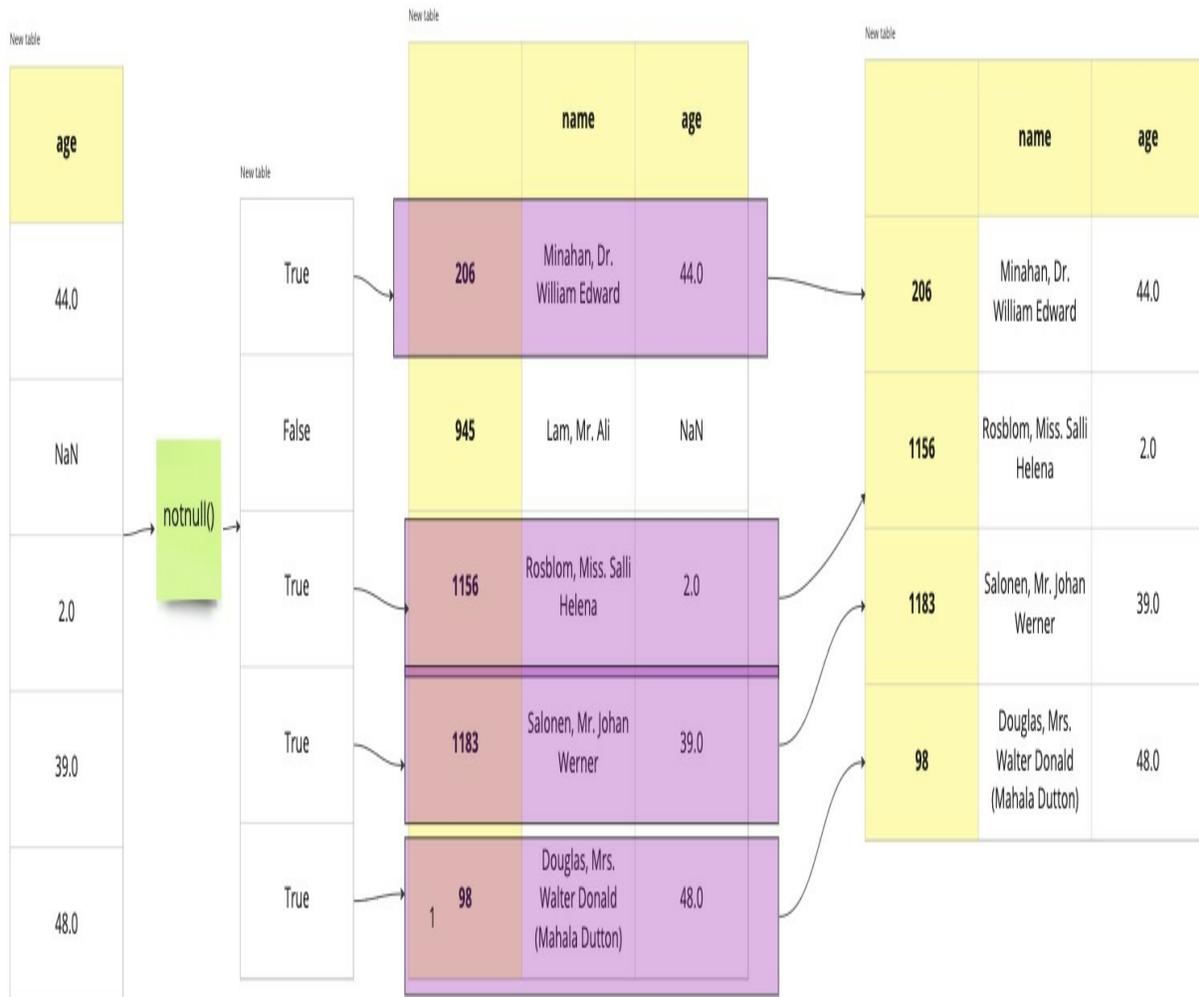


1

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.loc[df['fare'].notnull()]
df = df.loc[df['embarked'].notnull()]
```

**Figure 5.6. Removing rows in which a column contains NaN**



miro

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.

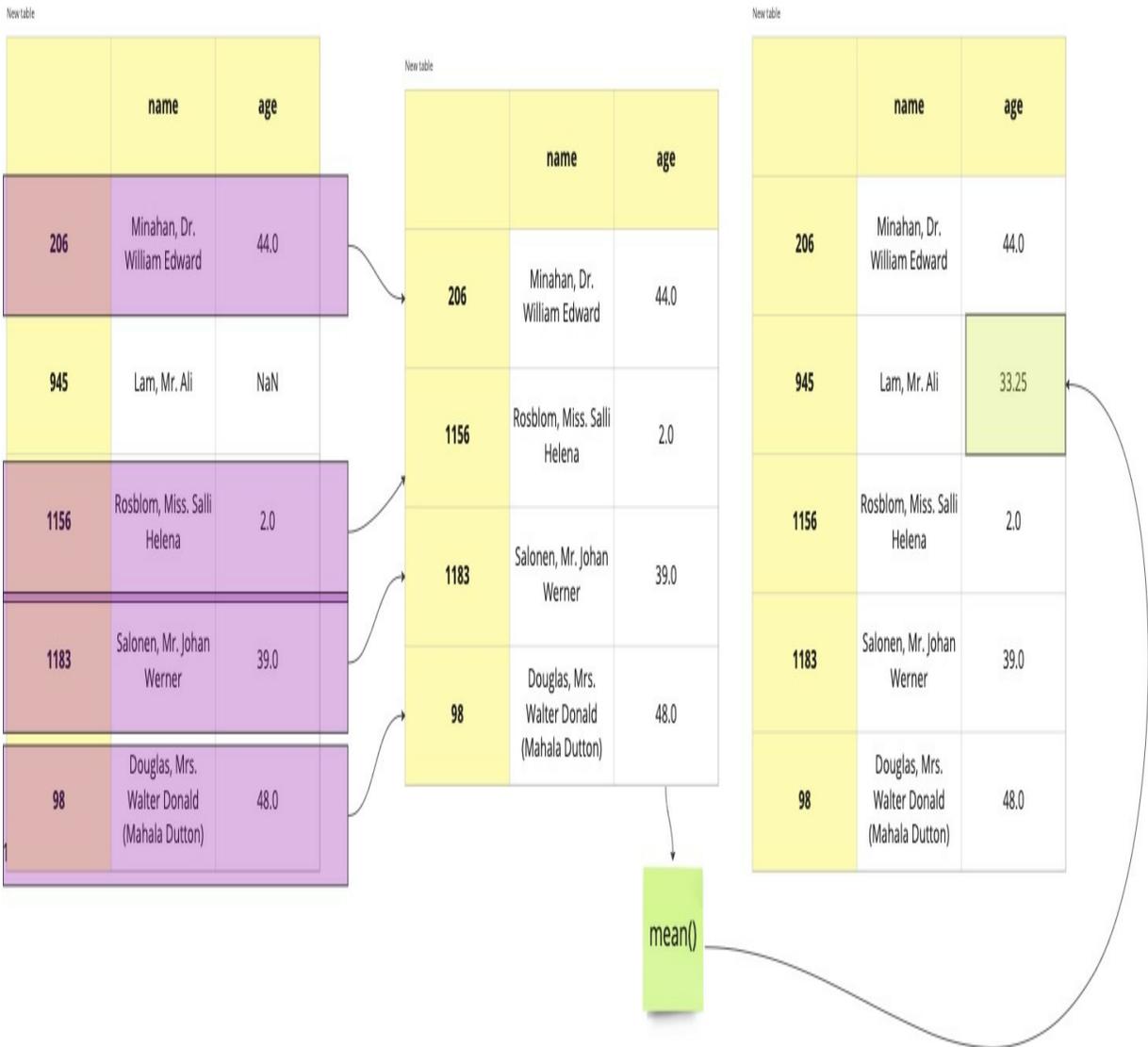
Using the mean age has some advantages: It won't affect the mean age, although it will reduce the standard deviation. It's not necessarily wrong,

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())
```

**Figure 5.7. Replacing NaN in the age column with the mean of age**



miro

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. We'll get a single float value back from this calculation—specifically, `29.8811345124283`.

- 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 invoking `fillna` will be a new series, identical to `df['age']`, except that the `NaN` values will be replaced with `29.8811345124283`, the float we got back in the previous step.
- The result of `df['age'].fillna` is a new series, which we then 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)#1

df.columns[df.isnull().sum() > 0 ]#2
df.isnull().sum()#3

df['age'] = df['age'].fillna(df['age'].mean())#4
df = df[df['fare'].notnull() ]#5
df = df[df['embarked'].notnull() ] #6
df['home.dest'] = df['home.dest'].fillna(df['home.dest'].mode())#
```

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
numpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%
0Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%20
ngIO%28' '%0A,pclass,survived,name,sex,age,sibsp,parch,ticket,fa
rked,boat,body,home.dest%5Cn1139,3,0,%22Rekic,%20Mr.%20Tido%22,
0,0,349249,7.8958,,S,,,%5Cn533,2,1,%22Phillips,%20Miss.%20Alice%2
0Louisa%22,female,21.0,0,1,S.O./P.P.%202,21.0,,S,12,,%22Ilfracomb
%22%5Cn459,2,0,%22Jacobsohn,%20Mr.%20Sidney%20Samuel%22,male,42.0
7,27.0,,S,,London%5Cn1150,3,0,%22Risien,%20Mr.%20Samuel%20Beard%
,0,364498,14.5,,S,,,%5Cn393,2,0,%22Denbury,%20Mr.%20Herbert%22,ma
0,C.A.%2031029,31.5,,S,,,%22Guernsey%20/%20Elizabeth,%20NJ%22%5Cn
22Sandstrom,%20Miss.%20Marguerite%20Rut%22,female,4.0,1,1,PP%2095
,S,13,,%5Cn5,1,1,%22Anderson,%20Mr.%20Harry%22,male,48.0,0,0,1995
2,S,3,,%22New%20York,%20NY%22%5Cn231,1,1,%22Peuchen,%20Major.%20A
dfrey%22,male,52.0,0,0,113786,30.5,C104,S,6,,%22Toronto,%20ON%22%
,%22Ashby,%20Mr.%20John%22,male,57.0,0,0,244346,13.0,,S,,,%22West
,%20NJ%22%5Cn887,3,1,%22Johannesen-Bratthammer,%20Mr.%20Bernt%22,
65306,8.1125,,S,13,,%5Cn%0A' '%29%0A%0Adf%20%3D%20pd.read_csv%28d
f.columns%5Bdf.isnull%28%29.sum%28%29%20%3E%200%20%5D%0Adf.isnull
%28%29%0Adf%5B'age'%5D%20%3D%20df%5B'age'%5D.fillna%28df%5B'age'%
%29%29&d=2023-01-04&lang=py&v=1
```

### 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.dest` 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` column with values from embarked. (Since values in embarked and `home.dest` are 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 once encountered this while 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.

While people understand that these all refer to the same country, a computer doesn't. 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, 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:

```
len(df['Vehicle Color'].value_counts().index)
```

`value_counts` is 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 most to least common. Because `value_counts` returns a series, you can ask for its index, and call `len` on it.

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': 'YELLOW', 'BLK': 'BLACK', 'GRY': 'GRAY',
            'WHT': 'WHITE', 'WHI': 'WHITE', 'OR': 'ORANGE',
            'BK.': 'BLACK', 'WT': 'WHITE', 'WT.': 'WHITE'}
```

The above dict has 18 key-value pairs, in order to standardize 18 color names.

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)
```



#### Note

Any values not in `colormap` will remain unchanged. And the match in `colormap` must be a precise match—including whitespace, punctuation, and case.

If we check the number of distinct colors again:

```
len(df['Vehicle Color'].value_counts().index)
```

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 entered 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': 'YELLOW', '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:

```
len(df['Vehicle Color'].value_counts().index)
```

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.





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 type of action is known in the Pandas world as "split-apply-combine," and is our focus in this chapter. If you have experience with SQL and relational databases, then you'll find many similarities, in both principle and name, with functionality in Pandas.

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 SQL technique is "joining," which lets us keep our data in small, specific data frames, combining them when only when we need to. For example, one data frame might list each sales region and that region's manager, while a second might contain this quarter's regional sales results. In order to show the monthly sales results for each region along with each region's manager, you'll want to join the data frames together.

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.

You'll want to have each of these techniques—grouping, joining, and sorting—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
<code>s.isnull</code>	Returns a boolean series indicating where there are null (typically NaN) values in the series <code>s</code>	<code>s.isnull()</code>	<a href="http://mn">http://mn</a>
<code>df.sort_index</code>	Reorder the rows of a data frame based on the values in its index, in ascending order	<code>df = df.sort_index()</code>	<a href="http://mn">http://mn</a>
<code>df.sort_values</code>	Reorder the rows of a data frame based on the values in one or more specified columns	<code>df = df.sort_values('distance')</code>	<a href="http://mn">http://mn</a>

<code>df.transpose()</code> or <code>df.T</code>	Returns a new data frame with the same values as <code>df</code> , but with the columns and index exchanged	<code>df.transpose()</code> or <code>df.T</code>	<a href="http://mn">http://mn</a>
<code>df.expanding</code>	Lets us run window functions on an expanding (growing) set of rows.	<code>df.expanding().sum()</code>	<a href="http://mn">http://mn</a>
<code>df.rolling</code>	Lets us run window functions on an expanding (growing) set of rows.	<code>df.rolling(3).mean()</code>	<a href="http://mn">http://mn</a>
<code>df.pct_change</code>	For a given data frame, indicates the percentage difference between each cell and the corresponding cell in the previous row.	<code>df.pct_change()</code>	<a href="http://mn">http://mn</a>
<code>df.groupby</code>	Allows us to invoke one or more aggregate methods for each	<code>df.groupby('year')</code>	<a href="http://mn">http://mn</a>

	value in a particular column.		
<code>df.loc</code>	Retrieve selected rows and columns	<code>df.loc[:, 'passenger_count'] = df['passenger_count']</code>	<a href="http://mn">http://mn</a>
<code>s.iloc</code>	access elements of a series by position	<code>s.iloc[0]</code>	<a href="http://mn">http://mn</a>
<code>df.dropna</code>	Remove rows with NaN values	<code>df = df.dropna()</code>	<a href="http://mn">http://mn</a>
<code>s.unique</code>	Get the unique values in a series <b>NOTE Pandas drop_duplicates is better</b>	<code>s.unique()</code>	<a href="http://mn">http://mn</a>
<code>df.join</code>	Join two data frames together based on their indexes	<code>df.join(other_df)</code>	<a href="http://mn">http://mn</a>
<code>df.merge</code>	Join two data frames together based on any columns	<code>df.merge(other_df)</code>	<a href="http://mn">http://mn</a>

<code>df.corr</code>	Show the correlation between the numeric columns of a data frame	<code>df.corr()</code>	<a href="http://mn">http://mn</a>
<code>s.to_frame</code>	Turn a series into a one-column data frame	<code>s.to_frame()</code>	<a href="http://mn">http://mn</a>
<code>s.removesuffix</code>	Returns a new string with the same contents as <code>s</code> , but without a specified suffix (if it's there)	<code>s.removesuffix('.csv')</code>	<a href="http://mn">http://mn</a>
<code>s.removeprefix</code>	Returns a new string with the same contents as <code>s</code> , but without a specified prefix (if it's there)	<code>s.removeprefix('abcd')</code>	<a href="http://mn">http://mn</a>
<code>s.title</code>	Returns a new string based on <code>s</code> , in which each word starts with a capital letter	<code>s.title('hello out there')</code>	<a href="http://mn">http://mn</a>
	Returns one new data frame based		

<code>pd.concat</code>	on a list of data frames passed to <code>pd.concat</code>	<code>pd.concat([df1, df2, df3])</code>	<a href="http://mn">http://mn</a>
------------------------	---	---	-----------------------------------

## 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.

If you want to display data—say, sales records, network statistics, or inflation projections—then you'll likely want to order them. How you order them depends on the context, though. Sales records might need to be ordered by department, network statistics might need to be ordered by subnets, and inflation projections 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. 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 9 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`.

```
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)
```

**Figure 6.1. Running `sort_values` on a data frame returns a new data frame with the same rows, but ordered according to the named column**

New table

	passenger_count	trip_distance	total_amount
3626666	0	1.50	8.80
974073	1	0.48	6.80
6370644	1	1.90	13.30
2125992	2	2.10	13.80
4959601	5	1.76	12.25

New table

	passenger_count	trip_distance	total_amount
2125992	2	2.10	13.80
6370644	1	1.90	13.30
4959601	5	1.76	12.25
3626666	0	1.50	8.80
974073	1	0.48	6.80

sort\_values('trip\_d  
istance',  
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']
```

**Figure 6.2. Running `sort_values` on a data frame, then keeping only one column**

New table

	passenger_count	trip_distance	total_amount
3626666	0	1.50	8.80
974073	1	0.48	6.80
6370644	1	1.90	13.30
2125992	2	2.10	13.80
4959601	5	1.76	12.25

New table

	total_amount
2125992	13.80
6370644	13.30
4959601	12.25
3626666	8.80
974073	6.80

sort\_values('trip\_d  
istance',  
ascending=False)

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]
```

**Figure 6.3. Running `sort_values` on a data frame, then keeping only one column, then getting only the first rows with `iloc`**

New table

	passenger_count	trip_distance	total_amount
3626666	0	1.50	8.80
974073	1	0.48	6.80
6370644	1	1.90	13.30
2125992	2	2.10	13.80
4959601	5	1.76	12.25

New table

	total_amount
2125992	13.80
6370644	13.30
4959601	12.25
3626666	8.80
974073	6.80

sort\_values('trip\_d  
istance',  
ascending=False)

.iloc[3]

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 290.01000000000000076, which I think we can reasonably round to an average of \$290 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. First, we sort our data frame by values:

```
df.sort_values('trip_distance')
```

Remember that by default, `sort_values` sorts in ascending order, so we don't need to specify anything there. Once again, we keep only the `total_amount` column:

```
df.sort_values('trip_distance')['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')[
    '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.



#### Note

Wondering whether it's faster to run `tail` or `iloc` with a slice? From some performance checks that I did, they were almost exactly the same.

Finally, we invoke `mean()` on the 20 longest-ride fares:

```
df.sort_values('trip_distance')[
    'total_amount'].iloc[-20:].mean()
```

And the result is... 290.01000000000001. 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 (slight) 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 290.010000000000000076. But of course, now our column consumes twice as much memory as before.



## Note

If 128-bit floats are the most accurate, then why not always use them?

First, because they're very large, at 16 bytes (!) per number. If you have 1 million floats, that translates into about 16 MB of data. Not every problem you're trying to solve needs such extreme accuracy.

But there's a second problem, namely that I've found 128-bit floats cause some problems. On my Mac, some Pandas methods didn't work when my columns had a dtype of `np.float128`. And it seems that `np.float128` doesn't even exist on computers running Windows.

So if you need the precision, and if you're on a platform that supports them, and if the Pandas methods you need can use them, then sure—go ahead and use `np.float128`. But keep in mind that this will make your program less portable.

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])
```

**Figure 6.4. Sorting a data frame by `passenger_count` (ascending order), then `trip_distance` (descending order)**

New table

	passenger_count	trip_distance	total_amount
3626666	0	1.50	8.80
974073	1	0.48	6.80
6370644	1	1.90	13.30
2125992	2	2.10	13.80
4959601	5	1.76	12.25

New table

	passenger_count	trip_distance	total_amount
3626666	0	1.50	8.80
6370644	1	1.90	13.30
974073	1	0.48	6.80
2125992	2	2.10	13.80
4959601	5	1.76	12.25

```
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 135.49740000000001.

## 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()#1
df.sort_values('trip_distance')[
              'total_amount'].iloc[-20:].mean()#2
df.sort_values(['passenger_count',
               'trip_distance'],
              ascending=[True, False])[
               'total_amount'].iloc[:50].mean()#3
```

You can explore a version of this in the Pandas Tutor at:

<https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0Anumpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%0ASeries,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%20>

```
ngI0%28''',passenger_count,trip_distance,total_amount%5Cn0,1,1.5,
99999992895%5Cn1,1,2.6,16.300000000000000071%5Cn2,3,0.0,5.79999999
%5Cn3,5,0.0,7.5499999999999998224%5Cn4,5,0.0,55.54999999999999715
.0,13.310000000000000497%5Cn6,5,0.0,55.549999999999997158%5Cn7,1,
000000000007105%5Cn8,1,3.7,18.5%5Cn9,2,2.1,13.0%5Cn10,2,2.8,19.55
00071%5Cn11,1,0.7,8.5%5Cn12,1,8.7,42.9500000000000002842%5Cn13,1,6
Cn14,1,2.7,15.3000000000000007105%5Cn15,1,0.38,4.7999999999999998
,1,0.55,9.75%5Cn17,1,0.3,6.36000000000000003197%5Cn18,1,1.42,9.359
9994316%5Cn19,1,1.72,10.3000000000000007105%5Cn'''%29%0A%0Adf%20%
ad_csv%28data%29%0A%0Adf.sort_values%28'trip_distance',%20%0A%20%
20%20%20%20%20%20%20%20%20ascending%3DFalse%29%5B'total_amo
iloc%5B%3A20%5D.mean%28%29&d=2023-02-08&lang=py&v=1
```

## 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 20 longest trips.

### 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 products 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 0x13174f9
```

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()
```

You can see this visually here:





- 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
```

## 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 the 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 ( $\geq 2$  miles and  $\leq 10$  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

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 loaded 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'])
```

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, #1  
          df.loc[df['passenger_count'] == i, #2  
                 'total_amount' #3  
                ].mean()) #4
```

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.

**Figure 6.5. Graphical depiction of how `groupby` and then `mean` work together**

New table

	passenger_count	trip_distance	total_amount
7457997	1	0.30	5.80
5176884	5	0.78	7.80
3808538	1	2.09	13.00
4746439	6	0.74	5.80
6897983	1	2.66	16.56
3093558	1	2.70	16.00
3354288	1	2.61	18.30
5492350	1	1.70	13.00
6451927	1	0.76	8.80
3070078	1	2.20	13.50
502287	2	1.00	11.62
1924539	1	2.11	14.76
858620	3	4.10	17.80
7037227	1	0.95	10.70
2237791	1	2.11	10.30
2805107	1	2.70	11.80
3601249	1	1.21	6.30
4306225	1	1.90	16.56
1934421	2	6.30	32.56
4333172	3	0.78	9.96

New table

passenger_count	mean(total_amount)
1	12.527143
2	22.090000
3	13.880000
4	
5	7.800000
6	5.800000

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()
```

**Figure 6.6. Graphical depiction of how you can run `sort_values` on the `groupby` result**

New table

passenger_count	mean(total_amount)
1	12.527143
2	22.090000
3	13.880000
4	
5	7.800000
6	5.800000

sort\_values

New table

passenger_count	mean(total_amount)
6	5.8
5	7.8
1	12.527143
3	13.880000
2	22.09

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( #1
    df['trip_distance'], #2
    [df['trip_distance'].min(), 2, 10,
     df['trip_distance'].max()], #3
    labels=['short', 'medium', 'long'], #4
    include_lowest=True) #5
```

With 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)
```

The above says: 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 distinct 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'])

df.groupby('passenger_count')['total_amount']
    .mean().sort_values()#1
df.groupby('passenger_count')['total_amount']
    .mean().sort_index()#2

df['trip_distance_group'] = pd.cut(
    df['trip_distance'],
    [df['trip_distance'].min(), 2, 10,
     df['trip_distance'].max()],
    labels=['short', 'medium', 'long']) #3
df.groupby('trip_distance_group')['passenger_count']
    .mean().sort_values(ascending=False) #4
```

You can explore a version of this in the Pandas Tutor at:

### 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.

#### 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.

**Figure 6.7. Graphical depiction of `products_df`**

New table

	product_id	name	wholesale_price	retail_price	department
0	23	computer	500.0	1000	electronics
1	96	Python Workout	35.0	75	books
2	97	Pandas Workout	35.0	75	books
3	15	banana	0.5	1	food
4	87	sandwich	3.0	5	food

miro

**Figure 6.8. Graphical depiction of sales\_df**

New table

	product_id	date	quantity
0	23	2021-August-10	1
1	96	2021-August-10	5
2	15	2021-August-10	3
3	87	2021-August-10	2
4	15	2021-August-11	1
5	96	2021-August-11	1
6	23	2021-August-11	2
7	87	2021-August-12	2
8	97	2021-August-12	6
9	97	2021-August-12	1
10	87	2021-August-13	2
11	23	2021-August-13	1
12	15	2021-August-14	2

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')
```

**Figure 6.9. Graphical depiction of `products_df` with `product_id` as the index**

New table

product_id	name	wholesale_price	retail_price	department
23	computer	500.0	1000	electronics
96	Python Workout	35.0	75	books
97	Pandas Workout	35.0	75	books
15	banana	0.5	1	food
87	sandwich	3.0	5	food

**Figure 6.10. Graphical depiction of sales\_df with product\_id as the index**

New table

product_id	date	quantity
23	2021-August-10	1
96	2021-August-10	5
15	2021-August-10	3
87	2021-August-10	2
15	2021-August-11	1
96	2021-August-11	1
23	2021-August-11	2
87	2021-August-12	2
97	2021-August-12	6
97	2021-August-12	1
87	2021-August-13	2
23	2021-August-13	1
15	2021-August-14	2

Now that our data frames have a common reference point in the index, we can create a new data frame combining the two:

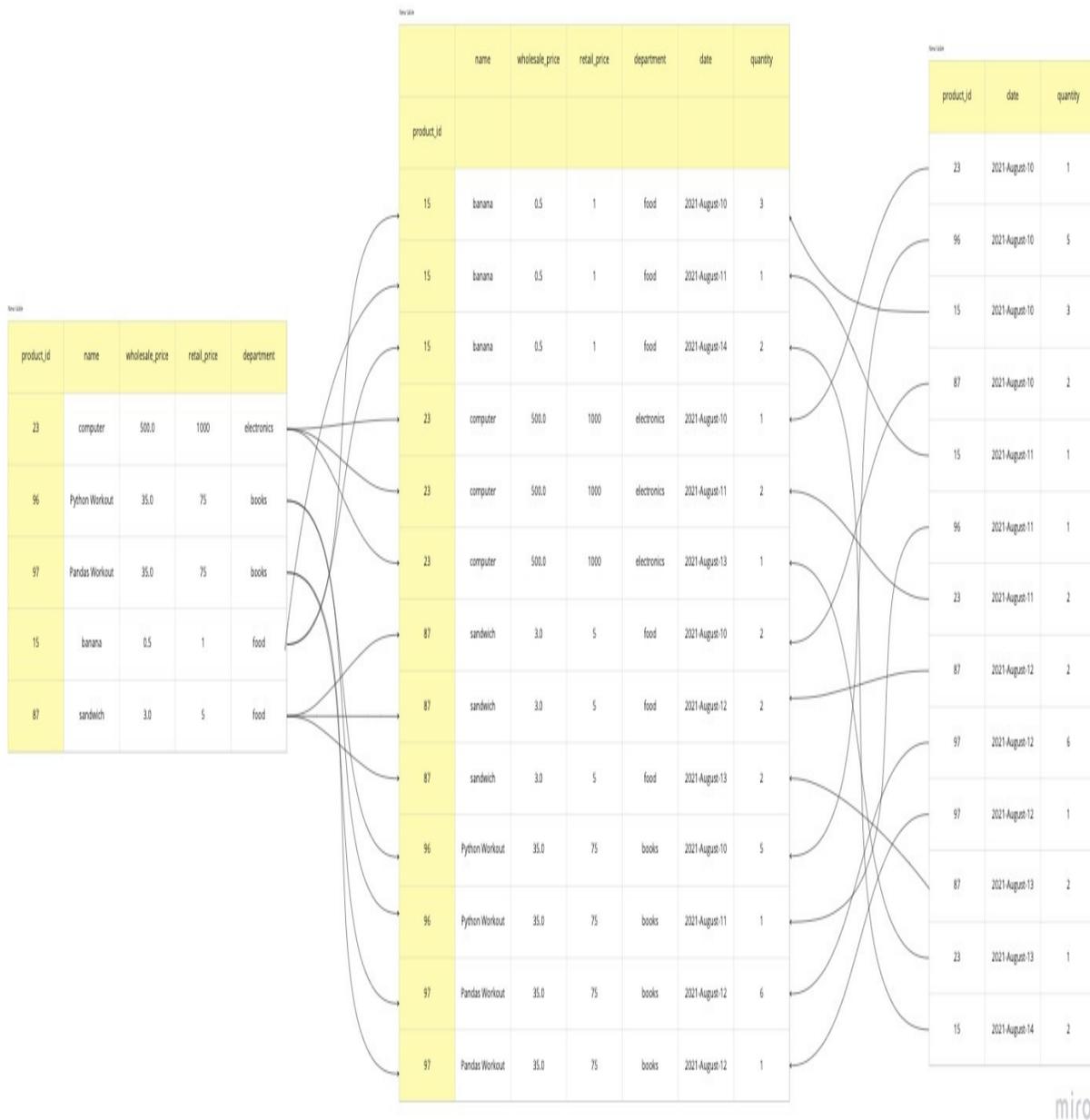
```
products_df.join(sales_df)
```

The result of this join is a new table with 13 rows and 6 columns. The columns would combine all of the columns from `products_df`, and then all of the columns from `sales_df`:

- name
- wholesale\_price
- retail\_price
- department
- date
- quantity

Each row is the result of a match between the index (`product_id`) on the left (from `products_df`) and the index (`product_id`) on the right (from `sales_df`). Because several products had multiple sales, we end up with more rows than either of the original tables contained.

**Figure 6.11. Graphical depiction of joining `products_df` and `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 pieces. 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

Before the covid-19 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, changed all of that, with many countries restricting who could enter and leave, and under what circumstances.

This was certainly a serious problem for corporate Python trainers. But it was 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 (Organization for Economic Cooperation and Development), which the Economist 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, across 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.
- 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 datasets. 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 abbreviations. 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 differ in how much tourist income they receive. Breaking it apart by country will give us many more insights than an overall mean.

How 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-letter 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 (on average,

per year) 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 (on average, per year) 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`.

**Figure 6.12. Graphical depiction of making the `LOCATION` column the index of `tourism_df`**

New table

	LOCATION	SUBJECT	TIME	Value
0	AUS	INT_REC	2008	31159.8
1	AUS	INT_REC	2009	29980.7
2	AUS	INT_REC	2010	35165.5
3	AUS	INT_REC	2011	38710.1
4	AUS	INT_REC	2012	38003.7
1229	SRB	INT-EXP	2015	1253.644
1230	SRB	INT-EXP	2016	1351.098
1231	SRB	INT-EXP	2017	1549.183
1232	SRB	INT-EXP	2018	1837.317
1233	SRB	INT-EXP	2019	1999.313

set\_index('LOCATION')

New table

LOCATION	SUBJECT	TIME	Value
AUS	INT_REC	2008	31159.8
AUS	INT_REC	2009	29980.7
AUS	INT_REC	2010	35165.5
AUS	INT_REC	2011	38710.1
AUS	INT_REC	2012	38003.7
SRB	INT-EXP	2015	1253.644
SRB	INT-EXP	2016	1351.098
SRB	INT-EXP	2017	1549.183
SRB	INT-EXP	2018	1837.317
SRB	INT-EXP	2019	1999.313

```
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.

**Figure 6.13. Graphical depiction of making the `LOCATION` column the index of `locations_df`**

New table

	LOCATION	NAME
0	AUS	Australia
1	AUT	Austria
2	BEL	Belgium
3	CAN	Canada
4	DNK	Denmark
5	FIN	Finland
6	FRA	France
7	DEU	Germany
8	HUN	Hungary
9	ITA	Italy
10	JPN	Japan
11	KOR	Korea
12	GBR	United Kingdom
13	USA	United States
14	BRA	Brazil
15	ISR	Israe



New table

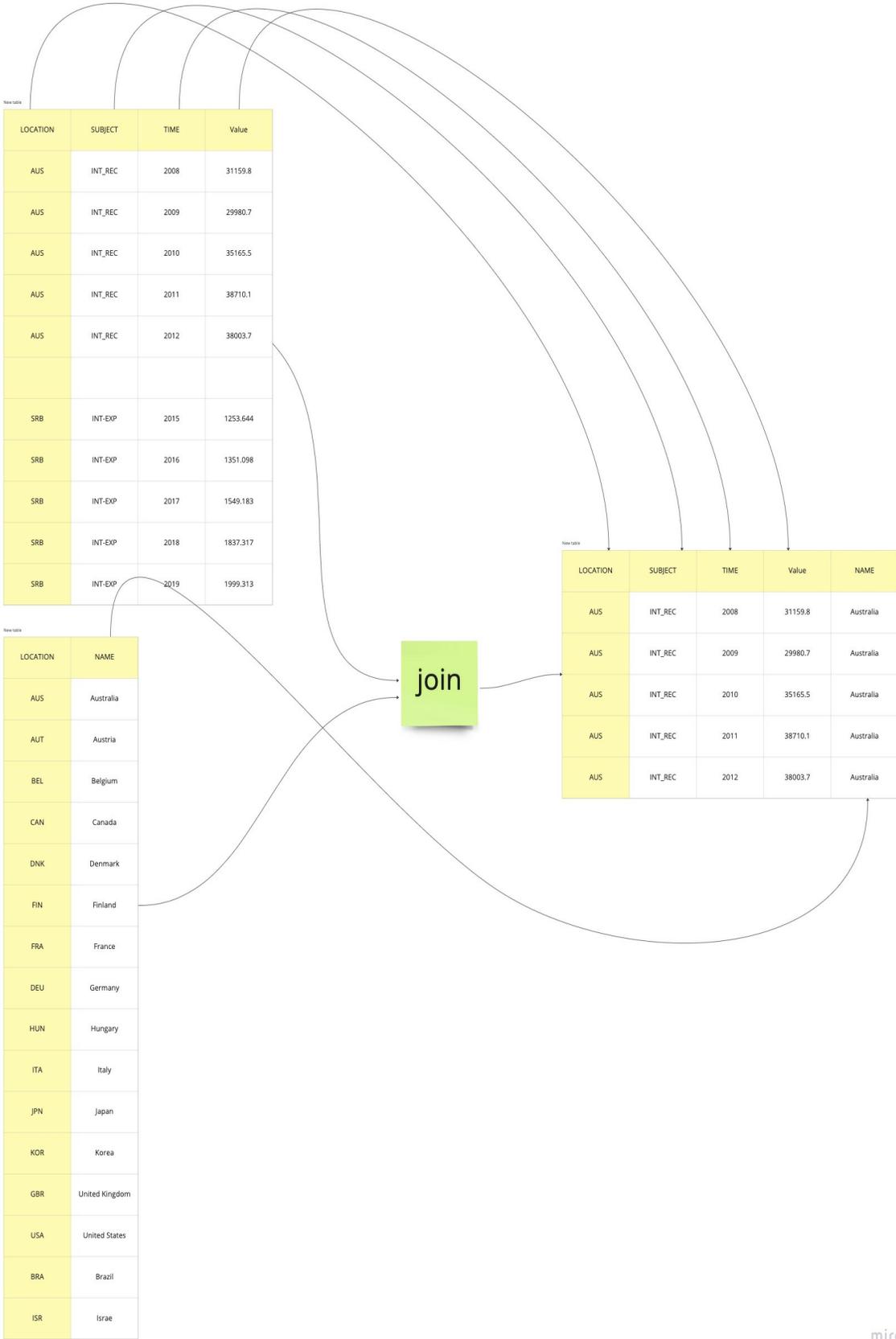
LOCATION	NAME
AUS	Australia
AUT	Austria
BEL	Belgium
CAN	Canada
DNK	Denmark
FIN	Finland
FRA	France
DEU	Germany
HUN	Hungary
ITA	Italy
JPN	Japan
KOR	Korea
GBR	United Kingdom
USA	United States
BRA	Brazil
ISR	Israe

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.

**Figure 6.14. Graphical depiction of joining `locations_df` and `tourism_df`. Notice that any rows referring to a country not in `locations_df` is dropped from the result.**



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, #1
    usecols=['LOCATION',
    'SUBJECT', 'TIME', 'Value'])

tourism_df.loc[tourism_df['SUBJECT'] ==
    'INT_REC'].groupby('LOCATION')['Value'].mean().sort_values(ascending=False).head(

tourism_df.loc[tourism_df['SUBJECT'] ==
    'INT-EXP'].groupby('LOCATION')['Value'].mean().sort_values().head())#3

locations_filename = '../data/oecd_locations.csv'
locations_df = pd.read_csv(locations_filename,
    header=None,
    names=['LOCATION', 'NAME'],
    index_col='LOCATION')#4
```

```

fullname_df = locations_df.join(
    tourism_df.set_index('LOCATION'))#5

fullname_df.loc[fullname_df['SUBJECT'] ==
    'INT_REC'].groupby('NAME')['
    Value'].mean().sort_values(ascending=False).head
fullname_df.loc[fullname_df['SUBJECT'] ==
    'INT-EXP'].groupby('NAME')['
    Value'].mean().sort_values().head()#7

```

You can explore a version of this in the Pandas Tutor at:

[### 6.4.3 Beyond the exercise](https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0Anumpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%0ASeries,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0AtourismD%20StringIO%28''%0A, LOCATION, SUBJECT, TIME, Value%5Cn665, USA, INT-32324.0%5Cn45, AUT, INT-EXP, 2018, 14272.0%5Cn231, HUN, INT_REC, 2015, 6944, EST, INT_REC, 2012, 1593.8%5Cn895, IDN, INT_REC, 2017, 14691.072%5Cn7REC, 2012, 4043.182%5Cn1133, SVN, INT-EXP, 2014, 1642.6%5Cn1141, ZAF, INT.10706.33%5Cn52, BEL, INT_REC, 2014, 15234.9%5Cn618, TUR, INT-EXP, 2011, 53, IRL, INT-EXP, 2012, 6026.0%5Cn1003, MAR, INT_REC, 2013, 8201.688%5Cn5,C, 2013, 36965.0%5Cn386, MEX, INT_REC, 2014, 16606.9%5Cn1137, SVN, INT-EX1.2%5Cn18, AUS, INT-EXP, 2014, 38774.5%5Cn184, DEU, INT_REC, 2012, 51727.RT, INT-EXP, 2018, 6544.0%5Cn1149, ZAF, INT_REC, 2019, 9063.867%5Cn156, F, 2017, 6640.8%5Cn247, ISL, INT_REC, 2009, 1257.888%5Cn1196, KAZ, INT_REC.685%5Cn916, ISR, INT_REC, 2014, 6616.2%5Cn317, JPN, INT_REC, 2013, 16904,ZAF, INT-EXP, 2012, 7144.71%5Cn390, MEX, INT_REC, 2018, 23802.7%5Cn55, B, 2017, 9598.9%5Cn299, ITA, INT_REC, 2017, 46245.7%5Cn677, BRA, INT_REC, 28%5Cn1231, SRB, INT-EXP, 2017, 1549.183%5Cn492, PRT, INT_REC, 2008, 139491, LVA, INT_REC, 2015, 1282.1%5Cn298, ITA, INT_REC, 2016, 42309.8%5Cn142,C, 2014, 5459.5%5Cn459, NOR, INT-EXP, 2008, 13794.1%5Cn1001, MAR, INT_REC.423%5Cn692, BRA, INT-EXP, 2016, 17067.798%5Cn808, HRV, INT-EXP, 2012, 9698, MAR, INT_REC, 2008, 8885.294%5Cn726, CHL, INT_REC, 2014, 3202.0%5Cn41REC, 2018, 26043.5%5Cn418, NLD, INT-EXP, 2013, 21008.6%5Cn85, CAN, INT-EX46.3%5Cn108, CZE, INT-EXP, 2013, 4697.9%5Cn461, NOR, INT-EXP, 2010, 135019%0A%0Alocation_data%20%3D%20StringIO%28''%0ALOCATION, NAME%5CnAU%5CnAUT, Austria%5CnBEL, Belgium%5CnCAN, Canada%5CnDNK, Denmark%5CnFI5CnFRA, France%5CnDEU, Germany%5CnHUN, Hungary%5CnITA, Italy%5CnJPN, J,Korea%5CnGBR, United%20Kingdom%5CnUSA, United%20States%5CnBRA, BrazIsrael%5Cn%0A''%29%0A%0Atourism_df%20%3D%20pd.read_csv%28tourismAlocations_df%20%3D%20pd.read_csv%28location_data%29%0A%0Alocatio%28tourism_df.set_index%28'LOCATION'%29%29%0A&d=2023-02-09&lang=p</a></p>
</div>
<div data-bbox=)

- 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 more people to vote, then we'll want to count voters from each age range, location, or ethnicity, 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.

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

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(
    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 successfully 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.

**Figure 6.15. Graphical depiction of using `pd.concat` to join separate data frames into a single one**

New table

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

New table

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

New table

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

New table

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

pd.concat

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.removeprefix(
        '../data/').removesuffix(
        '.csv').split(',')
    one_df = pd.read_csv(one_filename,
                        usecols=[0, 1, 2],
                        names=['date_time',
                              'max_temp',
                              'min_temp'],
                        header=0)
    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 minimum 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, grouping 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. As we'll see in Chapter 9, Pandas allows us to work with actual dates and times, performing calculations and comparisons on them. Here, the `date_time` column was a string, which made it possible to do some basic queries, but nothing as sophisticated as what is possible with actual timestamp objects, as you'll see.

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 = [] #1

for one_filename in glob.glob('../data/*,*.csv'): #2
    print(f'Loading {one_filename}...')
    city, state = one_filename.removeprefix(
        '../data/').removesuffix(
        '.csv').split(',') #3
    one_df = pd.read_csv(one_filename,
                        usecols=[0, 1, 2], #4
                        names=['date_time',
                              'max_temp',
                              'min_temp'], #5
                        header=0) #6
    one_df['city'] = city.replace('+', ' ').title() #7
    one_df['state'] = state.upper() #8
    all_dfs.append(one_df) #9

df = pd.concat(all_dfs) #10

df.groupby(['state', 'city'])[
    'date_time'].min().sort_values() #11
df.groupby(['state', 'city'])[
    'date_time'].max().sort_values() #12
```

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

You can explore a version of this in the Pandas Tutor at:

[### 6.5.3 Beyond the exercise](https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0Anumpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%20ringIO%28''%0A,date_time,max_temp,min_temp,city,state%5Cn272,20100%3A00%3A00,-3,-11,Springfield,IL%5Cn210,2019-01-06%2006%3A00%3Aingfield,IL%5Cn89,2018-12-22%2003%3A00%3A00,13,2,Boston,MA%5Cn4434%2009%3A00%3A00,11,4,Boston,MA%5Cn272,2019-01-14%2000%3A00%3A00,on,MA%5Cn604,2019-02-24%2012%3A00%3A00,7,-1,Springfield,MA%5Cn6307%2018%3A00%3A00,14,8,San%20Francisco,CA%5Cn485,2019-02-09%2015%3,-6,Boston,MA%5Cn507,2019-02-12%2009%3A00%3A00,-4,-10,Springfield,2019-01-25%2009%3A00%3A00,-15,-20,Chicago,IL%5Cn699,2019-03-08%2A00,15,11,Los%20Angeles,CA%5Cn53,2018-12-17%2015%3A00%3A00,3,0,ChCn506,2019-02-12%2006%3A00%3A00,2,0,Chicago,IL%5Cn296,2019-01-17%3A00,-1,-6,New%20York,NY%5Cn536,2019-02-16%2000%3A00%3A00,14,10,Les,CA%5Cn575,2019-02-20%2021%3A00%3A00,11,5,San%20Francisco,CA%5C02-01%2000%3A00%3A00,-8,-17,Albany,NY%5Cn404,2019-01-30%2012%3A00,San%20Francisco,CA%5Cn18,2018-12-13%2006%3A00%3A00,15,10,San%20FA%5Cn467,2019-02-07%2009%3A00%3A00,12,6,San%20Francisco,CA%5Cn4879%2021%3A00%3A00,-2,-6,Boston,MA%5Cn362,2019-01-25%2006%3A00%3A00Chicago,IL%5Cn638,2019-02-28%2018%3A00%3A00,14,9,San%20Francisco,2019-02-28%2006%3A00%3A00,-2,-13,Springfield,MA%5Cn510,2019-02-10%3A00,2,-1,Springfield,IL%5Cn42,2018-12-16%2006%3A00%3A00,2,0,SpMA%5Cn60,2018-12-18%2012%3A00%3A00,4,-1,New%20York,NY%5Cn440,20190%3A00%3A00,14,10,Los%20Angeles,CA%5Cn600,2019-02-24%2000%3A00%3ANew%20York,NY%5Cn292,2019-01-16%2012%3A00%3A00,16,11,Los%20Angele0A''%29%0A%0Adf%20%3D%20pd.read_csv%28data%29%0A%0Adf.groupby%28,%20'city'%5D%29%5B'min_temp'%5D.min%28%29%20%0A&d=2023-02-10&lan</a></p></div><div data-bbox=)

- 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?

## 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.

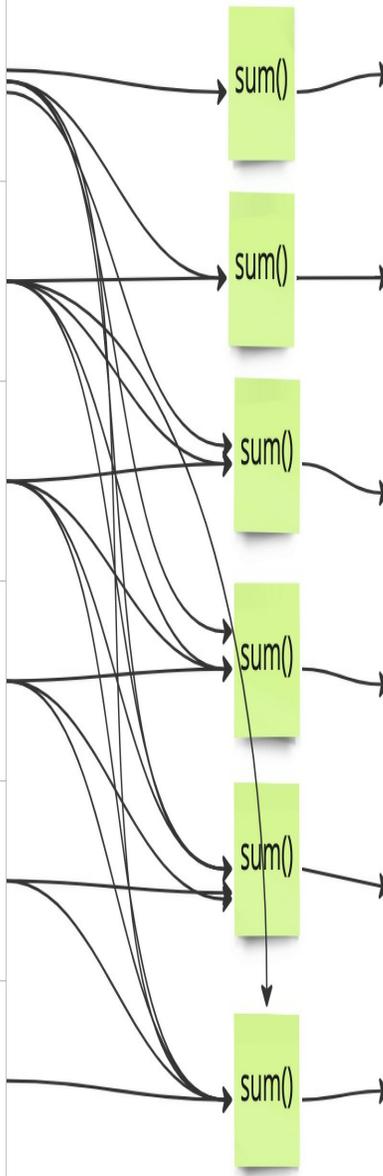
**Figure 6.16. Graphical depiction of an expanding window function with sum.**

New table

	sales	quarters
0	100	Q1
1	150	Q2
2	200	Q3
3	250	Q4
4	200	Q1
5	150	Q2

New table

	sales
0	100
1	250
2	450
3	700
4	900
5	1050



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()
```

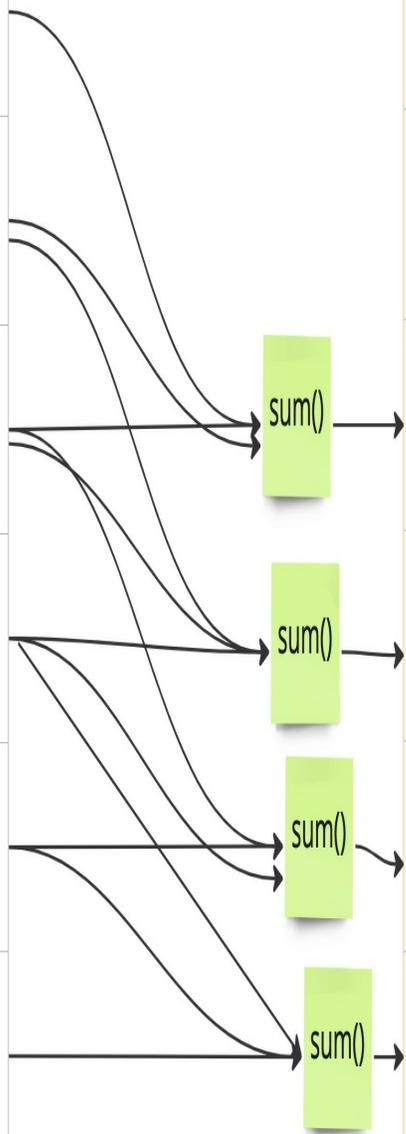
**Figure 6.17. Graphical depiction of a rolling window function (looking at 3 lines) with sum.**

New table

	sales	quarters
0	100	Q1
1	150	Q2
2	200	Q3
3	250	Q4
4	200	Q1
5	150	Q2

New table

	sales
0	NaN
1	NaN
2	450
3	600
4	650
5	600



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  $(\text{later\_row} - \text{earlier\_row}) / \text{earlier\_row}$ :

- 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.



```
'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(numeric_only=True).sort_index()
```

This query is similar to what we've done before: We want to invoke `mean` on every column in `df`, grouping the results by year. We'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 I passed `numeric_only=True`, thus removing any of the non-numeric columns. In previous versions of Pandas, non-numeric columns were silently ignored. Now, however, we need to either explicitly choose numeric columns, or ask `mean` to do it for us with this keyword argument.

Moreover, because we 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? We'll use `pct_change`, described in the above "Window functions" sidebar.

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,  $\tau$ , 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.

**Figure 6.18. Example of using  $\tau$  to transpose a data frame**



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

We 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.

**Figure 6.19. Get the mean after transposing**

New table

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

↓  
mean

New table

162	428	263	490	334
583	530	553	569	626

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()
```



#### Note

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

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

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

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

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 descending order. Then we can invoke `head()` to see the top-ranking income brackets:

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

All 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  $\leq 0$ :

```
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']) #1

df.columns = ['Year', 'State.Code', 'Total.Math',
    'income<20k',
    '20k<income<40k',
    '40k<income<60k',
    '60k<income<80k',
    '80k<income<100k',
    'income>100k',
    ] #2

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

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

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

```
change = df.groupby('Year')[['income<20k',
                              '20k<income<40k',
                              '40k<income<60k',
                              '60k<income<80k',
                              '80k<income<100k',
                              'income>100k']].mean().T.pct_change() #6
```

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

You can explore a version of this in the Pandas Tutor at:

```
https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A
numpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%
20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%2
ingIO%28' '%0A,Year,State.Code,Total.Math,income%3C20k,20k%3Cinco
40k%3Cincome%3C60k,60k%3Cincome%3C80k,80k%3Cincome%3C100k,income%
162,2008,CO,572,555,567,571,578,533,583%5Cn428,2013,GA,488,451,47
426,530%5Cn263,2010,AZ,527,491,506,523,531,472,553%5Cn490,2014,ME
520,535,555,468,569%5Cn334,2011,MI,605,549,565,590,609,528,626%5C
MT,558,545,545,559,548,521,568%5Cn90,2006,PA,500,467,492,503,521,
n210,2009,AL,554,533,560,565,585,468,596%5Cn319,2011,DE,491,453,4
,417,538%5Cn480,2014,FL,485,465,485,497,510,426,531%5Cn443,2013,M
552,567,574,498,605%5Cn96,2006,TX,506,473,501,514,531,438,559%5Cn
TX,487,465,483,497,515,432,543%5Cn330,2011,LA,553,488,525,539,538
5Cn64,2006,ID,545,529,531,546,551,530,568%5Cn338,2011,MT,537,528,
532,504,557%5Cn538,2015,IN,501,463,487,500,512,436,538%5Cn75,2006
560,583,587,601,544,616%5Cn505,2014,ND,619,588,537,548,635,520,48
2011,MS,546,497,530,547,569,427,576%5Cn' '%29%0A%0Adf%20%3D%20pd.
28data%29%0Adf&d=2023-02-14&lang=py&v=v1
```

### 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?

#### 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 of 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.

Here are some examples of how we might use filter in real-world data sets:

- 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.

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 (i.e., that an operation on a series and a scalar is repeated on each element of the series), we're guaranteed to get a result of the correct dimensions. We can then replace the original score column with our transformed column:

```
df['score'] = df.groupby('year')['score'].transform(lambda x: x/1
```

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.

Here are some ways that we might use `transform` with real-world data sets:

- 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



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 Celsius.
- 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

In this exercise, we used `filter` and `transform` on `DataFrameGroupBy` objects in order to selectively filter and transform rows according to their aggregate properties.

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) #1
```

```

df = pd.concat(all_dfs) #2

def has_multiple_readings_at_least(df):
    return df['precipMM'][df['precipMM'] >= 15].count() > 3 #3

output = df.groupby(['city', 'state']).filter(
    has_multiple_readings_at_least)[
    ['city', 'state', 'precipMM']] #4
output.set_index(['city', 'state']).index.unique() #5

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

output = df.groupby(['city', 'state']).filter(
    has_multiple_readings_at_least,
    min_mm=10, times=3)[['city', 'state', 'precipMM']] #7
output.set_index(['city', 'state']).index.unique() #8

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

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

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

```

You can explore a version of this in the Pandas Tutor at:

[https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0Anumpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%20ringIO%28''%0A,max\\_temp,min\\_temp,precipMM,city,state,precip\\_pct%20-6,0.0,Springfield,IL,0.0%5Cn117,2,-4,0.0,Springfield,MA,0.0%5Cn40.0,San%20Francisco,CA,0.0%5Cn582,0,-5,0.0,Chicago,IL,0.0%5Cn528,San%20Francisco,CA,0.002145922746781116%5Cn77,7,-3,0.0,SpringfielCn35,4,1,0.0,Chicago,IL,0.0%5Cn491,1,-6,0.2,Springfield,IL,0.000684963%5Cn483,-3,-7,0.0,Springfield,MA,0.0%5Cn702,3,-9,0.0,New%20Y%5Cn349,5,-3,0.0,Boston,MA,0.0%5Cn721,5,-2,0.0,Springfield,MA,0.0,-6,0.0,Boston,MA,0.0%5Cn265,0,-3,0.6,Chicago,IL,0.002188183807431,-2,0.0,Chicago,IL,0.0%5Cn201,6,3,0.0,Boston,MA,0.0%5Cn647,1,-11Albany,NY,0.0%5Cn489,10,7,1.4,San%20Francisco,CA,0.003338102050542,13,8,0.0,San%20Francisco,CA,0.0%5Cn137,7,-1,2.3,Albany,NY,0.00558143%5Cn716,3,-1,0.0,Chicago,IL,0.0%5Cn287,-2,-10,0.0,Albany,NY,9,2,0.0,Springfield,IL,0.0%5Cn195,3,-4,0.0,Springfield,MA,0.0%5Cn.0,Springfield,IL,0.0%5Cn466,2,1,5.5,Boston,MA,0.01315160210425636,1,0.0,New%20York,NY,0.0%5Cn490,1,-6,0.0,Springfield,IL,0.0%5Cn1](https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0Anumpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%20ringIO%28''%0A,max_temp,min_temp,precipMM,city,state,precip_pct%20-6,0.0,Springfield,IL,0.0%5Cn117,2,-4,0.0,Springfield,MA,0.0%5Cn40.0,San%20Francisco,CA,0.0%5Cn582,0,-5,0.0,Chicago,IL,0.0%5Cn528,San%20Francisco,CA,0.002145922746781116%5Cn77,7,-3,0.0,SpringfielCn35,4,1,0.0,Chicago,IL,0.0%5Cn491,1,-6,0.2,Springfield,IL,0.000684963%5Cn483,-3,-7,0.0,Springfield,MA,0.0%5Cn702,3,-9,0.0,New%20Y%5Cn349,5,-3,0.0,Boston,MA,0.0%5Cn721,5,-2,0.0,Springfield,MA,0.0,-6,0.0,Boston,MA,0.0%5Cn265,0,-3,0.6,Chicago,IL,0.002188183807431,-2,0.0,Chicago,IL,0.0%5Cn201,6,3,0.0,Boston,MA,0.0%5Cn647,1,-11Albany,NY,0.0%5Cn489,10,7,1.4,San%20Francisco,CA,0.003338102050542,13,8,0.0,San%20Francisco,CA,0.0%5Cn137,7,-1,2.3,Albany,NY,0.00558143%5Cn716,3,-1,0.0,Chicago,IL,0.0%5Cn287,-2,-10,0.0,Albany,NY,9,2,0.0,Springfield,IL,0.0%5Cn195,3,-4,0.0,Springfield,MA,0.0%5Cn.0,Springfield,IL,0.0%5Cn466,2,1,5.5,Boston,MA,0.01315160210425636,1,0.0,New%20York,NY,0.0%5Cn490,1,-6,0.0,Springfield,IL,0.0%5Cn1)

```
.1, Chicago, IL, 0.0003646973012399709%5Cn587, 4, -1, 0.0, Boston, MA, 0.0  
-1, 0.0, Springfield, MA, 0.0%5Cn88, 12, 1, 0.0, Springfield, MA, 0.0%5Cn63  
, Springfield, IL, 0.0%5Cn0, 1, -7, 0.0, Springfield, MA, 0.0%5Cn15, 15, 11,  
Francisco, CA, 0.0%5Cn404, -1, -5, 0.0, Boston, MA, 0.0%5Cn726, 5, -2, 0.0, S  
, MA, 0.0%5Cn618, -3, -10, 0.0, Albany, NY, 0.0%5Cn317, -5, -9, 0.2, Albany, N  
62293618920125%5Cn267, 16, 9, 0.0, Los%20Angeles, CA, 0.0%5Cn%0A'' '%29%  
%20pd.read_csv%28data%29%0A%0Adef%20has_multiple_readings_at_leas  
A%0A%20%20%20%20return%20df%5B'precipMM'%5D%5Bdf%5B'precipMM'%5D%  
%5D.count%28%29%20%3E%203%0A%20%20%20%20%0Aoutput%20%3D%20df.grou  
city',%20'state'%5D%29.filter%28has_multiple_readings_at_least%29  
city',%20'state',%20'precipMM'%5D%5D%0A&d=2023-02-14&lang=py&v=v1
```

### 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')
```

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

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

This data frame isn't 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')
```

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

```
LOCATION      TIME      Value
AUS          2008      31159.8
AUS          2009      29980.7
AUS          2010      35165.5
AUS          2011      38710.1
```

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()
```

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

country	Value
Australia	37634.433333333334
Austria	16673.886363636364
Belgium	16525.237545454547
Brazil	13942.913958333333
Canada	32593.6125

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'])
```

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

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

As soon as 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()
```

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

country	points
Albania	88.0
Argentina	85.9960930562955
Australia	87.89247528747227
Austria	89.27674190382729
Bos + Herz	84.75

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. Here are the first five rows from my result:

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

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.

Here's what the first five rows looked like after performing the above join:

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



## 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 known 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.

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

country	points	Value
Albania	88.0	NaN
Argentina	85.9960930562955	NaN
Australia	87.89247528747227	37634.433333333334
Austria	89.27674190382729	16673.886363636364
Belgium	NaN	16525.237545454547

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](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()
```

You can explore a version of this in the Pandas Tutor at:

<https://pandastutor.com/vis.html#code=import%20numpy%20as%20np%0A>

```

numpy%20import%20nan%0Aimport%20pandas%20as%20pd%0Afrom%20pandas%
20Series,%20DataFrame%0Afrom%20io%20import%20StringIO%0A%0Adata%2
ingIO%28'' '%0Acountry,points,Value%5CnAlbania,88.0,%5CnArgentina,
562955,%5CnAustralia,87.89247528747227,37634.433333333334%5CnAust
74190382729,16673.886363636364%5CnBelgium,,16525.237545454547%5Cn
nd%20Herzegovina,84.75,%5CnBrazil,83.24,13942.913958333333%5CnBul
6753246753246,%5CnCanada,88.23979591836735,32593.6125%5CnChile,86
82668,%5CnChina,82.0,%5CnCroatia,86.28089887640449,%5CnCyprus,85.
3549,%5CnCzech%20Republic,85.83333333333333,%5CnDenmark,,10362.56
%5CnEgypt,83.66666666666667,%5CnEngland,92.88888888888889,%5CnFin
.658590909091%5CnFrance,88.92586975068727,58228.804000000004%5CnG
51162790697674,%5CnGermany,88.62642740619903,75011.82309090909%5C
.11764705882354,%5CnHungary,87.32900432900433,5108.871590909091%5
.625,%5CnIsrael,87.17619047619047,6634.454041666667%5CnItaly,88.4
432,39539.56%5CnJapan,85.0,28606.891666666666%5CnKorea,,21677.131
CnLebanon,85.70270270270271,%5CnLithuania,84.25,%5CnLuxembourg,87
edonia,84.8125,%5CnMexico,84.76190476190476,%5CnMoldova,84.718309
5CnMontenegro,82.0,%5CnMorocco,88.16666666666667,%5CnNew%20Zealan
686746988,%5CnPortugal,88.05768508079669,%5CnRomania,84.920863309
Serbia,87.71428571428571,%5CnSlovakia,83.66666666666667,%5CnSlove
04255319149,%5CnSouth%20Africa,87.22542072630647,%5CnSouth%20Kore
nSpain,86.64658925979681,%5CnSwitzerland,87.25,%5CnTunisia,86.0,%
88.09615384615384,%5CnUS,87.81878936487331,%5CnUS-France,88.0,%5C
84.6,%5CnUnited%20Kingdom,,63507.15909090909%5CnUnited%20States,,
333333334%5CnUruguay,84.47826086956522,%5Cn%0A'' '%29%0Adf%20%3D%2
csv%28data%29%0A%0Adf.dropna%28%29&d=2023-02-14&lang=py&v=v1

```

### 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.

# 7 Midway project

Congratulations! You've made it halfway through this book. If you've truly been doing the exercises, then I hope that you've found your pandas skills improving, little by little. (And if you opened up to this chapter without doing the exercises first? Shame on you!)

Are you forgetting some of the syntax, method names, and parameter names? Are you making frustrating, "stupid" mistakes? That's only natural, and it happens to everyone, no matter how long they've been using pandas, or any other large software library. Over time, though, it'll become more natural and more obvious, at least when using the functionality that's most common in your work.

The whole point of this book is to gain experience and fluency through practice. Such gains happen incrementally, and over time. But they do happen, even if it doesn't always feel that way.

In this chapter, we're taking a break from the exercises that concentrate on particular topics and themes. Instead, I'm going to ask you to do a small project, one that'll require you use many of the parts of pandas that you've learned to use over the last few chapters. I hope that this project gives you a chance to integrate the different skills you've learned so far.

We'll look at data from the 2020 Python Developer Survey, alongside the 2021 survey from Stack Overflow. The Python survey, which is run by JetBrains (the company behind the popular PyCharm editor for Python, among others), is our best snapshot of the global Python community—who they are, and what they do. Separately, the well-known programming Q&A site Stack Overflow runs an annual survey of programmers of all types, including those using Python.

## 7.1 Problem

Here is what I'd like you to do:

- Load the CSV file with results from the Python survey into a data frame. Let's call that `py_df`.
- Turn the columns into a multi-index. How you do this depends on the column:
  - Most of the columns have the form `first.second.third`, with two or more words separated by `.` characters. Divide the column name into two parts, one before the final `.` and one after. The multi-index column for this example would then be (`'first.second'`, `'third'`). If there were only two parts, it would be (`'first'`, `'second'`).
  - In the case of about 20 columns, the top level should be general, and the second level should be the original column name. The columns you should treat this way are:
    - `age`,
    - `are.you.datascientist`,
    - `is.python.main`,
    - `company.size`,
    - `country.live`,
    - `employment.status`,
    - `first.learn.about.main.ide`,
    - `how.often.use.main.ide`,
    - `is.python.main`,
    - `main.purposes`
    - `missing.features.main.ide`
    - `nps.main.ide`,
    - `python.version.most`,
    - `python.years`,
    - `python2.version.most`,
    - `python3.version.most`,
    - `several.projects`,
    - `team.size`,
    - `use.python.most`,
    - `years.of.coding`
  - Use the function `pd.MultiIndex.from_tuples` to create the multi-

index, and then reassign it back to `df.columns`. (Hint: A function, along with a Python `for` loop or list comprehension, will come in handy here.)

- Sort the columns, such that they're in alphabetical order. (This isn't technically necessary, but it makes the data easier to see and understand.)
- Which 10 IDEs were most commonly listed as the main editor (i.e., in the `ide` column)?
- Which 10 other programming languages (other `.lang`) are most commonly used by Python developers?
- According to the Python survey, what proportion of Python developers have each level of experience?
- Which country has the greatest number of Python developers with 11+ years of experience?
- Which country has the greatest **proportion** of Python developers with 11+ years of experience?
- Now load the Stack Overflow data into a data frame. Let's call that `so_df`.
- Show the average salary for different types of employment. Contractors and freelancers like to say that they earn more than full-time employees. What does the data here show us?
- Create a pivot table in which the index contains countries, the columns are education levels, and the cells contain the average salary for each education level per country.
- Create this pivot table again, only including countries in our OECD subset. In which of these countries does someone with an associate degree earn the most? In which of them does someone with a doctoral degree earn the most?
- Remove rows from `so_df` in which `LanguageHaveWorkedWith` is `NaN`.
- Remove rows from `so_df` in which Python isn't included as a commonly used language (`LanguageHaveWorkedWith`). How many rows remain?
- Remove rows from `so_df` in which `YearsCode` is `NaN`. How many rows remain?
- Replace the string value `Less than 1 year` in `YearsCode` with `0`. Replace the string value `More than 50 years` with `51`.
- Turn `YearsCode` into an integer column.

- Create a new column in `so_df`, called `experience`, which will categorize the values in the `YearsCode`. Values can be:
  - Less than 1 year
  - 1–2 years
  - 3–5 years
  - 6–10 years
  - 11+ years
- According to the Stack Overflow survey, what proportion of Python developers have each level of experience? # Create a data frame in which the index contains the level of experience, and the two columns are the proportion of Python developers with that level from each of the two surveys. Would you say that the Stack Overflow respondents are less experienced, more experienced, or the same as those in the Python community survey?

### 7.1.1 Discussion

This project was all about understanding the world of Python developers a bit better, using data from two different surveys. There are hundreds, if not thousands, of other questions that you could ask (and answer) using this data; if you find this project of interest, then I encourage you to continue the analysis on your own.

We started off by loading the data from the Python community survey into a data frame. On the face of it, this shouldn't be too hard:

```
py_filename = '../data/2020_sharing_data_outside.csv'  
py_df = pd.read_csv(py_filename)
```

If you load the data in this way, you'll likely get a warning from pandas indicating that some columns had mixed types. We've seen this problem before; the issue is that pandas does a good job of guessing a column's dtype, but that consumes a great deal of memory. The warning tells us that we can either explicitly specify the dtypes of our columns in our call to `pd.read_csv`, or (if we have sufficient memory) we can let pandas read all of the data in and guess. The fact is that we won't be using very many columns in our analysis for this project, and thus the real-life, practical solution would

be to specify which columns we want to load with `usecols`. However, I want you to get some practice creating a multi-index, and also have the data available for further exploration after this project is complete. Thus, we'll read all of the data in, and tell pandas to use as much memory as it needs in order to guess the dtype correctly:

```
py_filename = '../data/2020_sharing_data_outside.csv'  
py_df = pd.read_csv(py_filename, low_memory=False)
```



#### Note

I'm assuming that your computer has enough memory to load all of the columns. If not, then you should indeed pass `usecols` to `read_csv`, specifying only the columns that we'll be using in this exercise. That will reduce the memory usage enough to let pandas guess correctly without overburdening your computer.

There's nothing technically wrong with using the data frame as we have loaded and defined it. But as loaded, it contains 264 (!) columns, too many for most people to understand and think about. Moreover, while a CSV file cannot have hierarchical column names, the names were clearly designed to give us a sense of hierarchy. For example, we have `other.lang.Java`, `other.lang.JavaScript`, `other.lang.C/C++`, and so forth—all of which could really fit under an `other.lang` category. Can we take a flat list of columns, and turn it into a multi-index, thus making it easier to think about and also to work with?

The answer is "yes," but it'll take a little bit of work. The first thing to notice (which I also mentioned in the instructions for this exercise) is that each column name is of the type `first.second.third`. If we break the column name apart at the final `.` character, we can create a multi-index in which the column becomes `first.second` and then `third`. In other words, we would have a top-level multi-index column of `other.lang`, with second-level columns of `Java`, `JavaScript`, `C/C++`, and so on.

This sounds very nice, but how can we do it? How can we create a multi-index, and then apply it to our data frame?

We can use `pd.MultiIndex.from_tuples`, a function that pandas provides for precisely this purpose. If we pass a list of tuples to this function, it returns a multi-index object, one which we can then assign to a data frame's `index` or `columns` attribute, as appropriate. In our case, we'll want to assign it to `columns`, replacing the existing index object used on the columns.

That's nice, but we'll first need to create the list of tuples. Each tuple's first element will be all of the text up to the final `.` in the column name, and the second element will be the word following that final `.`. We can do this using Python's `str.rsplit` method, which works similarly to `str.split`, but works from the right, rather than from the left. By itself, `str.rsplit` won't make a difference. But if we pass a second, integer argument of `1`, it'll return a list of two elements, split from the final `.`:

```
s = 'abcd.efgh.ijkl'
s.rsplit('.', 1)
```

The above code will return `['abcd.efgh', 'ijkl']`, perfect for our purposes. (Except that it's a list, not a tuple.)

But wait, it gets even more complex: For a bunch of otherwise uncategorized column names, I indicated that we'll give it a top-level column of `general`. I'll define those columns in a list:

```
general_columns = ['age',
                   'are.you.datascientist',
                   'is.python.main',
                   'company.size',
                   'country.live',
                   'employment.status',
                   'first.learn.about.main.ide',
                   'how.often.use.main.ide',
                   'is.python.main',
                   'main.purposes',
                   'missing.features.main.ide',
                   'nps.main.ide',
                   'python.version.most',
                   'python.years',
                   'python2.version.most',
                   'python3.version.most',
                   'several.projects',
                   'team.size',
```

```
        'use.python.most',
        'years.of.coding'
    ]
```

In order for all of this to work, I think that it'll be easiest to write a function, `column_multi_name`, which takes a single column name (i.e., a string). If the column name is one of those which will get a general top-level column, then we'll return a two-element tuple containing `general` and then the existing column name. But in all other cases, we'll return a two-element tuple based on a list we get back from `str.rsplit`. Here's how the function could look:

```
def column_multi_name(column_name):
    if column_name in general_columns:
        return ('general', column_name)
    else:
        first, rest = column_name.rsplit('.', 1)
        return (first, rest)
```

What do I want to do with this function? Invoke it on each of the column names in `py_df`. Then I want to take all of those results, and pass them as a list to `pd.MultiIndex.from_tuples`. To create such a list, I'll use a list comprehension:

```
pd.MultiIndex.from_tuples([column_multi_name(one_column_name)
                           for one_column_name in py_df.columns    ])
```

I'll then assign the result of this function call back to `py_df.columns`, replacing the original columns with my multi-index:

```
py_df.columns = pd.MultiIndex.from_tuples([column_multi_name(one_
                                           for one_column_name in py_df.columns    ])
```

Most of the time, it doesn't matter whether your columns are sorted. But I've found that when working with a multi-index, it's often best to sort the column names, if only to make it easier to skim through them. In order to do this, we take advantage of the fact that we can pass a list of columns to `py_df` to get those columns back. If we sort the list before we apply it, then we can get the columns back in a particular order. Assigning that back to `py_df` will thus sort the columns:

```
py_df = py_df[sorted(py_df.columns)]
```

Whew! I'll admit that this involved a lot of steps, some of which used builtin Python (i.e., not pandas) knowledge. But now we can think about and work with our columns in a much smarter way.

Let's start off by finding out what IDEs Python developers use most often. We can get that from the `ide` top-level column, and the `main` second-level column. (This is a multi-index column, remember.) We can retrieve this by passing a tuple in the square brackets:

```
py_df[('ide', 'main')]
```

Now that we have our column, we can count how often each of the IDEs appears in the survey. And because `value_counts` returns a series, we can limit the output to the 10 most popular IDEs:

```
py_df[('ide', 'main')].value_counts().head(10)
```

Next, I asked you to find what other languages are most commonly used by Python developers. This requires figuring out which columns contain this information, how it is structured, and then how to perform such a calculation. As you might have discovered, non-Python languages were listed under the `other.lang` top-level index, with the particular language that each developer used as a second-level index entry. Thus, asking for `py_df['other.lang']` returns all of the columns under the `other.lang` top-level index as a data frame—one with 54,462 rows (one for each survey respondent) and 24 columns (one for each non-Python language). Each cell either contains the name of the language or `NaN` (indicating that the survey respondent does not use this language). With this data, how can we calculate how many people use each of these languages?

The answer is easier than it might at first appear: The `count` method returns the number of non-`NaN` values in a series. When applied to a data frame, the `count` method returns a series whose indexes are the data frame's columns, and whose values are the number of non-null values in that column. In other words, we can run:

```
py_df['other.lang'].count()
```

The result will be the following series:

Bash / Shell	13793
C#	4460
C/C++	11623
Clojure	361
CoffeeScript	319
Go	3398
Groovy	719
HTML/CSS	15469
Java	8109
JavaScript	16662
Kotlin	1384
None	6402
Objective-C	583
Other	3592
PHP	4060
Perl	886
R	2465
Ruby	1165
Rust	1853
SQL	13391
Scala	927
Swift	854
TypeScript	3717
Visual Basic	1604

dtype: int64

Now that we have this series, we can find the 10 most popular non-Python languages. We do this by sorting the values in descending order:

```
py_df['other.lang'].count().sort_values(ascending=False)
```

Finally, we can get the 10 first values:

```
py_df['other.lang'].count().sort_values(ascending=False).head(10)
```

The next question asked for the names of the 10 countries with the greatest number of survey respondents. That information is in the general top-level index, and the country.live second-level index. We can retrieve that specific column with:

```
py_df['general', 'country.live']
```

This returns the country name for each of the survey respondents. To count the number of times each country appears in this column, we can use

`value_counts`:

```
py_df['general', 'country.live'].value_counts()
```

The result of running `value_counts` is to return a series, one in which country names form the index and the number of appearances of each country form the values. An added bonus is that `value_counts` automatically sorts its results by descending value, so we don't need to worry about that. Finally, we can use `head(10)` to retrieve the 10 most commonly named countries in the survey:

```
py_df['general', 'country.live'].value_counts().head(10)
```

Next, I asked you to find what proportion of Python developers have each level of experience. Once again, we can turn to `value_counts` to do this work for us. Normally, `value_counts` returns the raw numbers. However, passing `normalize=True` to `value_counts` outputs floating-point values, indicating the percentage for each level:

```
py_df[('general', 'python.years')].value_counts(normalize=True)
```

Since `value_counts` automatically orders the values from highest to lowest, we can see that in this survey, the greatest proportion of developers have 3-5 years of experience, followed by those with less than 1 year, followed by those with between 1-2 years. All told, about 75 percent of the respondents to the survey have been using Python for up to five years, and half of them have been using it for less than two years.

What about the most experienced Python developers? In particular, what countries have the greatest number of Python developers with 11+ years of experience using the language? To do that, we'll first need to get only those rows of `py_df` in which the experience is 11+ years:

```
py_df[py_df[('general', 'python.years')] == '11+ years']
```

But wait: We're going to want to group by `country.live`, whose top-level index is `general`—the same as `python.years`. We can thus restrict our query, applying our boolean index only to those columns within `general`:

```
py_df['general'][py_df[('general', 'python.years')] == '11+ years']
```

Now that we only have the columns in `general`, we can prepare a new query, one that'll give us results on a per-country basis:

```
py_df['general'][py_df[('general', 'python.years')] == '11+ years']
```

This sets up the grouping query to operate on a per-country basis, but doesn't actually ask any questions. Let's find out how many non-null values each column has, for each country:

```
py_df['general'][py_df[('general', 'python.years')] == '11+ years']
```

The resulting data frame is a bit big and daunting, but gives us a ton of information: The rows are the country names, the columns are all from `general`, and the values are integers, indicating how many non-null rows there were for each column, for each country. The numbers in each column will be similar, but not identical, reflecting the fact that there will be some occasional null values. But given that we're only interested in finding out how many super-experienced Python developers there are in each country, we would be wise to cut our result down to one column only, namely `python.years`:

```
py_df['general'][py_df[('general', 'python.years')] == '11+ years']
```

This returns a series in which the index contains country names, and the values are integers, the number of 11+ year veterans of Python. We're not quite done yet, though—we want to know which country has the greatest number of very experienced Python developers. It might seem like we could use the `max` method here, but that'll return the highest value—and we want to know the index (i.e., country name) corresponding to that value. For this reason, we'll once again call `sort_values`, from highest to lowest. Then we'll apply `head(1)`, returning the name of the country with the greatest number, as well as the number itself:

```
py_df['general'][py_df[('general', 'python.years')] == '11+ years']
```

You might not be surprised to find that the United States tops this list. After all, the US has a large number of Python developers, and is also a large

country. But that's often a problem with finding "the greatest number per country" for anything, since it depends so greatly on the size of the country. So perhaps a more interesting question would be: Which country has the greatest proportion of Python developers with 11+ years of experience?

To get this calculation, we'll need to find out how many total developers there are in each country. To do that, I created a new variable called `country_experience`, taken from `py_df['general']` and consisting of two columns—`country.live` and `python.years`:

```
country_experience = py_df['general'][['country.live', 'python.years']]
all_per_country = country_experience['country.live'].value_counts
```

We'll also need to get the number of senior Python developers in each country. We did that in a previous part of this exercise, but with `country_experience` in place, I have another method for finding this out:

```
expert_per_country = country_experience.loc[country_experience['python.years'] > 10].value_counts
```

We now have two series (`expert_per_country` and `all_per_country`) with matching indexes (country names). We can now take advantage of the fact that pandas will use the index when dividing one series by another:

```
(expert_per_country / all_per_country).sort_values(ascending=False)
```

In the above calculation, I first divided the number of experts by the total number of Python developers per country. I then sorted the values in descending order, so that we could find the country with the greatest proportions of experienced Python developers. In order to avoid null values, I used `dropna` on the resulting series. The result looked quite different:

Norway	0.265432
Ireland	0.225490
Australia	0.225420
Belgium	0.225108
Slovenia	0.224490
New Zealand	0.197917
Sweden	0.194030
Finland	0.190141
United Kingdom	0.186486
Austria	0.186170

```
Name: country.live, dtype: float64
```

While the United States certainly has a very large number of senior Python developers, things look dramatically different when we take country size into account.

However, this raises the question of whether the data is an accurate portrait of the modern Python community. Do one quarter of Norwegian Python developers really have more than a decade of experience with the language? Maybe it's just me, but I'm a bit skeptical. Instead, I have to wonder whether the type of person who fills out such a survey is also more enthusiastic than the average Python developer—and thus skews to a more experienced population.

Next, we switched gears, and started to look at the Stack Overflow survey. I loaded the CSV file into a data frame:

```
so_filename = '../data/so_2021_survey_results.csv'  
so_df = pd.read_csv(so_filename, low_memory=False)
```

Once again, I passed `low_memory=False`, telling pandas that it should use as much memory as it needs in order to guess the dtype correctly.

The Stack Overflow survey includes a great deal of information about people's jobs and salaries. I asked you to verify if, based on the data collected here, freelancers and contractors really do earn more than full-time employees, as is often assumed to be the case. In order to find this out, I took my data frame and ran `groupby` on `Employment`:

```
so_df.groupby('Employment')
```

This means that whatever query we run, the rows will be the distinct values in the `Employment` column. We're interested in the mean annual salary, reported here in dollars as `ConvertedCompYearly`, per type of employment, which we can calculate as follows:

```
so_df.groupby('Employment')['ConvertedCompYearly'].mean()
```

This is good, but we can make it easier to compare the data points by sorting

them:

```
so_df.groupby('Employment')['ConvertedCompYearly'].mean().sort_va
```

We can see, from these results, that from this survey, it would seem that people who are employed full time earn the most, followed by retirees, followed by contractors:

```
Employed full-time          129913.09
Retired                    120252.50
Independent contractor, freelancer, or self-employed  111160.26
I prefer not to say        44589.43
Employed part-time         43344.53
Not employed, and not looking for work
Not employed, but looking for work
Student, full-time
Student, part-time
Name: ConvertedCompYearly, dtype: float64
```

I find this a bit hard to believe, and especially wonder whether it's accurate to say that retirees are earning almost as much as full-time employees.

While it's not part of the exercise, I'd like to spend a bit more time on this output. There's nothing technically wrong with it, but if we were to present this data to a non-programmer, it might seem a bit messy, or hard to read. For one, we might want to remove the NaN values. For another, dollar figures can generally be rounded to two digits after the decimal point. And it's often nice to put separators between every set of three digits in a large number. Dropping the null values is easy with `dropna`. But is there a good way to format our floating-point values?

The answer is "yes," and there are a few possible solutions. I prefer to use Python's f-strings, which include a great deal of formatting logic and power.

This functionality is provided by Python's f-strings, which let us apply a number of formatting directives to strings, and other data structures, following a `:` (colon). For example:

```
x = 12345.6789
print(f'{x:,.2f}') #1
```

I can't use an f-string directly on each element of a series. But I can use a function to do it for me. In particular, I can use `apply` to run a function on each element, and then use `lambda` to create an anonymous function that applies the f-string to each one, thus giving me a column of strings:

```
so_df.groupby('Employment')['ConvertedCompYearly'].mean().sort_va
```

Many Python developers dislike using `lambda`. How, then, could we ensure that our floats are formatted nicely? One alternative to the above would be to use the `str.format` method, which was commonly used before f-strings were introduced in Python 3.8. Because `str.format` is a method, we can hand it to `apply`, without needing to rely on `lambda`:

```
so_df.groupby('Employment')['ConvertedCompYearly'].mean().sort_va
```

Notice that we're not invoking the method, but rather passing it to `apply`, where it'll be invoked on each value. Moreover, notice that there isn't any value preceding the `:` in the curly braces; that's because `str.format` was able to implicitly handle positional arguments, without naming or numbering them.

If we know that we'll want to display all floats this way, then we can avoid using `apply` explicitly, and tell pandas that we always want this to be our display format:

```
pd.options.display.float_format = '{:,.2f}'.format
```

The above basically does what we did with `apply`, telling pandas that whenever it sees a floating-point value, it should output the format with commas and only two numbers after the decimal point. However, this now applies across the board to all floats in the current session, which might be more than you want. That said, I'll use it so as to avoid needing to set formatting.

Now let's ask a different question: Rather than looking at average salaries for different types of work, let's instead look at average salaries for different levels of education. Moreover, let's further divide that up by country. What I'm asking for, of course, is a pivot table—one in which the index will contain country names, the columns will contain the distinct values from

EdLevel, and the cells will contain the mean of ConvertedCompYearly for each country-education combination:

```
so_df.pivot_table(index='Country', columns='EdLevel', values='Con
```

Next, I asked you to load the subset of OECD countries into a data frame:

```
oeed_filename = '../data/oeed_locations.csv'  
oeed_df = pd.read_csv(oeed_filename, header=None, index_col=1, na
```

The data frame we created in the above code used the country name for the index. That's because I'm next going to use it in a join with so\_df, which means that the indexes need to be aligned. We'll use the country names, and assume (hope?) that the spellings and punctuation of our OECD subset are all the same in both data frames. Regardless, we'll need to have the country name as the index.

Then we want to join our OECD subset data frame with the Stack Overflow data, and then recreate our pivot table. The effect will be to reduce the number of rows (i.e., countries) in our output. And indeed, once we run our join, we get back only 13 rows, one for each country in our OECD subset:

```
oeed_df.join(so_df.set_index('Country')).pivot_table(index='Count  
                                                    columns='EdL  
                                                    values='Conv
```

Notice that we called so\_df.set\_index('Country') so as to temporarily set the country to be the index of so\_df. That allows us to join it with oeed\_df—and then to create the pivot table, which is our ultimate goal.

Now that we know average salaries in all of these countries and all education levels, we can ask some questions of the data. For example, I asked you to find in which country someone with an associate's degree can expect to earn the most. I could have stored the pivot table to a variable, but instead I decided to chain the relevant methods together:

```
oeed_df.join(so_df.set_index('Country')).pivot_table(index='Count  
                                                    columns='EdL  
                                                    values='Conv  
            'Associate degree (A.A., A.S., etc.)'].sort_values(ascendin
```

After creating the pivot table, I then retrieved the column for associate's degrees, and then sorted them from highest to lowest. From the results we see here, it looks like the country that offers the best pay for people with an associate's degree is Australia, followed by Germany and Israel.

What about PhDs? Do countries that pay you well with an associate's degree also pay you well if you have a PhD or similar post-graduate degree? We can perform a similar query:

```
oe.cd_df.join(so_df.set_index('Country')).pivot_table(index='Count
                                                    columns='EdL
                                                    values='Conv
                                                    'Other doctoral degree (Ph.D., Ed.D., etc.)'].sort_values(a
```

There does seem to be some overlap; the highest-paying countries for PhDs are Japan, Australia, France, Israel, and Germany. But there might also be some reason to suspect that this data isn't totally accurate; is it really possible that the mean salary in Hungary for someone with an associate's degree is \$63k/year, whereas with a PhD it's only \$52k/year? Or that there would be a salary difference of only \$12k/year between Germans with an associate's degree and a PhD? It's certainly possible, but my point here is that data analysis requires more than just number crunching—you also have to ask whether the numbers make sense. And if they don't, we should ask ourselves why that might be the case. For example, perhaps the sample sizes are so small that the data isn't truly representative of the total population.

Next, I wanted to analyze Python programmers in the Stack Overflow survey. The `LanguageHaveWorkedWith` column will allow us to identify who they are—but that column contains text, with languages separated from one another with `;` characters. So someone who works with both Python and JavaScript could have a value of `Python;JavaScript`. If we're going to find people who work with Python, then we'll need to find those who have "Python" in that column. For that, we'll want to use `str.contains`, to look inside of the string. But there's a problem with that: Some survey respondents didn't fill out this information, which means that it's `NaN`. And trying to run `str.contains` on a `NaN` value will result in an error.

We'll thus need to first remove all of the rows that contain `NaN` for `LanguageHaveWorkedWith`. After that, we'll find people who have worked

with Python:

```
so_df = so_df[~so_df['LanguageHaveWorkedWith'].isna()]
so_df = so_df[so_df['LanguageHaveWorkedWith'].str.contains('Pytho
```

Notice that we couldn't use `dropna`, which would let us remove all NaN values from a column or data frame. That's because we are interested in dropping rows where `LanguageHaveWorkedWith` is null, and `dropna` doesn't let us do that. Instead, we'll use `isna`, which indicates whether a particular cell contains a null value. We can then use that as a boolean index to identify rows with non-null values for this column, which we'll keep.

Once we've done that, we can be sure that all of the values in `LanguageHaveWorkedWith` are strings. We apply `str.contains` and look for Python. We end up with nearly 40,000 people who use Python—a smaller sample than the 54,000 who responded to the Python survey, but still a substantial sample size. Also, while the survey asked what languages people had used in the last year, we don't know whether that they used Python once in the last year year, every day, or somewhere in between.

Now that we have found the Python developers from Stack Overflow, I would like to compare them with the respondents to the Python community survey. In particular, I'd like to know if they have similar levels of experience. The problem is that in the data's original form, that's not going to be possible: Whereas the Python community survey lumps people into categories (e.g., "Less than 1 year," and "1–2 years"), the Stack Overflow survey asks for a specific number of years of experience.

In order to compare these, I asked you to create a new column in the Stack Overflow data frame, called `experience`, which would turn the raw year numbers into categories. I know that I can use `pd.cut` in order to accomplish this, but `pd.cut` will only work if all of the values in a column are numeric—and two options were non-numeric, `Less than 1 year` and `More than 50 years`. My first task was thus to turn those into numbers:

```
so_df.loc[so_df['YearsCode'] == 'Less than 1 year', 'YearsCode']
so_df.loc[so_df['YearsCode'] == 'More than 50 years', 'YearsCode']
```

In other words, I assigned the integer 0 to anyone with `Less than year` as

their value, and the integer 51 to anyone with More than 50 years as their value. With those in place, we can then turn YearsCode into an integer column:

```
so_df['YearsCode'] = so_df['YearsCode'].astype(int)
```

Now I can use `pd.cut` to recreate the same categories as we had in the Python community survey:

```
so_df['experience'] = pd.cut(so_df['YearsCode'],
                             bins=[-1, 1, 2, 5, 10, 100],
                             labels=['Less than 1 year',
                                     '1-2 years',
                                     '3-5 years',
                                     '6-10 years',
                                     '11+ years'])
```

Remember that `pd.cut` uses the bins as the extreme edges of the bins—which means that if we want to give the first label to a number of 0, we should start the bin at -1. And yes, there is the option of including values on the left (or right), but I decided that this was easier, and would ensure that bins didn't overlap.

Next, I wanted to see the distribution of experience levels in the Stack Overflow survey. I once again used `value_counts`:

```
11+ years          0.373388
6-10 years         0.318589
3-5 years          0.222530
1-2 years          0.047440
Less than 1 year   0.038054
Name: experience, dtype: float64
```

From this, we can see that the Stack Overflow respondents are much more experienced than the Python survey respondents. As you may recall, we saw that 75 percent of the Python survey respondents have been using Python for up to five years, whereas in the Stack Overflow survey, the number of new programmers is about 25 percent. Half of the Python survey respondents have been using it for less than two years, whereas that's true for less than 10 percent of the Stack Overflow group.

However, we need to think a bit before we say anything too sweeping when comparing these surveys. After all, the Stack Overflow survey was asking about all of the experience that the respondent had as a programmer—whereas the Python survey asked how long the person had been programming in Python. The same person, filling out both surveys, might have been programming in Java for 20 years and Python for only two, and would thus have answered the questions differently on each survey. Making such comparisons, and integrating data from different sources, can be tricky, and requires some thought; just joining two data frames together isn't sufficient. That said, it is interesting to see just how heavily the Python survey skewed toward newcomers, and how heavily Stack Overflow skewed toward experienced developers. The Python community survey might do well to include an "overall programming experience" question in the future, to help with such analysis, and to better understand how much Python plays a role in the members of its community.

## 7.1.2 Solution

```
py_filename = '../data/2020_sharing_data_outside.csv'
py_df = pd.read_csv(py_filename, low_memory=False)

general_columns = ['age',
                  'are.you.datascientist',
                  'is.python.main',
                  'company.size',
                  'country.live',
                  'employment.status',
                  'first.learn.about.main.ide',
                  'how.often.use.main.ide',
                  'is.python.main',
                  'main.purposes',
                  'missing.features.main.ide',
                  'nps.main.ide',
                  'python.version.most',
                  'python.years',
                  'python2.version.most',
                  'python3.version.most',
                  'several.projects',
                  'team.size',
                  'use.python.most',
                  'years.of.coding'
                 ]
```



```
values='Conv
```

```
so_df = so_df[~so_df['LanguageHaveWorkedWith'].isna()]
so_df = so_df[so_df['LanguageHaveWorkedWith'].str.contains('Pytho

so_df.shape

so_df = so_df[~so_df['YearsCode'].isna()]

so_df.shape

so_df.loc[so_df['YearsCode'] == 'Less than 1 year', 'YearsCode']
so_df.loc[so_df['YearsCode'] == 'More than 50 years', 'YearsCode'

so_df['YearsCode'] = so_df['YearsCode'].astype(int)

so_df['experience'] = pd.cut(so_df['YearsCode'],
    bins=[-1, 1, 2, 5, 10, 100],
    labels=['Less than 1 year',
    '1-2 years',
    '3-5 years',
    '6-10 years',
    '11+ years'])

so_df['experience'].value_counts(normalize=True)
```

## 7.2 Summary

Whew! This was a big and long exercise, meant to help you integrate and use many of the ideas and techniques we've discussed in this book so far. Of course, there are many pieces of pandas that we didn't use in this project—but to be honest, it's a rare project that uses all of the capabilities that pandas has to offer. That said, we did a lot of things here—loading and cleaning data, joining data frames together, analyzing the data, and even comparing different data sets and thinking critically about how trustworthy they are. If you felt comfortable with all of the techniques in this project, then I'd say you're well on your way to internalizing the way that pandas does things, and to using it productively in your projects.

# 8 Strings

When most people think of pandas, or of data analysis in general, they think of numbers. And indeed, much of the analysis work that people do with pandas is with numbers. That's why pandas is built on top of NumPy, which takes advantage of C's fast, efficient integers and floats. And that's why so many of the exercises in this book involve working with numbers.

However, we often have to work with textual data—usernames, product names, sales regions, business units, ticker symbols, and company names are just a few examples. Sometimes the text is central to the analysis you're doing—such as when you're preparing data for a text-based machine-learning model—and at other times, it's secondary to the numbers, used as a description or categorical data.

It turns out that pandas is also well equipped to handle text. It does this not by storing string data in NumPy, but rather by using full-fledged string objects, either those that come with Python or (more recently) a pandas-specific string class that reduces both ambiguity and errors. (I'll have more to say about these two string types, and when to use each one, later in the chapter.) In either case, we have access to a wide variety of methods that we can apply to these strings, normally via the `str` accessor.

While a data frame might contain one or more textual columns, the `str` accessor exists on `Series` objects. The result of invoking methods via the `str` accessor is a new series, which can then replace the existing one, be assigned to a new variable, or assigned as a new column alongside the original one.

In this chapter, you'll work through exercises that will help you to identify and understand how to work with textual data and the `str` accessor in pandas. After going through these exercises, you'll know which string methods are available, feel more comfortable using them, and even know how to apply your own custom functions to string columns.

## Text data types

For many years, pandas would use Python's internal string type for storing text. This was already a big improvement over NumPy, which stored characters in C arrays—more efficient than Python strings, but with much more limited functionality. In order to refer to such Python strings, pandas would assign a dtype of object. The good news is that this worked fairly well, giving us great string functionality within pandas. The bad news, however, was that your series could contain **any** type of Python object, not just strings. This could lead to bugs, because you could accidentally store a list, dictionary, or None into such a column without noticing. After all, these are all Python objects, so there was no way for pandas to stop you from adding them.

pandas 1.0.0 added a new `pd.StringDtype`, which aims to solve such problems. As the name indicates, it is meant to be used as a dtype on a series. Moreover, because it's specific to textual data, you cannot mix it up with other types of objects. Further, the pandas documentation indicates that

ut wait—previously, a series with a dtype of object could be a string, and could also be NaN. What happens now? After all, NaN isn't an instance of `pd.StringDtype`. The answer is that if you're going to use `pd.StringDtype`, then you should also use `pd.NA` instead of NaN. `pd.NA` is specifically designed for this sort of thing, ensuring that you can have a null value that's also a valid string.

Should you use `pd.StringDtype`? As of this writing, the pandas documentation is somewhat inconsistent: On the one hand, they list several benefits of `pd.StringDtype`. On the other hand, they write, "`StringDType` is considered experimental. The implementation and parts of the API may change without warning."

In this chapter, I'm going to assume that you are using the old-fashioned (and definitely stable) object type in your columns. However, you will likely need (and want) to switch to `pd.StringDtype` in the future. If all goes well, then such a change will mean no changes to your programs, other than better checking of your values and potentially even better performance.

## 8.1 Useful references

**Table 8.1. What you need to know**

Concept	What is it?	Example	To learn more
<code>s.explode</code>	Returns a new series with each element on its own line	<code>s.explode()</code>	<a href="http://pandas.pydata.org/pandas-docs/stable/10min.html#exploding-a-series">http://pandas.pydata.org/pandas-docs/stable/10min.html#exploding-a-series</a>
<code>str.contains</code>	Returns a series of booleans, whether the row contains the string argument	<code>s.str.contains('a')</code>	<a href="http://pandas.pydata.org/pandas-docs/stable/10min.html#boolean-indexing">http://pandas.pydata.org/pandas-docs/stable/10min.html#boolean-indexing</a>
<code>str.get_dummies</code>	Returns a data frame containing 1s and 0s based on a categorical series	<code>s['country'].get_dummies(sep=';')</code>	<a href="http://pandas.pydata.org/pandas-docs/stable/10min.html#boolean-indexing">http://pandas.pydata.org/pandas-docs/stable/10min.html#boolean-indexing</a>
<code>str.index</code>	Returns a series of integers, the first occurrence of the	<code>s.str.index('a')</code>	<a href="http://pandas.pydata.org/pandas-docs/stable/10min.html#boolean-indexing">http://pandas.pydata.org/pandas-docs/stable/10min.html#boolean-indexing</a>

	argument in each row		
<code>str.len</code>	Returns a series of integers, the length of each element	<code>s.str.len()</code>	<a href="http://r">http://r</a>

**Table 8.2. What you need to know**

Concept	What is it?	Example	To learn more
<code>str.replace</code>	Returns a series based on an existing series, replacing the first argument with the second	<code>s.str.replace('a', 'e')</code>	<a href="http://mng.bz/o2lj">http://mng.bz/o2lj</a>
<code>str.split</code>	Returns a series of Python lists	<code>s.str.split(';')</code>	<a href="http://mng.bz/nNd8">http://mng.bz/nNd8</a>
<code>str.strip</code>	Returns a series of Python strings without the argument's characters on either side	<code>s.str.strip('.!?')</code>	<a href="http://mng.bz/v6Rq">http://mng.bz/v6Rq</a>

## The str accessor

Traditional Python strings support a large number of methods and operators, ranging from search (`str.index`) to replacement (`str.replace`) to substrings (slices) to checks of the string's content (e.g., `str.isdigit` and `str.isspace`). But if we have a series containing strings, how can we invoke such a method on every element?

Experienced Python developers would normally expect to use a `for` loop, or perhaps a list comprehension. But in pandas, we'll do whatever we can to avoid such loops. We do have the option of the `apply` method, which we can use to apply a string method to every element. And indeed, `apply` is needed if you want to use a custom function, rather than a method that comes with pandas.

pandas encourages us to use the `str` accessor, which gives us access to a variety of string methods—including, but not limited to, standard Python string methods. The method will be applied to every element in the series, and will return a new series of the same length and with the same index, whose values are the results of invoking the method on each element. For example, we can get the length of a string by invoking the `len` method on the `str` accessor:

```
s = Series('this is a test 123 456'.split())
s.str.len()
```

The result is a new series, containing the lengths of the values in `s`:

```
0    4
1    2
2    1
3    4
4    3
5    3
dtype: int64
```

What if we want to find all of the values in `s` that can be turned into integers?

```
s.str.isdigit()
```

The result is a boolean series, indicating which values contain only the character 0-9:

```
0    False
1    False
2    False
3    False
4     True
5     True
dtype: bool
```

Because it contains only booleans, and shares an index with `s`, it's suitable for use as a boolean (mask) index on `s`, in order to find numeric values.

The `str` accessor supports methods beyond those available in Python's `str` class. For example, you can search in a string using `contains`. However, `contains` allows you to use a regular expression. I can thus find all of the words with either a or e:

```
s.str.contains('[ae]')
```

The above query returns the following series:

```
0    False
1    False
2     True
3     True
4    False
5    False
dtype: bool
```

Applied to our original series `s`, we can find all words that contain either a or e:

```
s[s.str.contains('[ae]')]
```

This results in:

```
2     a
3    test
dtype: object
```

Note that while `str.contains` currently (as of this writing) defaults to

treating its argument as a regular expression, there are plans for that default value to change. It's thus a good idea to be explicit about your intentions by passing `regex=True`, so that the string isn't taken literally:

```
s[s.str.contains('[ae]', regex=True)]
```

The `str` accessor makes it easy to use pandas to call string methods, and thus work with textual data. However, you should spend some time reviewing the list of string methods in the pandas documentation, to get a good sense of what they are and what they can do.

## 8.2 Exercise 36: Analyzing Alice

In this exercise, we're going to look at the famous book *Alice in Wonderland*, the text of which is made freely available via Project Gutenberg, as well as included along with the data files for this book. Here is what I'd like you to do:

- Open the file `alice-in-wonderland.txt`, and read it into a pandas series or data frame, such that each word is a separate value. (If you choose to read it as a data frame, that's fine. I'll refer to the "series" or "column" when describing our data in this exercise.)
- What are the 10 most common words in the book?
- Does this change if we count the words without regard to case?
- Does it change if we remove punctuation (as defined in `string.punctuation`) from the beginning and end of each word?
- How many capitalized words does the book contain?
- If we ignore punctuation and quotes before the start of a word, how many capitalized words does the book contain?
- Count the number of vowels (a, e, i, o, and u) in each word. What is the average number of vowels per word?

### 8.2.1 Discussion

In this exercise, we used the string functionality in pandas in a variety of ways. To start off, I asked you to read the contents of *Alice in Wonderland* into a series. Normally, we don't read text files into pandas—although to be

honest, a library such as pandas has so many users and use cases that it's quite possible people do this on a regular basis. If you were to feed `open(filename)` into `Series`, then the series would have contained the lines from `alice-in-wonderland.txt`. Instead of that, I asked you to create a series containing the separate words from the file. Doing so is easiest if we use the `read` method on our Python file object, which returns a string from the content. We can then invoke the `str.split` method, returning a list of strings from that original string. We can then use that list to create a series:

```
filename = '../data/alice-in-wonderland.txt'  
s = Series(open(filename).read().split())
```



#### Note

The `read` method returns a string containing the contents of the file. What if the file contains several terabytes of data? Then unless the IT department at your company is unusually generous, you'll find yourself running out of memory. Normally, I suggest that people **not** read an entire file into memory at once, instead iterating over its lines. In this particular case, I know that the file is small, and that there won't be any issues with reading it all at once.

With our series in place, we can start to perform analyze the text that it contains. First, I asked you to find the most common words. As we've seen countless times before, `value_counts` will help us here. Invoking it on our series returns a new series whose index contains our words (i.e., the values from `s`) and whose values (sorted in descending order) are integers, describing how many times each word appeared in `s`:

```
s.value_counts()
```

Not surprisingly, the most common words are "the," "and," "a", and "to." But what if these words appeared at the start of a sentence? Then they would be capitalized, and wouldn't be included in our count. How can we transform all of the words to lowercase, and then find how common they are? We can use the `str` accessor to run the `lower` method on our series. That'll return a new series of strings, on which we can then run `value_counts`:

```
s.str.lower().value_counts().head(10)
```

But wait a second—because of the way that we created our series, by using whitespace characters to indicate the boundary between words, it’s possible that the words have punctuation marks before or after their letters. I thus asked you to repeat the query for the 10 most common words, but only after removing/ignoring punctuation characters. This turns out to be easier than you might imagine, using the `str.strip` method. This method is typically used to remove whitespace from the start or end of a string:

```
s = '   abc   '
s.strip() #1
```

But we can also pass a string argument to `str.strip`, removing any characters from that argument which appear at the start or end of our string:

```
s = ';;;abc;;;'
s.strip(';') #1
```

The `string` module includes a number of predefined strings, including `string.punctuation`, which comes in handy on such occasions:

```
import string
s = ';;;abc;;;'
s.strip('string.punctuation') #1
```

Given a series containing strings, we can get a new series containing those same strings, but without leading and trailing punctuation, by invoking `split` via the `str` accessor:

```
s.strip(string.punctuation)
```

To find the 10 most common words in `s`, ignoring punctuation, we can thus say:

```
s.str.strip(string.punctuation).value_counts().head(10)
```

And while I didn’t ask you to do this, we could get the 10 most common words, ignoring both case and punctuation:

```
s.str.lower().str.strip(string.punctuation).value_counts().head(10)
```

Notice that we had to use `str` twice here—once to run `lower` on the original

series `s`, and then a second time to run `strip` on the series of strings returned by `str.lower`. We'll see more examples of this as we review the other parts of this exercise.

Next, I asked you to count the number of capitalized words in the book. This means finding all of the words that begin with a capital letter, from A through Z. There are several ways to do this, but my favorite (as you might have guessed by now) would be to use a regular expression. Given that the pandas string method `str.contains` supports regular expressions, we can say the following:

```
s.str.contains('^[A-Z]\w*$', regex=True)
```

This returns a boolean series with the same index as `s`. The value will be `True` whenever the word starts with a capital letter (anchored to the start of the string with `^`) and contains zero or more alphanumeric characters (`\w*`) through the end of the word. (We have to allow for zero-or-more characters, because of single-letter capitalized words, such as `A` and `I`.)

With this in hand, we can apply the boolean series to `s`:

```
s[s.str.contains('^[A-Z]\w*$', regex=True)]
```

Then we can apply the `count` method, to find how many values the series contains:

```
s[s.str.contains('^[A-Z]\w*$', regex=True)].count()
```

But wait: What if the word has a punctuation mark, such as quotes, before the initial capital letter? To get an accurate count, we'll need to remove punctuation from the ends of the words, and then look for which are capitalized. Here's how we can do that:

```
s[s.str.strip(string.punctuation).str.contains('^[A-Z]\w*$', rege
```

Here, we first remove punctuation from the start and end of each word, then feed the resulting series into `str.contains` with our regular expression. That returns a boolean series, which we can apply back to `s`, thus finding the total number of capitalized words.

Next, I asked you to calculate the mean number of vowels in each word. This requires first finding a way to calculate the number of vowels in each word, and then calculating the mean value. The easiest way to do this is with the `apply` method, which lets us run a function of our choice on each element of the series. So first, we'll have to write a function that counts vowels. Here's my implementation:

```
def count_vowels(one_word):
    total = 0
    for one_letter in one_word.lower():
        if one_letter in 'aeiou':
            total += 1

    return total
```

This is a simple Python function that takes a string as an argument, counts the vowels in it, and returns an integer. We can then apply this function to every element of our series `s`, getting a new series back:

```
s.apply(count_vowels)
```

I asked for the mean number of vowels in each word. Since we now have a series of integers, we can get that back with:

```
s.apply(count_vowels).mean()
```

## 8.2.2 Solution

```
filename = '../data/alice-in-wonderland.txt'
s = Series(open(filename).read().split()) #1

s.value_counts().head(10) #2
s.str.lower().value_counts().head(10) #3
s.str.strip(string.punctuation).value_counts().head(10) #4
s[s.str.contains('^[A-Z]\w*$', regex=True)].count() #5
s[s.str.strip(string.punctuation).str.contains('^[A-Z]\w*$', rege

def count_vowels(one_word): #7
    total = 0
    for one_letter in one_word.lower():
        if one_letter in 'aeiou':
            total += 1
```

```
    return total

s.apply(count_vowels).mean() #8
```

### 8.2.3 Beyond the exercise

- One: What is the mean of all integers in Alice?
- Two: What words in Alice don't appear in the dictionary? Which are the five most common such words?
- Three: What are the min and max number of words per paragraph?

## 8.3 Exercise 37: Wine words

If you're like me, then you might enjoy having a glass with your dinner. On occasion, you might even read the wine's description on the back of the bottle, where the winemakers use flowery language to describe the winemaking process, and the flavors that you might detect when drinking the wine. I know that I'm not the only person who sometimes raises an eyebrow at the words they use in these descriptions. I decided to use pandas to better understand what words are used in describing wine, and whether we can find any interesting insights from these words.

We've already looked at the wine-review database in exercise 35. In this exercise, we're going to look at the words that reviewers used to describe the wines, and see if particular words are more likely to occur in specific provinces and varieties. Along the way, we'll find ways to use pandas to analyze text in some new ways. Here's what I want you to do:

- Open the file `winemag-150k-reviews.csv`, and read it into a data frame. We only need the columns "country," "province," "description," and "variety."
- What are the 10 most common words containing 5 or more letters in the wine descriptions? Turn all words into lowercase, and remove all punctuation and symbols at the start or end of each word, for easier comparison. Also: remove the words flavors, aromas, finish, and drink.
- What are the 10 most common words for California wines?
- What are the 10 most common words for French wines?

- What are the 10 most common words for white wines? For our purposes, we'll look for Chardonnay, Sauvignon Blanc, and Riesling.
- What are the 10 most common words for red wines? For our purposes, we'll look for Pinot Noir, Cabernet Sauvignon, Syrah, Merlot, and Zinfandel.
- What are the 10 most common words for rosé wines?
- Show the 10 most common words for the 5 most common wine varieties.

### 8.3.1 Discussion

For starters, I asked you to create a data frame with the wine information. We're only going to need four columns, so we can load only those:

```
filename = '../data/winemag-150k-reviews.csv'
df = pd.read_csv(filename,
                 usecols=['country', 'province', 'description', 'var
```

Next, I wanted to start performing some analysis on the words. But because I'm going to be running the same type of analysis on different subsets of the data frame, I decided that I would benefit from writing a function. What would this function have to do?

- accept a series of text (i.e., wine descriptions)
- turn the text into lowercase (for easier comparison)
- turn that into a series of individual words
- remove leading and trailing punctuation
- remove words with fewer than five letters
- remove common wine-related words
- find the 10 most commonly occurring words

Fortunately, it's not that hard to write such a function, which I called `top_10_words`. The function expects to receive one argument, a pandas series of strings, which we call `s`. Each of the strings in the series is assumed to contain multiple words, separated by whitespace.

The first thing we want to do is turn all of the strings in the series to lowercase, for easier counting. We can do that by using the `str` accessor and

the `len` method:

```
words = s.str.lower()
```

We next want to take our series of sentences, and turn it into a series of words. That is, instead of having multiple words in each row, we want to have a single word in each row. This means that we're going to create a series that's larger—and potentially **much** larger—than the input series `s`.

If you're familiar with Python string methods, then you won't be surprised to know that we'll use the `split` method here, via the `str` accessor. (Note that this means we'll need to specify `str` a second time, so that we can run `split` on each element of the series returned from `str.lower()`.) `split` takes a string, and breaks it apart wherever it encounters a delimiter, such as `:` or `,`. In this case, we won't specify a delimiter, which means that `split` will use any whitespace—space, tab, newline, carriage return, and vertical tab—to break them apart:

```
words = s.str.lower().str.split()
```

The good news is that we have now separated the words from one another. The bad news is that our series still contains the same number of rows as it did before. Now, each row contains a list of strings, rather than a single string.

Fortunately, the `explode` method takes a series containing an iterable of objects (e.g., a list of strings), and returns a new series, one in which each object has its own row. We can thus get each word in its own row as follows:

```
words = s.str.lower().str.split().explode()
```

We could stop there, but I'd like to clean things up just a bit more: I want to remove any punctuation characters that might be at the start or finish of any word. That'll avoid problems when counting words that come at the start or end of a sentence; otherwise, we would include leading and trailing punctuation. The easiest way to do that is with the Python `str.strip` method. We normally think of `strip` as a method that removes whitespace at the start or end of a string, but that's just the default behavior. We can pass a string containing characters we want gone from the start and finish of each string.

The result is a new series, one in which the strings don't have any of these characters at their start or end:

```
words = s.str.lower().str.split().explode().str.strip(',$.?!$%')
```

We now have, in `words`, a series containing individual, lowercase words without any leading or trailing punctuation. Now we want to remove words that have fewer than five letters. We can do that using a boolean index based on the output from the `len` method on the `str` accessor:

```
words[(words.str.len())>=5]
```

But that's not the only filter we want to put on `words`. We also want to remove a number of common words that'll crop up in nearly any wine description or review. We can use the `isin` method in a series, passing a list of strings as an argument, to find out which rows are and aren't in that list:

```
common_wine_words = ['flavors', 'aromas', 'finish', 'drink', 'pal  
~words.isin(common_wine_words)
```

We can then combine these two mask indexes to get only those words containing at least 5 characters, and which don't appear in `common_wine_words`:

```
words[(words.str.len())>=5) & (~words.isin(common_wine_words))]
```

Note that we use `~`, the boolean "not" operator in pandas, to flip the boolean index that we get back from `words.isin`.

Our function is called `top_10_words`, because it's supposed to return the ten most common words found in the wine reviews. Given that `words` is now a series of words, we can run `value_counts`, followed by `head(10)`, and return the ten words most commonly found:

```
return words[(words.str.len())>=5) & (~words.isin(common_wine_word
```

With this, we now have a complete function, `top_10_words`, which we can apply to any series of words:

```
def top_10_words(s):
```

```
common_wine_words = ['flavors', 'aromas', 'finish', 'drink',  
words = s.str.lower().str.split().explode().str.strip(',$?!$'  
return words[(words.str.len())>=5) & (~words.isin(common_wine_
```

Right away, we can try to apply our function to all of wines in the review database:

```
top_10_words(df['description'])
```

Next, I asked you to find the 10 most common words used in French wine reviews. We'll need to extract the description column for wines made in France:

```
df.loc[df['country'] == 'France', 'description']
```

We can then pass the resulting series to `top_10_words`:

```
top_10_words(df.loc[df['country'] == 'France', 'description'])
```

Next, I asked to find the words most commonly associated with wines made outside of California. We'll need to search on the province column, and then apply the `!=` operator to find those from outside of that state:

```
top_10_words(df.loc[df['province'] != 'California', 'description']
```

Notice that in this data set, you have to pay attention to the province column, which is distinct from the country column. Additional columns allow you to zero in on a particular region within a country; as you might know, different regions are known for producing not only different types of wines, but distinctive flavors specific to those regions.

Next, I thought it would be interesting to compare the words used most often for white, red, and rose wines. I gave a (very non-definitive) list of wines of each type, and then asked you to find the top 10 words used in each of their descriptions. The queries will be identical, except for the lists:

```
top_10_words(df.loc[df['variety'].isin(['Chardonnay', 'Sauvignon  
top_10_words(df.loc[df['variety'].isin(['Pinot Noir', 'Cabernet S  
top_10_words(df.loc[df['variety'] == 'Rosé', 'description'])
```

Notice how the `isin` method allows us to perform an "or" search—one that

we could certainly do with pandas boolean operators and a mask index, but which becomes shorter and more readable with `isin`.

Finally, I asked that you find the 10 most common words for the five most commonly mentioned wine varieties. In order to do that, we'll first need to determine the three most mentioned varieties:

```
df['variety'].value_counts().head(5).index
```

Here, we run `value_counts` on the varieties, in order to find out how common each variety is in the database. We then use `head(5)` to find the five most common varieties. We can then find all reviews for one of these varieties using `isin`:

```
df.loc[df['variety'].isin(df['variety'].value_counts().head(5).in
```

Notice that we couldn't just use `isin` on the values we got back from `value_counts`, because those would be numbers. Instead, we needed to check the index of the resulting series, which contained

Finally, we can find the top 10 words used in reviews for these varieties by again applying our function, `top_10_words`:

```
top_10_words(df.loc[df['variety'].isin(df['variety'].value_counts
```

### 8.3.2 Solution

```
filename = '../data/winemag-150k-reviews.csv'
df = pd.read_csv(filename,
                 usecols=['country', 'province', 'description', 'var

def top_10_words(s):
    common_wine_words = ['flavors', 'aromas', 'finish', 'drink',
    words = s.str.lower().str.split().explode().str.strip(',$.?!$
    return words[(words.str.len()>=5) & (~words.isin(common_wine_

top_10_words(df['description'])
top_10_words(df.loc[df['country'] == 'France', 'description'])
top_10_words(df.loc[df['province'] != 'California', 'description'])
top_10_words(df.loc[df['variety'].isin(['Chardonnay', 'Sauvignon
top_10_words(df.loc[df['variety'].isin(['Pinot Noir', 'Cabernet S
top_10_words(df.loc[df['variety'] == 'Rosé', 'description'])
```

```
top_10_words(df.loc[df['variety'].isin(df['variety'].value_counts
```

### 8.3.3 Beyond the exercise

- Which country's wines got the highest average score for all wines?
- Create a pivot table in which the index contains countries, the columns contain varieties, and the cells contain mean scores. Only include the top 10 varieties.
- What is the correlation between the number of wines offered by a country, and the mean score for that country? That is: If a country enters more wines, does its average score in reviews go up?

## 8.4 Exercise 38: Values into columns (learning goals: `str.split`, `expand=True`)

In the Stack Overflow survey, developers indicated which programming languages they're currently using. Unfortunately, the languages are in a single text column, separated by semicolons. In this exercise, you'll work with that data, extracting and analyzing it in a variety of ways:

- Open the file `so_2021_survey_results.csv`, and read it into a data frame. We only need the columns `LanguageHaveWorkedWith`, `LanguageWantToWorkWith`, `Country`, and `CompTotal`.
- What are the different programming languages that developers currently use?
- What are the 10 programming languages most commonly used today?
- What are the 10 programming languages people most want to use?
- What languages are on both top-10 lists?
- What languages in the top 10 have people worked with, but **don't** want to work with in the future?
- What is the most popular (current) language used by people in each country?
- What is the mean number of languages used in the last year?
- What is the greatest number of languages people listed as having used in the last year?
- How many people chose that largest number?

- How many people in the survey claim salaries of \$2m or above?
- Remove rows in which salaries are below \$2m
- Turn the 'LanguageHaveWorkedWith' column into "dummy" columns in df, such that each language is its own column.
- If you want to maximize your salary, and have to choose two languages from Python, JavaScript, and Java, then what combination would be best?

### 8.4.1 Discussion

In this exercise, we looked at one of the most useful and interesting parts of the Stack Overflow survey, namely the list of programming languages that participants marked themselves as having used in the last year. The good news is that we have rich data which can give us insights into developers from around the world. The bad news is that these languages are all in a single column of the original CSV, making it challenging to work with. This exercise asked you to use a number of techniques that we can use when working with such data.

To begin with, we had to load the Stack Overflow data, which I did by reading it all into a data frame:

```
filename = '../data/so_2021_survey_results.csv'
df = pd.read_csv(filename, usecols=['LanguageHaveWorkedWith',
                                   'LanguageWantToWorkWith',
                                   'Country', 'CompTotal'])
```

In order to reduce memory usage, and thus allow pandas to correctly determine what type of data should be in each column, I specified which columns I wanted to load into the data frame.

The first question I wanted to answer was what different programming languages programmers currently use. The answers are all in `LanguageHaveWorkedWith`, a text (string) column. However, people answering the survey could provide more than one answer—which explains why this field contains numerous sub-fields, separated by semicolons. For example, here are five rows from the file:

```
0          C++;HTML/CSS;JavaScript;Objective-C;PHP;Swift
```

```
9                                     C++;Python
11      Bash/Shell;HTML/CSS;JavaScript;Node.js;SQL;Typ...
12                                     C;C++;Java;Perl;Ruby
16      C#;HTML/CSS;Java;JavaScript;Node.js
```

Notice that in the third row, the respondent indicated so many programming languages that pandas doesn't even display all of them by default, ending the string with ....

If we are going to query the data frame based on which programming language(s) people used, we'll need to be able to treat these strings as separate fields, not just as large strings. The best way to do that, as we saw above, is to first run `split` on our string column (resulting in a series of Python lists), and then run the `explode` method on the result:

```
df['LanguageHaveWorkedWith'].str.split(';').explode()
```

The result of this query is a series of strings—all of the different strings that the `LanguageHaveWorkedWith` column had contained. But now, each of the programming languages is in a separate row. This allows us to count them by using `value_counts`:

```
df['LanguageHaveWorkedWith'].str.split(';').explode().value_count
```

In this way, we can see how many times each of the languages was mentioned, sorted from the most popular (JavaScript) to the least popular (APL). We're only interested in the 10 most commonly found languages, so we'll cut it off after the top 10:

```
df['LanguageHaveWorkedWith'].str.split(';').explode().value_count
```

I'm actually less interested in the numbers than in the names of those languages. I can thus request the index from the returned series:

```
df['LanguageHaveWorkedWith'].str.split(';').explode().value_count
```

Finally, I'll assign that to a variable, `have_worked_with`, because I'll be needing these values in just a little bit, and it's easier to work with them from a variable than a long, repeated query.

```
have_worked_with = df['LanguageHaveWorkedWith'].str.split(';').ex
```

Next, I performed the same query on the column `LanguageWantToWorkWith`, containing the answers to the question: What language do you hope to work with in the next year? Besides the name of the column and the variable to which I assign the results, the query is the same:

```
want_to_work_with = df['LanguageWantToWorkWith'].str.split(';').e
```

Next, I asked what languages are on both top-10 lists. Because pandas index objects are similar to series, I could run the `isin` method, asking which elements of `want_to_work_with` are in `have_worked_with`—then using the resulting boolean index on `want_to_work_with`:

```
want_to_work_with[want_to_work_with.isin(have_worked_with)]
```

But it turns out that pandas makes it easy to do this, with the "intersection" method. Note that this method works on index objects, and not on series:

```
want_to_work_with.intersection(have_worked_with)
```

The next question asks: What languages in the top 10 have people worked with, but **don't** want to work with in the coming year? We can again use `isin` to find which elements of `have_worked_with` are in `want_to_work_with`:

```
have_worked_with.isin(want_to_work_with)
```

This returns a boolean index. We can reverse it, such that we find which elements of `have_worked_with` are **not** in `want_to_work_with`:

```
~have_worked_with.isin(want_to_work_with)
```

Now we can apply the resulting boolean index to `have_worked_with`:

```
have_worked_with[~have_worked_with.isin(want_to_work_with)]
```

And we discover that despite their current popularity, people aren't excited about working with either shell scripts or C++ in the future. (I understand and agree!)

Next, I wanted to find out which language is most popular in each country. That is, we've already found that JavaScript is the most popular programming language overall. Is this universally true? Our data frame has a Country column, so it stands to reason that we could use groupby to find the most popular language per country. But there's a problem, namely that the languages are all in the LanguageHaveWorkedWith column. If we use explode to put each language on its own row, the resulting series will be a different length than df, meaning that we cannot add it as a new column.

However, the series that we get back from explode has the same index as the original series on which it was run. Meaning that if the original column had an index of 0 and mentioned both Python and JavaScript, then the resulting series will have two rows, both with an index of 0, one with Python and the other with JavaScript.

This means that while we cannot assign the exploded series as a column, we can use join to merge the series onto the data frame.

First, let's create a new series, all\_languages, containing the programming languages. We don't need to do this, but it'll make the join easier to understand:

```
all_languages = df['LanguageHaveWorkedWith'].str.split(';').explod
```

Then, we can perform our join. Note that while join is a method on data frames (not series), we can pass either a data frame or a series as the argument to it. In other words, we can say:

```
df.join(all_languages)
```

Actually, the above code won't work: We get an error, because the data frame that results from this join will have two columns named LanguageHaveWorkedWith. There are several ways to solve this problem; we could set LanguageHaveWorkedWith.name to a different value. We could pass a value to one or both of the lsuffix or rsuffix parameters, adding a suffix to joined columns from the left or right, thus avoiding a clash. But I think that the easiest thing is actually to realize that we really only care about the Country column in the data frame, meaning that we can run join on it, and it

alone:

```
df[['Country']].join(all_languages)
```

Notice that we used double square brackets around 'Country', to ensure that the result was a data frame (i.e., multiple columns) rather than a series (i.e., one column). Now that we've created this new data frame, we can use `groupby` on it:

```
df[['Country']].join(all_languages).groupby('Country')
```

This gives us a `groupby` object, but now we have to apply a method. And what aggregation method do we want to use? The normal choices are `mean`, `count`, and `std`, but here we want the value that appears the most—often known as the "mode." However, there isn't any `mode` method that we can apply—at least, no such method is provided directly. However, we can use the method `pd.Series.mode`, applying it by passing it to the `agg` method on our `groupby` object:

```
df[['Country']].join(all_languages).groupby('Country').agg(pd.Series
```

The result is a 1-column data frame whose index contains country names, and whose values represent the most popular language in each country. We can even find the relative popularity of different languages with `value_counts`:

```
df[['Country']].join(all_languages).groupby('Country').agg(pd.Series
```

Next, I asked you to find the mean number of languages that developers used in the last year. What we can do is break `LanguageHaveWorkedWith` into pieces, and then run `len` on that list. That'll give us a series of integers, on which we can then run `mean`:

```
df['LanguageHaveWorkedWith'].str.split(';').str.len().mean()
```

Notice that we have to use the `str` accessor twice here: First, we use it to run the `split` method, turning our series of strings into a series of lists. Then we use `str` a second time, this time to run `len` on each element, giving us a series of integers—on which we can then run `mean`. And yes, we're using the `str` accessor to run `len` on lists; the accessor will try to run the method on

whatever data it has, and since lists also support `len`, we're just fine.

Next, I wanted to find out the greatest number of languages anyone had indicated they used in the last year. I can do that by running `max`:

```
df['LanguageHaveWorkedWith'].str.split(';').str.len().max()
```

At least one person indicated they had worked with 38 different programming languages in the last year... out of the 38 that were listed on the survey questionnaire, leading me to wonder if they simply checked all of the boxes. Maybe there were others who did the same thing? I asked you to determine how many people had that same number of languages marked:

```
df['LanguageHaveWorkedWith'][df['LanguageHaveWorkedWith'].str.spl
```

Here, I used the length of the post-split list in a comparison, resulting in a boolean index. I applied the boolean index to the column `LanguageHaveWorkedWith`, and then applied `count` to find out how many rows matched.

Next, I asked you to look at developer salaries, as reported in the survey. First, how many developers are making more than \$2 million/year?

```
df['CompTotal'][df['CompTotal'] >= 2_000_000].count()
```

Wow—2,360 people reported that kind of salary! Let's remove them from our data, since it'll otherwise get rather skewed:

```
df = df[df['CompTotal'] < 2_000_000]
```

We'll get back to salaries in a moment, but now I want to take the `LanguageHaveWorkedWith` column, and turn it into multiple columns. That'll allow us to more easily analyze the individual languages. Doing this is known as creating "dummy columns." Instead of a column containing the string `'JavaScript;Python'`, we'll create one column called `JavaScript` and another called `Python`, putting 1s where the person marked themselves as using JavaScript, and 0s where they indicated they did not.

I can create a new data frame of dummy values based on

LanguageHaveWorkedWith using the `str.get_dummies` method:

```
df['LanguageHaveWorkedWith'].str.get_dummies(sep=';')
```

But how can I then integrate this new data frame into our existing one? The answer is `pd.concat`, which we've used in the past. The difference is that we want to join them horizontally (i.e., combining them left-and-right, rather than top-and-bottom). To tell `pd.concat` this, we need to indicate `axis='columns'`, similar to what we've done with other methods in the past, such as `df.drop`. We can then assign the result of the concatenation back to `df`:

```
df = pd.concat([df, df['LanguageHaveWorkedWith'].str.get_dummies(
```

With these dummy columns in place, we can ask questions regarding salaries and language knowledge. First, what was the average salary of someone who knows Python and JavaScript, but not Java?

```
df['CompTotal'][(df['Python'] == 1) &
                 (df['JavaScript'] == 1) &
                 (df['Java'] == 0)].mean()
```

I got a result of \$126,817.

What about someone who knows Python and Java, but not JavaScript?

```
df['CompTotal'][(df['Python'] == 1) &
                 (df['JavaScript'] == 0) &
                 (df['Java'] == 1)].mean()
```

Here, I got a result of \$162,737.

Finally, what about someone who knows Java and JavaScript, but not Python?

```
# Java and Javascript, not Python
df['CompTotal'][(df['Python'] == 0) &
                 (df['JavaScript'] == 1) &
                 (df['Java'] == 1)].mean()
```

This resulted in \$140,867.

## 8.4.2 Solution

```
filename = '../data/so_2021_survey_results.csv'
df = pd.read_csv(filename, usecols=['LanguageHaveWorkedWith',
                                   'LanguageWantToWorkWith',
                                   'Country', 'CompTotal'])

df['LanguageHaveWorkedWith'].str.split(';').explode().value_count

have_worked_with = df['LanguageHaveWorkedWith'].str.split(';').ex
want_to_work_with = df['LanguageWantToWorkWith'].str.split(';').e

want_to_work_with.intersection(have_worked_with)
have_worked_with[~have_worked_with.isin(want_to_work_with)]

all_languages = df['LanguageHaveWorkedWith'].str.split(';').explor
df[['Country']].join(all_languages).groupby('Country').agg(pd.Ser

df['LanguageHaveWorkedWith'].str.split(';').str.len().mean()
df['LanguageHaveWorkedWith'].str.split(';').str.len().max()

df['LanguageHaveWorkedWith'][df['LanguageHaveWorkedWith'].str.cou
df['CompTotal'][df['CompTotal'] >= 2_000_000].count()
df = df[df['CompTotal'] < 2_000_000]

df = pd.concat([df, df['LanguageHaveWorkedWith'].str.get_dummies(

df['CompTotal'][(df['Python'] == 1) &
                 (df['JavaScript'] == 1) &
                 (df['Java'] == 1)].mean()

df['CompTotal'][(df['Python'] == 1) & (df['JavaScript'] == 0) & (
df['CompTotal'][(df['Python'] == 0) & (df['JavaScript'] == 1) & (
```

## 8.4.3 Beyond the exercise

- When developers are stuck (as indicated in the column `NEWStuck`), what are the three things they're most likely to do?
- What proportion of the survey respondents marked their gender as Man? Does that proportion seem similar to your real-life experiences?
- On average, what proportion of their years coding have been done professionally?

## 8.5 Summary

In this chapter, we looked at various ways that pandas lets us work with textual data, especially via the `str` accessor. The combination of Python's rich string methods along with the various ways that pandas lets us manipulate series and data frames gives us a great deal of flexibility, and lets us ask a wide variety of sophisticated questions that aren't directly numerical. Many data sets, such as the ones we looked at in this chapter, contain a mix of numeric and textual data, and being able to work with the text alongside the numbers is especially useful.

# 9 Dates and times

Programming languages' core data structures reflect the types of information that we work with on a regular basis. It makes sense that we'll have numbers, because we use numbers a lot. We use lots of text, so strings make sense, as well. And of course, we need collections of various sorts, so every language provides some of those—in the case of Python, we have lists, tuples, dictionaries, and sets, for starters.

Modern programming languages also support another type of data, one which we (as people) use on a regular basis, but which weren't part of the programming canon when I started my career: Dates and times. It seems obvious in retrospect that dates and times, which are such an essential part of our lives, should be a main part of our programming languages. But it turns out that dealing with dates and times is hard, with all sorts of tricky issues to deal with—from leap years, to time zones, to the odd data structures we need in order to computerize a calendar that wasn't exactly designed with computers in mind.

Both the Python language and pandas handle time data with two different data structures: A "timestamp" data type (also known as a "datetime" in many languages and systems) handles specific points in time, one that you can point to using a calendar. A timestamp happens once, and only once—when you were born, when your plane will be taking off, when you and your date will meet at a restaurant, or when the meeting was scheduled to end. You can describe a timestamp with a particular year, month, day, hour, minute, and second.

A second, complementary data type is that of the "timedelta," known in some systems as an "interval." A time delta represents a time stamp—the distance between two timestamp objects. So the meeting's scheduled start and end can be represented as timestamps, but the time that the meeting takes is a timedelta.

Not surprisingly, lots of the data that we want to analyze contains time and

date information. And thus, it's good to know that pandas can handle dates and times quite flexibly. We can read data in from files, turning columns into timestamps. We can also convert existing values—both individual values and series objects—into timestamps. We can perform calculations with `timedeltas`, and perform comparisons with them.

But pandas goes further than that, allowing us to use date and time information in our indexes. This makes it even easier to search for data that took place during specific periods. Even better, we can perform "resampling," which is most easily described as "grouping by time periods."

This chapter's exercises all take advantage of these capabilities in pandas to explore information that has to do with dates and times. Along the way, you'll get experience working with a variety of date formats and input types, as well as producing reports based on those types.

## 9.1 Useful references

Table 9.1. What you need to know

Concept	What is it?	Example	To learn more
<code>pd.to_datetime</code>	If passed a series of strings, returns a series of <code>Timestamp</code> objects	<code>pd.to_datetime(s['when'])</code>	<a href="https://pandas.pydata.org/pandas-docs/stable/timeseries.html">pandas.pydata.org/pandas-docs/stable/timeseries.html</a>
<code>pd.to_timedelta</code>	If passed a series of strings, returns a series of <code>Timedelta</code> objects	<code>pd.to_timedelta(s['how_long'])</code>	<a href="https://pandas.pydata.org/pandas-docs/stable/timeseries.html">pandas.pydata.org/pandas-docs/stable/timeseries.html</a>

	objects		
<code>pd.read_csv</code>	Returns a new data frame based on CSV input	<code>df = pd.read_csv('myfile.csv')</code>	<a href="https://pandas.pydata.org/pandas-docs/stab">pandas.py docs/stab</a>
<code>time.strftime</code>	Produce a string, based on a time value	<code>time.strftime(a_time, a_format)</code>	<a href="https://docs.python.org/3/library/time.html">docs.pyth</a>
<code>time.strptime</code>	Parses a string into a time object	<code>time.strptime(time_string)</code>	<a href="https://docs.python.org/3/library/time.html">docs.pyth</a>
<code>df.to_csv</code>	Writes a CSV file based on a data frame	<code>df.to_csv('mydata.csv')</code>	<a href="https://pandas.pydata.org/pandas-docs/stab">pandas.py docs/stab</a>
<code>df.resample</code>	Performs a time-based groupby operation, on a specified period of time	<code>df.resample('1M')</code>	<a href="https://pandas.pydata.org/pandas-docs/stab">pandas.py docs/stab</a>

<code>s.diff</code>	returns a new series with the same index as s, but whose values indicate the difference between that value and the previous value	<code>s.diff()</code>	<a href="https://pandas.pydata.org/pandas-docs/stab">pandas.pydocs/stab</a>
<code>s.pct_change</code>	returns a new series with the same index as s, but whose values indicate the <b>percentage</b> difference between that value and the previous value	<code>s.pct_change()</code>	<a href="https://pandas.pydata.org/pandas-docs/stab">pandas.pydocs/stab</a>

### Creating datetime and timedelta objects

As we've repeatedly seen, pandas largely avoids built-in Python data structures in favor of its own types, or those defined by NumPy. This is also

the case when it comes to dates and times: To represent a specific point in time, we use the `Timestamp` class, instead of either the `datetime.datetime` class that comes with Python or the `np.datetime64` class that comes with NumPy.

The standard way to create `Timestamp` objects is with the module-level function `to_timestamp`, which takes a variety of argument types. If passed a single argument, it returns one `Timestamp`. For example, we can get the current date and time by passing the string `'now'`:

```
pd.to_datetime('now')
```

But it's far more common, and useful, for us to call `pd.to_datetime`

```
s = Series(['1970-07-14', '1972-03-01', '2000-12-16', '2002-12-17'])
pd.to_datetime(s)
```

The above code returns a new series:

```
0    1970-07-14
1    1972-03-01
2    2000-12-16
3    2002-12-17
4    2005-10-31
dtype: datetime64[ns]
```

Don't be confused by the indication that the `dtype` is `datetime64`, a type from NumPy; the values are all of type `Timestamp`, a pandas type.

In this example, the strings that we fed to `to_datetime` were pretty unambiguous, and easy to parse. But what if we have slightly different strings, using month names instead of numbers?

```
s = Series(['1970-Jul-14', '1972-Mar-01', '2000-Dec-16', '2002-Dec-17'])
pd.to_datetime(s)
```

Actually, it'll work just fine: That's because `pd.to_datetime` is fairly smart and flexible, and can parse a number of different date formats. So the above format will work, as will this one:

```
s = Series(['14-Jul-1970', '01-Mar-1972', '16-Dec-2000', '17-Dec-2002'])
pd.to_datetime(s)
```

But what if we pass dates that are a bit more ambiguous? For example, what if the months are all numbers:

```
s = Series(['14-07-1970', '01-03-1972', '16-12-2000', '17-12-2002'])
pd.to_datetime(s)
```

Once again, it works just fine. However, there are times when it's less obvious, and where human culture and tradition plays a role. For example, take the following:

```
s = Series(['01/03/1972', '05/12/1995'])
pd.to_datetime(s)
```

Should pandas interpret these dates as the 1st of March, or the 3rd of January, and as the 5th of December, or the 12th of May? By default, ambiguous date formats are assumed to have the day first, as done in the United States.

However, you can override that by passing `dayfirst=False` to `pd.to_datetime`:

```
s = Series(['01/03/1972', '05/12/1995'])
pd.to_datetime(s, dayfirst=False)
```

All of the above examples have only included dates, but we could include time information, as well:

```
s = Series(['1970-07-14 8:00', '1972-03-01 10:00 pm', '2000-12-16 12:15:28'])
pd.to_datetime(s)
```

The above code returns:

```
0    1970-07-14 08:00:00
1    1972-03-01 22:00:00
2    2000-12-16 12:15:28
3    2002-12-17 18:17:00
4    2005-10-31 23:51:00
dtype: datetime64[ns]
```

Notice that I sometimes included seconds, and in one case indicated am/pm rather than using a 24-hour clock. pandas tries hard to understand all of these formats, and to interpret them as best as possible.

What if you have several series representing the year, month, and date? You

can use `pd.to_datetime` to get a new `Timestamp` series based on those inputs. This is especially useful if you're trying to create a `Timestamp` column from a data frame:

```
df = DataFrame([s.split('-')
                for s in ['14-07-1970', '01-03-1971', '16-12-2000', '17-12-2002']
                columns='day month year'.split()])
pd.to_datetime(df[['year', 'month', 'day']])
```

This code results in:

```
0    1970-07-14
1    1971-03-01
2    2000-12-16
3    2002-12-17
4    2005-10-31
dtype: datetime64[ns]
```

All of this is fine, but it ignores a common use case: We are loading a CSV file, and one or more columns in the file are datetime information. How can we ensure that these columns are interpreted as `Timestamp` data, and not as strings? We need to tell pandas to do this, using the `parse_dates` parameter in the `read_csv` function. We can pass a list of columns, either as names (strings) or as integers (indexes). For example:

```
pd.read_csv(filename, parse_dates=['birthday', 'anniversary'])
```

There are a variety of different parameters that you can pass to influence the parsing process. One of them is `dayfirst`, which works just as we saw above—to indicate that the dates being read start with days (as done in Europe) rather than with months (as done in the United States).

Once we have a `Timestamp` series, we can use the `dt` accessor to retrieve a variety of different parts of each object. For example:

```
s.dt.month           # month number
s.dt.month_name     # month name
s.dt.hour           # hour
s.dt.day_of_week    # day of week
s.dt.is_leap_year   # is it a leap year?
```

Some of these attributes return numbers, whereas others return boolean

values. The full list of attributes you can retrieve via the `dt` accessor starts at [pandas.pydata.org/docs/reference/api/pandas.Series.dt.date.html](https://pandas.pydata.org/docs/reference/api/pandas.Series.dt.date.html).

Finally: I mentioned at the start of this chapter that when we work with dates and times, we need two distinct data types. We've spent some time looking at the first one, namely timestamps. But what about time deltas, also known as intervals? We can generally say:

```
datetime - datetime = interval
datetime + interval = datetime
datetime - interval = datetime
```

In other words: Given two datetime objects, we can get an interval object representing the time between them. For example, given a birth date and a death date, we can calculate the length of someone's life. And given a datetime and an interval, we can get the datetime on the other side of that interval. For example, given a meeting start time and its length, we can find out when it ends—or similarly, if given a meeting end time and its length, we can calculate when it started.

pandas allows us to perform precisely this type of calculation. For example, if we have two timestamp series, subtracting one from the other gives us a `timedelta` series. For example:

```
s = Series(['1970-07-14 8:00', '1972-03-01 10:00 pm', '2000-12-16
s = pd.to_datetime(s)
pd.to_datetime('2021-July-01') - s
```

The subtraction operation is broadcast to every element in `s`, returning a series of `timedelta64` objects:

```
0    18614 days 16:00:00
1    18018 days 02:00:00
2     7501 days 11:44:32
3     6770 days 05:43:00
4     5721 days 00:09:00
dtype: timedelta64[ns]
```

If you want to create a `timedelta` object or series, you can also call `pd.to_timedelta`, much as you can call `pd.to_timestamp`. The function's argument is typically going to be a string or series of strings, each describing

a time span, such as '1 hour' or '2 days'.

The pieces of a `timedelta` can be retrieved using the `components` attribute. For example:

```
pd.to_timedelta('2 days 3:20:10').components
```

The above returns:

```
Components(days=2, hours=3, minutes=20, seconds=10, milliseconds=
```

If you have a series of `timedelta` objects, you can retrieve each individual component, as well as the `components` attribute, via the `dt` accessor, much as you can do with `Timestamp` objects.

Now that you've seen how you can create and retrieve from `Timestamp` and `timedelta` objects, you're all set to start working through the exercises in this chapter, which use these skills to answer questions about a number of data sets.

## 9.2 Exercise 39: Short, medium, and long taxi rides

We have already looked at taxi rides, and have even (in Exercise 30) looked at short, medium, and long taxi rides. However, in that exercise, we considered the distance traveled. In this exercise, we'll look at the taxi rides from the perspective of how much time the ride took. Specifically, I want you to:

- Load taxi data from January 2019 into a data frame, using only the columns `tpep_pickup_datetime`, `tpep_dropoff_datetime`, `passenger_count`, `trip_distance`, and `total_amount`, making sure to load `tpep_pickup_datetime` and `tpep_dropoff_datetime` as `datetime` columns.
- Create a new column, `trip_time`, containing the amount of time each taxi ride took as a `timedelta`.
- What number and percentage of rides took less than 1 minute?
- What was the average fare paid by people taking these short trips?
- What number and percentage of rides took more than 10 hours?

- Now create a new column, `trip_time_group`, in which the values will be short (< 10 minutes), medium ( $\geq$  between 10 minutes and 1 hour), or long (> 1 hour).
- What proportion of rides were in each group?
- For each value in `trip_time_group`, what was the average number of passengers?

## 9.2.1 Discussion

This exercise started similar to many others involving the taxi data. But whereas we were previously willing to let pandas determine the dtype of each column on its own, here we needed to tell it to parse two of the columns as `Timestamp` objects. We could, of course, have imported them as text (i.e., the default) and then run `pd.to_timestamp` on them, but this makes the process a bit easier and cleaner. We can say:

```
filename = '../data/nyc_taxi_2019-07.csv'

df = pd.read_csv(filename,
                 usecols=['tpep_pickup_datetime',
                          'tpep_dropoff_datetime',
                          'trip_distance', 'passenger_count', 'total_amount'],
                 parse_dates=['tpep_pickup_datetime', 'tpep_dropoff_datetime'])
```

Notice that I need to include the two timestamp columns, `tpep_pickup_datetime` and `tpep_dropoff_datetime`, both in the `usecols` list and also in the `parse_dates` list. In addition, this only works without any additional hints or tuning because the taxi dates are all stored in an unambiguous format of `YYYY-MM-DD`.

We can double check that the columns have been interpreted correctly by invoking the `dtypes` method on our data frame:

```
df.dtypes
```

The result makes it clear that the parsing succeeded:

```
tpep_pickup_datetime    datetime64[ns]
tpep_dropoff_datetime   datetime64[ns]
passenger_count         float64
```

```
trip_distance      float64
total_amount       float64
dtype: object
```

If we had not parsed the two timestamp columns, they would be listed as object, which as we've seen indicates that pandas is leaving them as Python objects—most often, as strings.

With these timestamp columns in place, we can create a new timedelta column called `trip_time` by subtracting the pickup time from the dropoff time:

```
df['trip_time'] = df['tpep_dropoff_datetime'] - df['tpep_pickup_d
```

With this timedelta column in place, we can now start to ask questions about our data. For example, how many of the taxi rides in July 2019 took less than 1 minute?

In order to answer this, we'll need to perform a comparison with our `trip_time` column. Now, we could create a timestamp object with `pd.to_timestamp`—but it turns out that pandas takes pity on us, and allows us to compare a timestamp column with a string, by doing the conversion behind the scenes:

```
df['trip_time'] < '1 minute'
```

The above returns a new boolean series, indicating where the trip took less than 1 minute. We can (as always) apply the boolean series to `df` as a mask index, getting only those short trips. Rather than get the entire data frame back, though, I decided to apply the mask only to `df['trip_time']`, and then run `count` on the resulting series:

```
df['trip_time'][df['trip_time'] < '1 minute'].count()
```

I found that 70,212 taxi rides were less than 1 minute long. This seems like a large number of taxi rides to be taking so little time, but New York is a big city. What percentage of rides does this represent? We can find out by dividing this into the total number of rides in our data set:

```
df['trip_time'][df['trip_time'] < '1 minute'].count() / df['trip_
```

Turns out that just over 1 percent of taxi rides take less than a minute. That seems high to me, but maybe people enjoy taking a taxi for one or two blocks when they're in New York.

How much, on average, did people pay for those super short taxi rides? In order to calculate that, I applied the mask index to `total_amount`, and then calculated the mean:

```
df['total_amount'][df['trip_time'] < '1 minute'].mean()
```

The result? More than \$30! The only thing odder about so many people taking 1-minute taxi rides is the fact that they then had to pay more than \$30 for the privilege.

Next, I asked you to find taxi rides that took more than 10 hours. I cannot imagine spending 10 hours in the back of a New York City taxi (or any other taxi, for that matter), but I thought that it might be interesting to find out just how many such rides existed in this data set. Once again, I compared the `trip_time` column to a string:

```
df['trip_time'] > '10 hours'
```

I then applied the resulting boolean series as a mask index, and got all of the long rides, which I then counted:

```
df['trip_time'][df['trip_time'] > '10 hours'].count()
```

I found that there were 16,698 rides that took more than 10 hours in our data set. That seems rather high to me, but maybe it's a reasonable percentage. Let's calculate that:

```
df['trip_time'][df['trip_time'] > '10 hours'].count() / df['trip_
```

Turns out, these consisted only 0.2 percent of all taxi rides. Even so, that means 2 out of every 1,000 taxi rides in New York takes more than 10 hours.

Next, I wanted to group taxi rides into three categories, namely "short," "medium," and "long." In order to do that, I wanted to use `pd.cut`, a method we've already used to perform a similar task. But in order for this to work,

we need to pass a `bins` value to `pd.cut`, one consisting of values that can be compared with our series.

Our intermediate cut points are going to be 10 minutes and 1 hour. Meaning, we'll call "short" trips those that are up to 10 minutes long, "medium" trips between 10 minutes and 1 hour, and "long" trips as longer than 1 hour. However, `pd.cut` won't let us use strings in order to compare with our `timedelta` column. We'll thus need to create a Python list (or pandas series) of `timedelta` objects. I decided to do this with a list comprehension, a standard Python technique but one that might seem a bit odd if you're mostly a data analyst:

```
[pd.to_timedelta(arg)
 for arg in ['0 seconds', '10 minutes', '1 hour', '100 hours']]
```

In short, this list comprehension:

- Iterates over a list of strings
- Converts each string to a `timedelta`
- Returns a list of four `timedelta` objects, based on the strings

We can then pass this list to `pd.cut`:

```
df['trip_time_group'] = pd.cut(df['trip_time'],
                               bins=[pd.to_timedelta(arg)
                                       for arg in ['0 seconds', '10
                                                  minutes', '1 hour', '100 hours']],
                               labels=['short', 'medium', 'long'])
```

Notice that in order to have three labels, we need to have four cut points, or "bins" as they're known here. And while we don't need to provide labels, we definitely want to do so. The result of invoking `pd.cut` is a new series, which we then assign to `df['trip_time_group']`.

With those categories in place, we can then perform a `groupby` query, seeing if there's any substantial difference in the number of passengers between short, medium, and long trips:

```
df.groupby('trip_time_group')['passenger_count'].mean()
```

While short and medium trips both have average passenger counts of 1.5,

there's a slightly larger average (1.7) for longer trips. That might imply that trips of more than 1 hour have more passengers—although it's hard to say just why.

## 9.2.2 Solution

```
filename = '../data/nyc_taxi_2019-07.csv'

df = pd.read_csv(filename,
                  usecols=['tpep_pickup_datetime',
                           'tpep_dropoff_datetime',
                           'trip_distance', 'passenger_count', 'total_amount'],
                  parse_dates=['tpep_pickup_datetime', 'tpep_dropoff_datetime'])

df['trip_time'] = df['tpep_dropoff_datetime'] - df['tpep_pickup_datetime']

df['trip_time'][df['trip_time'] < '1 minute'].count()#3
df['trip_time'][df['trip_time'] < '1 minute'].count() / df['trip_time'][df['total_amount']][df['trip_time'] < '1 minute'].mean()#5

df['trip_time'][df['trip_time'] > '10 hours'].count()#6
df['trip_time'][df['trip_time'] > '10 hours'].count() / df['trip_time'].count()#0.0001

df['trip_time_group'] = pd.cut(df['trip_time'],
                               bins=[pd.to_timedelta(arg)
                                       for arg in ['0 seconds', '10 minutes', '1 hour', '10 hours']],
                               labels=['short', 'medium', 'long'])

df.groupby('trip_time_group')['passenger_count'].mean()#10
```

## 9.2.3 Beyond the exercise

- The data set that we loaded is supposed to be for July 2019. How many trips are not from July 2019? That is, how many records are in the wrong file?
- What was the mean trip time for each number of passengers?
- Load taxi data from July 2019 and 2020. For each year, and then for each number of passengers, what was the mean amount paid?

## 9.3 Exercise 40: Writing dates, reading dates

In the previous exercise, we saw how easily we can read a CSV file into pandas, even if it includes date and time information. Generally speaking, we can pass values to the `parse_dates` keyword argument, indicating which columns should be passed to `pd.to_datetime`, and we don't have to think about it any more. But in the real world, we're often forced to deal with non-standard formats for date and time information. We might be asked to write data using a particular format, or (even more commonly) to read data that doesn't conform to a standard that pandas recognizes.

Fortunately, we can customize the ways in which datetime information is written to disk, as well as how it is parsed when we read it into pandas. In this exercise, we'll practice doing exactly that:

- Load taxi data from January 2019 into a data frame, using only the columns `tpep_pickup_datetime`, `passenger_count`, `trip_distance`, and `total_amount`, making sure to load `tpep_pickup_datetime` as `datetime`.
- Export this data frame to a tab-delimited CSV file. However, the datetime information should be written in the format of `day/month/year HHh:MMm:SSs`. That is:
  - The day should be a two-digit number
  - The month should be a two-digit number
  - The year should be a four-digit number
  - The hours should be a two-digit number, using a 24-hour clock, followed by the letter `h`
  - The minutes should be a two-digit number, followed by the letter `m`
  - The seconds should be a two-digit number, followed by the letter `s`
- Read the CSV file that we just created into a data frame. Make sure to parse the datetime column appropriately.

### 9.3.1 Discussion

This exercise was meant to give you some practice exporting and importing CSV files using alternative date formats. Most of the times that I've had to read (or write) CSV files, dates have been in standard formats, ones that

pandas was able to parse without trouble. But are always those oddball logfiles that need parsing, typically written by custom programs, that use non-standard formats.

The good news is that we can use a custom date-parsing function to handle these odd formats. We can also tell pandas to write datetime columns in a format of our choosing, using the specifiers for `time.strptime`. And indeed, these are the skills that I asked you to practice in this exercise.

I've had occasion to use this functionality in pandas just to translate files from one datetime format to another. In other words, I didn't use pandas for data analysis, but instead as a very fancy date-translation service. That might feel like using a sledgehammer to swat a fly, but it got the job done, while writing almost no code.

We started the exercise by importing New York taxi data from July 2019, including the `tpep_pickup_datetime` column. In order to ensure that `tpep_pickup_datetime` is treated as a datetime column, we specify `parse_dates` to `read_csv`:

```
filename = '../data/nyc_taxi_2019-07.csv'
df = pd.read_csv(filename,
                 usecols=['tpep_pickup_datetime', 'trip_distance',
                          'passenger_count', 'total_amount'],
                 parse_dates=['tpep_pickup_datetime'])
```

With the data frame in place, we can export the data to CSV file containing only these four columns, but with a slightly odd datetime format. In theory, we could create a new column based on `tpep_pickup_datetime`, but with the format we want, and then export that new column to the CSV file. But it turns out that pandas is one step ahead of us here, allowing us to specify the format in which datetime columns will be written by passing a value to the `date_format` parameter.

The format is specified using % signs, using the format specifiers from `time.strftime`, and described here:

[docs.python.org/3/library/time.html#time.strftime](https://docs.python.org/3/library/time.html#time.strftime). Our output can contain any combination of hours, minutes, months, days, time zones, and other elements that we might like. The format that I described wanting in the output

is a bit unusual, in that dates are specified as %d/%m/%Y, meaning two digits for the day, two digits for the month, and four digits for the year, followed by a space character, and then the time in 24-hour format, but with h after the hours, m after the minutes, and s after the seconds. We can specify that as follows:

```
'%d/%m/%Y %Hh:%Mm:%Ss'
```

We can, then, write to our CSV file as follows:

```
df.to_csv('ex40_taxi_07_2019.csv',
          sep='\t',
          columns=['tpep_pickup_datetime', 'passenger_count',
                  'trip_distance', 'total_amount'],
          date_format='%d/%m/%Y %Hh:%Mm:%Ss')
```

In the above code, I wrote to the file named `ex40_taxi_07_2019.csv`, and specified (with the `sep` keyword argument) that we will use tabs to separate the fields. `pandas` uses the `date_format` parameter to indicate how all datetime columns (only `tpep_pickup_datetime`, in our case) should be written.

Given that we're going to be using this special datetime format in a number of places in our program, I thought that it might be wiser to define it as a global string variable, `dt_format`. Then we can access that variable both within our call to `df.to_csv`, and also later on, in our date-parsing function. In such a case, the code will look like this:

```
dt_format='%d/%m/%Y %Hh:%Mm:%Ss'

df.to_csv('ex40_taxi_07_2019.csv',
          sep='\t',
          columns=['tpep_pickup_datetime', 'passenger_count', 'tri
          date_format=dt_format)
```

Once the file was written, I asked you to import it back into `pandas`, into a new data frame, and then to check that the re-loaded `tpep_pickup_datetime` column remains a `datetime` column. Given the odd format we used to write the data, we cannot expect that `pandas` will interpret the column correctly. For that reason, we'll need to use a custom date parser. Such a function should expect to get a single string value, and will be called once for each

row in the incoming data frame. The function should return a `time.time` (or `datetime.datetime`, if you prefer) object, which pandas then puts into the appropriate column of the data frame it creates. In this case, I wrote the function as follows:

```
def parse_weird_format(s):  
    return time.strptime(s, dt_format)
```

Finally, we call `df.read_csv`, specifying not only the filename, separator, and columns, but also which column requires parsing as a date, and (most importantly) the function we want to use to parse those dates:

```
df = pd.read_csv('ex40_taxi_07_2019.csv',  
                sep='\t',  
                usecols=['tpep_pickup_datetime', 'passenger_count', 't',  
                        parse_dates=['tpep_pickup_datetime'],  
                        date_parser=parse_weird_format)
```

Notice that the value we pass to `date_parser` is a function; we're not invoking `parse_weird_format`, but rather `pd.read_csv` is doing so on our behalf, once for each row in the incoming file.

Alternatively, we could have read the value into a string column, and then run `to_timestamp`, passing our string format. That would work just as well, but I prefer to get the parsing done while we're reading the file, rather than in a separate step afterwards.

Another alternative approach would be to use `lambda`, which creates an anonymous function. That would be particularly useful in this case, given that our custom date-parsing function is itself a one-liner, invoking `time.strptime`:

```
df = pd.read_csv('ex40_taxi_07_2019.csv',  
                sep='\t',  
                usecols=['tpep_pickup_datetime', 'passenger_count', 't',  
                        parse_dates=['tpep_pickup_datetime'],  
                        date_parser=lambda s: time.strptime(s, dt_format)
```

`lambda` is somewhat controversial in the general Python world; experienced programmers like the fact that it's short, in-place, and doesn't create a new function that we won't be using again. However, `lambda` is hard for many

inexperienced developers to understand. Passing a function as an argument to another function is hard enough for many to understand; making that function argument anonymous is a lot for many to swallow. However, as you get more experienced with pandas, you might also enjoy starting to use `lambda` in these sorts of one-off situations.

### 9.3.2 Solution

```
filename = '../data/nyc_taxi_2019-07.csv'
df = pd.read_csv(filename, #1
                 usecols=['tpep_pickup_datetime', 'trip_distance',
                          'passenger_count', 'total_amount'],
                 parse_dates=['tpep_pickup_datetime'])

dt_format='%d/%m/%Y %Hh:%Mm:%Ss'#2

df.to_csv('ex40_taxi_07_2019.csv',#3
         sep='\t',
         columns=['tpep_pickup_datetime', 'passenger_count',
                  'trip_distance', 'total_amount'],
         date_format=dt_format)

import time

def parse_weird_format(s): #4
    return time.strptime(s, dt_format)

df = pd.read_csv('ex40_taxi_07_2019.csv', #5
                sep='\t',
                usecols=['tpep_pickup_datetime',
                        'passenger_count', 'trip_distance', 'total_am
                parse_dates=['tpep_pickup_datetime'],
                date_parser=parse_weird_format) #6
```

### 9.3.3 Beyond the exercise

- Export the `tpep_pickup_datetime` date in Unix time—i.e., the number of seconds since 1 January 1970. This will be an integer value.
- Read the data frame from "Beyond 1" back into a data frame. Read the `tpep_pickup_datetime` column into a string column, and use `pd.to_datetime` to convert it into a datetime column.
- Repeat "Beyond 2", but using `date_format` and `lambda`

## Time series

We have already seen how a data frame's index can be an integer or string. But things get really exciting when you set a timestamp column to be your index. In the pandas world, we call that a "time series." And when you create a time series, you can take advantage of a number of useful pandas features.

First, let's create a time series. I created a data frame with the dates and designations of NASA's Apollo program missions, grabbed from Wikipedia:

```
all_dfs = pd.read_html('https://en.wikipedia.org/wiki/Apollo_prog
df = all_dfs[2].copy()[['Date', 'Designation']]
```

I then removed the ending dates from those entries that had one, in order to create a series of Apollo launch dates:

```
df['Date'] = pd.to_datetime(df['Date'].str.replace('(-.+)?', '',
```

I then set the Date column to be my data frame's index:

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

From this point on, df is a "time series." We can see this by looking at df.index:

```
DatetimeIndex(['1966-02-26', '1966-07-05', '1966-08-25', '1967-02
               '1967-11-09', '1968-01-22', '1968-04-04', '1968-10
               '1968-12-21', '1969-03-03', '1969-05-18', '1969-07
               '1969-11-14', '1970-04-11', '1971-01-31', '1971-07
               '1972-04-16', '1972-12-07'],
              dtype='datetime64[ns]', name='Date', freq=None)
```

As you can see, the index contains a number of datetime objects. With this in place, I can retrieve a row on a particular date, just as I would with a normal index:

```
df.loc['1970-04-11']
```

Better yet, I can specify the smaller (i.e., more specific) parts of a date. That is, I can leave out the day, thus retrieving all values in a single month:

```
df.loc['1970-07']
```

Or I can specify just a year, thus getting all missions in that year:

```
df.loc['1971']
```

I can retrieve a set of rows with a slice, by specifying a starting and ending date:

```
df.loc['1968-07-01':'1972-08-31']
```

Perhaps the most interesting and powerful feature of a time series is "resampling." Resampling is similar to a "groupby" query, except that instead of producing one result per value of a categorical column, we get one result per chunk of time, starting at the earliest point in time and ending with the latest one. For example, resampling allows us to retrieve the mean value for every day, or every two weeks, or every six months, or every year of a data frame. For example, I can find out how many missions there were in every six-month period covered by our data set:

```
df.resample('6M').count()
```

If the data is numeric, then you can run other aggregation methods, such as mean and std, as well.

## 9.4 Exercise 41: Oil prices

In this exercise, we're going to work with a CSV file containing oil prices—specifically, West Texas Intermediate oil prices. These prices have been reported and updated daily, at least in our data set, starting on January 2nd 1986, and continue to the present day. (I constructed the CSV file using a Python program downloaded from [github.com/datasets/oil-prices](https://github.com/datasets/oil-prices), which retrieves publicly available oil-price information from the US government.) In this exercise, we'll look at historical oil prices, using the datetime functionality in pandas to make such queries easier. Specifically, I want you to:

- Import the `wti-daily.csv` file into a data frame, in which the Date

column is both treated as a datetime value, and is set to be the index.

- What was the average price of a barrel of oil in June, 1992?
- What was the average price of a barrel of oil in all of 1987?
- What was the average price from September 2003 through July 2014?
- Show the price of oil at the end of each quarter in the data set.
- For each year in the data set, show the average price.
- On which date were oil prices the highest? When were they the lowest?

### 9.4.1 Discussion

We've already seen that by using the `dt` accessor, we can retrieve various parts of a datetime column. With that tool at our disposal, we can query our data in all sorts of ways. But we've also seen that certain queries can be easier to read and write when we change the index. This is particularly true when we set the index to be a datetime value; pandas provides us with all sorts of useful functionality. In this exercise, we explored a number of these functions while looking at historical oil prices.

The first task, as always, was to load the data from a CSV file into a data frame. In this case, the CSV file contained only two columns, named `Date` and `Price`. In loading the CSV file into memory, I asked pandas to treat the `Date` column as (not surprisingly) a datetime value. I also asked it to make that column the index, via the `index_col` parameter:

```
filename = '../data/wti-daily.csv'

df = pd.read_csv(filename,
                 parse_dates=['Date'],
                 index_col=['Date'])
```

With that in place, I was able to start making some queries. For starters, I wanted to find out the average price of oil during the month of June, 1992. As usual, I can retrieve items from the data frame by using the `loc` accessor, along with the index value that's of interest to me. But because it's a time series, I can provide a subset of the date, removing more specific parts in order to match a larger number of rows. I could thus say:

```
df.loc['1992-06-15']
```

and get back the row for June 15, 1992. (If there were more than one row, then I would get all of them back. But we know that each date in this data set is unique.) However, I'm interested in all days in June, 1992. I can thus say:

```
df.loc['1992-06']
```

By leaving off the day, pandas matches and retrieves all rows in which the date is somewhere in June of 1992. To get the mean price during that period, we can say:

```
df.loc['1992-06'].mean()
```

I get a result of just over 22.38.

I similarly asked you to find the mean price during 1987. Just as I can leave off the day to find all rows from a particular year and month, I can leave off the day and month to get all rows from a particular year:

```
df.loc['1987'].mean()
```

This retrieves all rows with a year of 1987. We then run mean on the Price column, which returns a number just over \$19.20.

Next, I asked you to find the average price from September 2003 through July 2014. The easiest way to do this, when we have a time series, is to use a slice. Normally, Python slices are specified as a starting value, and then one past the final value. For example, if I have a sequence (string, list, or tuple) named `s` and request `s[10:20]`, that retrieves the values from the index 10 up to (but not including) 20.

Slices with a time series are similar, but as with other non-numeric indexes in pandas, we **include** the end of the slice. I can specify the date on which I want to start, and also the date on which I want to end:

```
df.loc['2003-09':'2014-07']
```

The above retrieves all rows from `df` starting from September 1st, 2003, and going through July 31st, 2014. We can then run mean on the values we get back:

```
df.loc['2003-09':'2014-07'].mean()
```

This returns a value of just over 76.45.

Next, I asked you to find the price of oil at the end of each quarter in the data set. Now, the end of a quarter would theoretically be the final days in March, June, September, and December. But that's often not the case because of weekends and holidays. We could get around this by looking up each of these days on a calendar, but that gets tricky, and is almost certainly error prone.

Fortunately, pandas makes this easy to do, with the `is_quarter_end` attribute on the `dt` accessor for `datetime` series. But in our case, the `datetime` values aren't exactly in a series; they're on our index. How can we invoke `is_quarter_end` on our `datetime` index?

It turns out that we can invoke it directly on the index, getting a boolean series back:

```
df.index.is_quarter_end
```

This boolean series can then be applied as a mask index to `df`:

```
df[df.index.is_quarter_end]
```

The result is a one-column data frame whose index values are the final days of each quarter, regardless of whether it's the 30th or 31st of the month in question.

Finally, I asked you to find the mean price of oil for each year in our data set. This is most easily accomplished by using `resample`, which is a kind of `groupby` but for time series: It lets run an aggregation method (e.g., `mean`) for all of the values in a given time period. If the time period doesn't exist in the data frame, or is cut off, it'll still show up, to ensure that we have all of the periods from start to finish.

When we run `resample`, we tell it what time-period granularity we'll want, giving a number and a letter representing the measurement. For example, we can run our aggregation method on a weekly basis with `1W`, or on a bimonthly basis with `2M`. In this exercise, I asked to see annual average prices, which

meant specifying 1Y. The resulting query is thus:

```
df.resample('1Y').mean()
```

The result from a `resample` query will always be a data frame, in which the index contains the values from the end of each period. The index in my result thus starts at 1986-12-31, and goes through 2021-12-31. Note that even if we only have partial values for a year, we'll get the average amount for that year.

Finally, I asked you to determine on which dates we had the historically highest and lowest oil prices. There are numerous ways to accomplish this, but I thought it was easiest to sort the values in the `Price` column—and then retrieve the first and last values from the resulting sorted series. Remember that we can retrieve more than one value by passing a list of indexes to `loc` or `iloc`, and that if we use `iloc` (which retrieves by position), we can ask for index 0 (the first item) and -1 (the final item):

```
df['Price'].sort_values().iloc[[0, -1]]
```

You might be surprised that the lowest price of oil in this data set -36.98, meaning that you could get paid to accept a barrel of oil. If this sounds a bit odd, then you're right; it was the result of a dramatic drop in oil demand at the start of the covid-19 pandemic. There wasn't enough storage space for the oil that had already been extracted from the ground, resulting in this bizarre situation. Look it up—it's just one of the many economic oddities of the pandemic.

## 9.4.2 Solution

```
filename = '../data/wti-daily.csv'

df = pd.read_csv(filename,
                 parse_dates=['Date'],
                 index_col=['Date'])

df.loc['1992-06'].mean()
df.loc['1987'].mean()
df.loc['2003-09':'2014-07'].mean()
df[df.index.is_quarter_end]
df.resample('1Y').mean()
df['Price'].sort_values().iloc[[0, -1]]
```

### 9.4.3 Beyond the exercise

- Use `resample` to find, for each quarter, the mean and standard deviations in price.
- In which quarter did we see the biggest increase in mean price from the previous quarter?
- What was the biggest percentage increase in oil prices across quarters?

## 9.5 Exercise 42: Best tippers

We've looked at New York taxi data a number of times already, but now we'll use our time-related knowledge to look at them once more. This time, we'll try to understand when people tip their taxi drivers more generously. (If you're not from the United States, then you might not be familiar with the custom of tipping, often 15 or 20%, in addition to whatever a taxi meter says you officially need to pay. In many other countries, this practice is unexpected, rare, or even illegal.) In particular, I'd like you to:

- Import the taxi info from both January and July 2019. Include the following columns: `tpep_pickup_datetime`, `passenger_count`, `trip_distance`, `fare_amount`, `extra`, `mta_tax`, `tip_amount`, `tolls_amount`, `improvement_surcharge`, `total_amount`, and `congestion_surcharge`.
- Create a new column, `pre_tip_amount`, with all of the payment columns except for `total_amount` and `tip_amount`.
- Create a new column, `tip_percentage`, showing the percent of `pre_tip_amount` that the tip was.
- How many times did people tip more than the pre-tip amount?
- What was the overall tip percentage, across all trips in the data set?
- On which day of the week do people tip the greatest percentage of the fare, on average?
- At which hour do people tip the greatest percentage?
- Do people typically tip more in January or July?
- What was the 1-day period in our data set when people tipped the greatest percentage?

## 9.5.1 Discussion

In this exercise, we asked the same question—when do people tip taxi drivers the most?—in a number of different ways. All of them made use of the extensive support for dates and times that pandas offers.

For starters, we loaded the data from both January and July 2019. As I've done before, I thought it would be best to use a list comprehension along with `pd.read_csv`. This resulted in the creation of a list of data frames, one which I was able to turn into a single data frame with `pd.concat`:

```
filenames = ['../data/nyc_taxi_2019-01.csv', '../data/nyc_taxi_2019-07.csv']
all_dfs = [pd.read_csv(one_filename,
                      usecols=['tpep_pickup_datetime', 'passenger_count', 'tpep_dropoff_datetime',
                               'fare_amount', 'extra', 'mta_tax', 'tip_amount',
                               'improvement_surcharge', 'total_amount', 'congestion_surcharge'],
                      parse_dates=['tpep_pickup_datetime'])
           for one_filename in filenames]

df = pd.concat(all_dfs)
```

I asked you to include a large number of columns when creating the data frame, so that we could calculate the tip percentage more accurately. I considered not asking you to specify `usecols`, but rather to read all of the data anyway—but as tempting as it might be to do that, it's not a good habit to get into. You really want to specify the columns you want in your data frame, otherwise, you'll find yourself running out of memory when you work with large data sets.

With our data frame in place, I wanted to calculate the pre-tip amount—that is, the amount on which the tip would be based—for each ride. It's not always obvious what should (and shouldn't) be included in the tip. For example, do we include tolls for bridges and tunnels in our calculation? How about the surcharge that's sometimes added because the streets of New York are extra congested during those hours? For our purposes, I included all of these fees and charges.

I thus asked you to create a new column, `pre_tip_amount`, which would be the sum of six columns. How can we do that?

One possibility is to explicitly name those columns, and add them together:

```
df['pre_tip_amount'] = df['fare_amount'] + df['extra'] + df['mta_
```

This will certainly work, but it seems a bit wordy. Perhaps there's a way for us to name the columns and sum them together? The `sum` method would seem to be a perfect way to do this, except that it sums the rows, rather than the columns. But wait! Many pandas methods allow us to specify the axis on which they run—and sure enough, `sum` is one of them. We can thus sum our selected columns by specifying `axis='columns'`:

```
df['pre_tip_amount'] = df[['fare_amount', 'extra', 'mta_tax', 'to  
                        'improvement_surcharge', 'congestion_s
```

Notice that we select our six columns with a list of strings. We then run `sum` on these columns, producing a new pandas series. We assign this series back to `df['pre_tip_amount']`.

With that in hand, we're now ready to create another column, `tip_percentage`, which contains the percentage of the pre-tip charge that the user added as a tip:

```
df['tip_percentage'] = df['tip_amount'] / df['pre_tip_amount']
```

Our data frame now has all of the information we need in order to start answering the questions we'll want to pose about tipping in New York taxis. For starters, what was the mean tipping rate across all taxi rides in our data set? We can find that out by running the `mean` method on our `tip_percentage` column:

```
df['tip_percentage'].mean()
```

I found that it was 13%. That seems a bit low to me, so perhaps I'm calculating the pre-tip base amount differently than others do. But maybe the data set is a bit more complex than a straight percentage. For example, does anyone tip more than 100%? We can find out:

```
(df['tip_percentage'] > 1).value_counts()
```

Here, I use `value_counts` to find how many people tipped more than 100%

of the pre-tip amount. By applying `value_counts` to a boolean series, I'm able to find out how often the `True` value was returned, meaning how often my condition was met.

The number of people giving above-and-beyond tips isn't overwhelming—7,831 out of 13,969,564 rides. But it's not zero, either, which came as a bit of a surprise to me. However, this number would skew the average tip upward. Perhaps there are people who aren't tipping at all, which would skew things downward? Let's take a look, calculating the percentage of riders who don't tip at all:

```
(df['tip_percentage'] == 0).value_counts(normalize=True)
```

Once again, I use `value_counts`—but this time, I pass it `normalize=True`, to get a percentage answer. And the results were a bit surprising, at least to me—about 32 percent of taxi riders in New York don't tip at all! This almost certainly has an effect on the mean tipping rate.

Next, I was curious to know whether people tip more on any particular day of the week. In order to do this, we'll combine `groupby` with the `dt` accessor's `day_of_week` attribute, which returns the integer for the day of the week, with Monday being 0 and Sunday being 6. You might think that we need to define a new `day_of_week` column in our data frame, so that we can run a `groupby` on it. But no, the pandas developers make it possible to run a `groupby` on not only a column, but on the result we get back from `dt.day_of_week`:

```
df.groupby(df['tpep_pickup_datetime'].dt.day_of_week)
```

For each day of the week, we want to get the mean tip percentage. We thus run the following query:

```
df.groupby(df['tpep_pickup_datetime'].dt.day_of_week)['tip_perce
```

This gives us values, one for each day of the week. Just to make sure that I get the right data, I then want to sort the resulting values, from highest to lowest:

```
df.groupby(df['tpep_pickup_datetime'].dt.day_of_week)['tip_perce
```

Much to my surprise, the tipping percentages aren't that different from one another. I was sure, before analyzing this data, that people tip more on weekends, but the data doesn't support that. On the contrary, it shows that people tip the least on Fridays and Saturdays, and the most on Tuesdays and Wednesdays. However, the difference isn't that great, so I'm not sure if we can really draw significant conclusions. Certainly, if I were a taxi driver deciding which shifts to take, the tip amount on a given day wouldn't make much difference. (And besides, one third of your passengers aren't going to tip anything, right?)

But maybe the hour of the day does make a difference? That is, perhaps people tip better in the mornings, or in the afternoons. I thus asked you to create such a query, to find out at which hour of the day people tip the most, on average:

```
df.groupby(df['tpep_pickup_datetime'].dt.hour)['tip_percentage'].
```

The query in this case is quite similar to the previous one. Here, however, we did get some more interesting results; people tip about 11 percent early in the morning (between 3 and 6 a.m.), and nearly 14 percent at night (from 8-11 p.m.). You see similar, if slightly lower, rates from 7-9 a.m.—so if you're unsure whether to take the 5 a.m. or 9 a.m. slot as a taxi driver, I'd suggest, on average tips alone, choosing the latter.

Let's ask another question, one which our data set can help us answer: Do people tip more during the winter or the summer? (Or is there no difference?) We have data from both January and July, which should give us some useful insights. We can say:

```
df.groupby(df['tpep_pickup_datetime'].dt.month)['tip_percentage']
```

We see that the highest tips (of 20 percent, on average) were given in May, followed by August, March, and September, respectively.

But wait a moment: Our data set is supposed to contain data from January and July. How did other months get in there? The answer, of course, is that no data set is completely clean. Whether the dates are wrong, were reported late, or were otherwise scrambled along the way, our data contains

information from other months. If we only compare January with July from this data set, we see a slight difference between the months, with tipping in January at 13.7 percent, but in July at 12.1 percent. Whether that's because of summer tourists (who might, I'm just guessing, tip more), or just people feeling a bit more open with their cash during the summer months, I'm not sure.

Next, I asked what one-day period in our data set had the highest average percentage of tipping. This is one of those problems that's most easily solved with a time series, meaning that we use a datetime value as the index:

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

With that in place, we can now use `resample` with an argument of `1D` (i.e., one day) to find the day on which people tipped the most. First, we find the mean tipping percentage for each day in the time series:

```
df.resample('1D')['tip_percentage'].mean()
```

That works, but I'd like to sort these values, so that I can find the highest-tipping day. I do this by running `sort_values` on our results, and then listing only the top 10 dates:

```
df.resample('1D')['tip_percentage'].mean().sort_values(ascending=
```

The results include zero days from either January or July. I'm going to try this again, but will first get rid of dates that aren't in January or July:

```
df = pd.concat([df['2019-01-01':'2019-01-31'],  
               df['2019-07-01':'2019-07-31']])  
df.resample('1D')['tip_percentage'].mean().sort_values(ascending=
```

Having cleaned the data from non-January/July rows, we can see that all 10 of the highest-tipping days were all in January. Which means that at least in our data sample, people are more likely to tip better in the winter than in the summer. We can double-check that this is the case by resampling at a one-month granularity:

```
df.resample('1M')['tip_percentage'].mean().dropna()
```

Because we only have two months of data, but they're in January and July, using `resample` means that we'll get NaN values for February, March, April, May, and June. I thus removed those with `dropna`. And we see that the average tipping rate in January was 13.7 percent, whereas in July it was 12.1 percent—a finding that I hadn't anticipated.

## 9.5.2 Solution

```
filenames = ['./data/nyc_taxi_2019-01.csv', './data/nyc_taxi_20
all_dfs = [pd.read_csv(one_filename, #1
                    usecols=['tpep_pickup_datetime', 'passenger_count', 't
                    'fare_amount', 'extra', 'mta_tax', 'tip_amount',
                    'improvement_surcharge', 'total_amount', 'conge
                    parse_dates=['tpep_pickup_datetime'])
            for one_filename in filenames] #2

df = pd.concat(all_dfs) #3

df['pre_tip_amount'] = df[['fare_amount', 'extra', 'mta_tax', 'to
                    'improvement_surcharge', 'congestion_s

df['tip_percentage'] = df['tip_amount'] / df['pre_tip_amount'] #5

df['tip_percentage'].mean() #6

(df['tip_percentage'] > 1).value_counts() #7

(df['tip_percentage'] == 0).value_counts(normalize=True)#8

df.groupby(df['tpep_pickup_datetime'].dt.day_of_week)['tip_perce
df.groupby(df['tpep_pickup_datetime'].dt.hour)['tip_percentage'].

df.groupby(df['tpep_pickup_datetime'].dt.month)['tip_percentage']

df = df.set_index('tpep_pickup_datetime')#12
df.resample('1D')['tip_percentage'].mean().sort_values(ascending=

df = pd.concat([df['2019-01-01':'2019-01-31'],
                df['2019-07-01':'2019-07-31']])#14

df.resample('1D')['tip_percentage'].mean().sort_values(ascending=
```

## 9.5.3 Beyond the exercise

- We saw that 32 percent of riders don't tip at all. Of those who **do**, what percentage do they tip, on average?
- How many of the rides in our data set, supposedly from January and July 2019, are from outside of those dates?
- Looking only at dates in January and July, on what week did passengers tip the greatest percentage?

## 9.6 Summary

In this chapter, we explored various ways that pandas lets us examine data that includes a date-and-time component. We saw how to read datetime information into a data frame, how to extract datetime information from an existing column, how to break such a column apart, and even how to interpret odd datetime formats. We also saw how to create and work with a "time series," a data frame in which a datetime column serves as our index, and how to query it in various ways—including resampling, letting us run aggregation methods over particular time periods.

# 10 Visualization

Data analysis, as you've seen throughout this book, is largely about numbers. A typical pandas data frame contains columns and rows full of numbers, and data analysis involves lots of mathematical methods and statistical techniques. That's fine, except that we humans are typically bad at understand large collections of numbers. We're generally much better at understanding visual depictions of numbers, especially if we're trying to understand relationships among our data. So while we often think of visualization as a way to explain technical ideas in simple terms to non-experts, the fact is that visualization can also be helpful for the experts working on a problem. Seeing a chart or graph can help us to put the numbers in perspective, improve our understanding of a problem we're working on, and thus inform the very analysis that created the visualization.

The 900-pound gorilla in the world of Python data visualization is Matplotlib. There's no doubt that Matplotlib is powerful—but it's also overwhelming to many people. Fortunately, pandas provides a visualization API that allows us to create plots from our data without having to use Matplotlib explicitly. We thus get the best of both worlds—the ability to plot information in our data frame, without having to learn too much about Matplotlib's API. However, if and when you need more power, Matplotlib is there, under the hood, available to anyone who wants to use it.

At the end of this chapter, we'll also spend some time looking at Seaborn, an alternative to Matplotlib that you might want to explore. There are a number of alternatives to Matplotlib, with some (like Seaborn) built on top of it, providing a better, cleaner, and more modern API. Others are full-blown alternatives to Matplotlib. It's worth learning what your options are, to find a system with which you feel comfortable. I've grown not only to like Seaborn's API, but also its ability to create attractive plots with little or not customization.

This chapter will also provide you with the opportunity to explore one of the nicest features of the Jupyter notebook, the fact that it keeps images inline.

Whether it's on your own or by exploring the notebooks that I've prepared while writing this book, I strongly encourage you to experiment with Jupyter's plotting capabilities. The ability to have data, code, and plots in the same document is a game changer for many projects, making it possible for data scientists to both share information and get input from less technical colleagues.

## 10.1 Useful references

Table 10.1. What you need to know

Concept	What is it?	Example
<code>pd.read_csv</code>	Returns a new data frame based on CSV input	<code>df = df.read_csv('myfile</code>
<code>df.groupby</code>	Allows us to invoke one or more aggregate methods for each value in a particular column.	<code>df.groupby('year')</code>
<code>df.loc</code>	Retrieve selected rows and columns	<code>df.loc[:, 'passenger_cou df['passenger_count']</code>

<code>df.plot</code>	Entry to the plotting system	<code>df.plot.box()</code>
<code>s.quantile</code>	Get the value at a particular percentage of the values	<code>s.quantile(0.25)</code>
<code>df.join</code>	Join two data frames together based on their indexes	<code>df.join(other_df)</code>
<code>pandas.plotting.scatter_matrix</code>	Create scatter plots comparing every pair of numeric columns	<code>pandas.plotting.scatter_</code>
Matplotlib	Python library for plotting	<code>import matplotlib.pyplot plt</code>

	data	
Seaborn	Python library for plotting data	<code>import seaborn as sns</code>
<code>df.reset_index</code>	Get a data frame identical to our current one, but with a new numeric index starting at 0	<code>df.reset_index(drop=True)</code>
<code>pd.concat</code>	Return a list of data frames, combined, as a single, new data frame	<code>df = pd.concat(df1, df2)</code>

## 10.2 Exercise 43: Cities

We've already worked with the JSON file describing the 1,000 largest cities in the United States, back in Exercise 20. In this exercise, we'll look at the

same file—but instead of printing the analysis as a bunch of numbers, we'll visualize some of the most interesting numbers and trends in the file. Specifically, I want you to:

- Load data from `cities.json` into a data frame,
- Create a bar plot showing how many of the top 1,000 cities are in each state. Order the plot from smallest (on the left) to greatest.
- Create a bar plot showing growth in Pennsylvania cities, sorted from lowest (on the left) to highest.
- Create a pie plot from all Massachusetts cities, so that we can see the proportion that each city contributes to the overall population.
- Create a scatter plot of the cities, with `x` being longitude and `y` latitude. What does the resulting plot look like?

### 10.2.1 Discussion

In this exercise, we again loaded data from `cities.json` into a data frame. But this time, the results of our analysis were visual and graphic. Matplotlib offers a wide variety of plotting formats, and I used this exercise as a way to explore a number of them, using different techniques to understand our data in a variety of ways. As you saw, visualization isn't just about choosing a type of plot; you often need to clean, arrange, and modify the data before you can do so.

First, I asked you to create a bar plot showing how many of the top 1,000 cities in the United States are in each state. The data frame that we created from the JSON has several columns, one of which is `state`. We'll use that column, along with a call to `groupby`, to find the number of cities per state:

```
df.groupby('state').count()
```

This works, but it gives me a result for every column in the data frame. Since I'm only interested in the number of cities, I can choose a single column—in this case, the `city` column:

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

With that in place, I can create a bar plot. But wait: The question asked for

the bar to be sorted from the smallest value to the largest. This means that before producing the plot, I'll need to sort the values in the series that was returned by my groupby call. Fortunately, sorting a series is easily done with `sort_values`:

```
df.groupby('state')['city'].count().sort_values()
```

With that in place, I can now produce my bar plot:

```
df.groupby('state')['city'].count().sort_values().plot.bar()
```



#### Note

Another way to invoke this plot would be to invoke `plot` as a function, passing `kind='bar'` as a keyword argument. I prefer the other syntax, but either is considered standard and acceptable.

This works, but with 50 states (plus Washington, DC), we end up with a plot that's a bit small. I thus pass the `figsize` keyword argument to `bar`, which is in turn passed along to the Matplotlib backend. By giving `figsize` a value of `(10, 10)`, we can set it to be a 10-inch by 10-inch square. It's probably not particularly surprising that California has the greatest number of large cities, but the sheer number (and thus very tall bar in our plot) was still striking to me when producing this plot.

Next, I asked you to create a bar plot showing growth in Pennsylvania cities, sorted from lowest to highest. For this task, we needed to take data from the `growth_from_2000_to_2013` column, along with the `city` column, all from rows in which `state` was equal to `'Pennsylvania'`. I decided that it would be easiest to turn these rows and columns into a separate, smaller data frame, using `df.loc`:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

As we've seen on many occasions, I selected rows in which the `state` was equal to `'Pennsylvania'`, and then the two columns that were of interest. I then decided that since I'll want to show the city names in my plot's x index, I should make it the index of the data frame:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

At this point, it would be nice to produce the plot. But there's a problem: The growth is a string, ending with a '%' sign. If we want to plot it, we'll need to turn it into a number. How can we do that?

We could use the `str` accessor to run a method on our string. But before we can do that, we need to turn our data frame into a series. That's because `str` only works on a series. Fortunately, the index from a data frame remains when we extract one column as a series:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

With our data now in a series, we can remove the '%' in a variety of ways. I decided to use `str.replace`, turning all occurrences of '%' into the empty string, ''. But we could also have used a slice to keep all but the final character, or `str.rstrip` to remove '%' from the right side. Using `str.replace`, we end up with the following code:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

The result is still a series of strings. However, these strings can all be turned into floating-point values using `astype`:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

We now have every city in Pennsylvania, along with its growth percentage. We could plot it, but before doing so, I asked you to sort the values from lowest to highest. Once again, we invoke `sort_values`:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

And with that in place, I once again create a bar plot, setting a size of (10, 10) to see it more easily in my notebook:

```
df.loc[df['state']=='Pennsylvania', ['city', 'growth_from_2000_to_
```

Next, I asked you to find all of the cities in Massachusetts, and to create a pie plot with all of these cities. This will allow us to see what proportion of the urban population of Massachusetts lives in each city. Remember that a pie

plot takes all of the values, sums them together, and then produces a pie "slice" of the proportion that item has of the total.

We'll first need to get names and populations of cities in Massachusetts. We can do that using the following query:

```
df.loc[df['state'] == 'Massachusetts', ['city', 'population']].set
```

This is quite similar to what we did for Pennsylvania: We retrieved only two columns (`city` and `population`) from the data frame, and only for those rows in which the state was Massachusetts. We then set the index of our data frame to be `city`, and then retrieved the only remaining column, `population`, as a series.

Next, we drew a pie plot based on this data, giving it a size of 10 inches by 10 inches:

```
df.loc[df['state'] == 'Massachusetts', ['city', 'population']].set
```

Sure enough, we see that Massachusetts has many different cities—but of the urban population in the state, Boston clearly dominates, followed distantly by Worcester and Springfield.

Finally, I asked you to create a scatter plot with the longitude and latitude of the 1,000 cities in the data frame. We can do that by invoking `plot.scatter` on the data frame, indicating which column should be used for the `x` axis, and which should be used for the `y` axis:

```
df.plot.scatter(x='longitude', y='latitude')
```

What does the scatter plot look like? Well, we're plotting the 1,000 most populous cities in the United States, which means that the plot will look like... a map of the United States, at least the most densely populated areas.

## 10.2.2 Solution

```
filename = '../data/cities.json'  
df = pd.read_json(filename)#1
```

```
df.groupby('state')['city'].count().sort_values().plot.bar(figsize
```

```
df.loc[df['state']=='Pennsylvania',#3
       ['city','growth_from_2000_to_2013']].#4
       set_index('city')['growth_from_2000_to_2013'].str.replace(
       astype(np.float16).sort_values().plot.bar(figsize=(10,10))

df.loc[df['state'] == 'Massachusetts', ['city','population']].#7
       set_index('city')['population'].plot.pie(figsize=(10,10))#8

df.plot.scatter(x='longitude', y='latitude')#9
```

### 10.2.3 Beyond the exercise

Now that you've gotten your feet wet with visualization, let's create some more plots:

- Create a histogram of the growth rates among cities in both Texas and Michigan.
- Create a histogram of the growth rates among cities in both Texas and California.
- Create a bar plot from the average growth per state.

#### Box-and-whisker plots

When I took introductory statistics in graduate school, the professor started to tell us about plots. I was wondering why he felt the need to explain plots that we had seen since middle school—line plots, bar plots, and even pie plots. But then he got to box plots, more formally known as "box and whisker plots," and I was intrigued.

We frequently use the `describe` method to describe our data. The `describe` method includes the "Tukey five-number summary"—minimum, 0.25 quartile, median, 0.75 quartile, and maximum—along with the mean and standard deviation, which together help us to understand our data.

The "Tukey" in this name refers to John Tukey, a famous mathematician and statistician. It turns out that Tukey didn't only develop the five-number summary, but also a graphical depiction of that summary—the boxplot. (He also invented the words "bit," for "binary digit," and "software," which ... well, if you're reading this book, then you probably know what software is.)

A boxplot shows us this five-figure summary, but in graphical form:

(And yes, I'll have an image and label it!)

- The central "box" in the boxplot has three parts:
  - The top of the box indicates the 75% value
  - The middle line, often highlighted in a different color, indicates the median, the 50% value
  - The bottom of the box indicates the 25% value

Extending above and below the box are two lines, sometimes known as "whiskers." The top whisker ends at the maximum value, and the bottom whisker ends at the minimum value.

Thus, at a glance, we get a graphical depiction of the five-figure summary.

However, a boxplot will often have circles above and below the whiskers. These represent the outliers, defined in the case of our boxplots to be  $1.5 * \text{IQR}$  below the first quartile (25% mark) or  $1.5 * \text{IQR}$  above the third quartile (75% mark).

Boxplots allow us, at a glance, to better understand our data. They can be especially useful when it comes to comparing data sets; we can quickly see if they're on the same scale, and whether (and where) they overlap.

We can create boxplots in pandas using `plot.box` on a data frame. In the simplest case, we can say:

```
df.plot.box()
```

This will create a separate plot for each of the columns in the data frame `df`. This can be particularly useful when creating machine-learning models, when having all of the data in the same range increases the model's accuracy.

Note that nowhere in the boxplot do we see the mean value. I personally think that's a bit of a shame, because the mean can also be a useful measure, imperfect though it might be.

## 10.3 Exercise 44: Boxplotting Weather

One of the phrases I often use when teach data analytics is that you need to "know your data." And one of the best ways to know your data is with a box plot. In this exercise, we'll use box plots to understand the weather during the winter of 2018-2019, using data in three different US cities. We'll start with Chicago, and then add Los Angeles and Boston to emphasize the differences between these locations. (And to assure Chicago residents that yes, their winters really are that cold.)

- Load the weather data for Chicago. We only care about three columns: `date_time`, `min temp`, and `max temp`. Make `date_time` the index, and set the names of the min and max temp columns to "mintemp" and "maxtemp".
- Create a boxplot of Chicago's minimum temperatures during this period.
- Find the values that are represented as dots on that boxplot
- Create a boxplot of Chicago's minimum temperatures in February.
- Create a side-by-side boxplot of Chicago's minimum and temperatures in February and March
- Now read data from Los Angeles and Boston in, as well. Create a single data frame with data from all three cities, along with a new "city" column containing the name of the city. # Get descriptive statistics for mintemp and maxtemp, grouped by city
- Create side-by-side boxplots, showing minimum and maximum temperatures for each three cities

### 10.3.1 Discussion

In this exercise, we combined techniques we've previously seen—specifically, using `read_csv` with a variety of parameters, combining several CSV files into a single data frame, and the use of a `datetime` column as an index. But the main point of this exercise was to create a number of different boxplots, and in so doing to better understand the shape and nature of our data.

First, I asked you to load Chicago weather into a data frame, using the `date_time` column as both the index and as of type `datetime`. I also asked

you to load the columns with the minimum and maximum temperatures found on each day. I did that using the following code:

```
filename = '../data/chicago,il.csv'
df = pd.read_csv(filename,
                 usecols=[0, 1, 2],
                 header=0,
                 names=['date_time', 'mintemp', 'maxtemp'],
                 parse_dates=['date_time'],
                 index_col=['date_time'])
```

We've used each of these options to `read_csv` in the past, but here I used many all at once. For starters, I indicated that I was interested in only the first three columns. In previous exercises, I've often referred to these columns by name, using the names provided by the index. But here, I referred to the columns by number. That's because I wanted to give them names of my own, specified in the `names` parameter. I thus chose them by number, and renamed them in `names`. I also indicated that `date_time` should be parsed as a `datetime` column, and that it should furthermore be used as the index of the data frame. Finally, just to be on the safe side, I passed `header=0`, to indicate that the first row of the file contains headers, and thus shouldn't be treated as data.

At the conclusion of this process, I ended up with a data frame with 728 rows and two columns. The values started at midnight on December 12th, 2018, and ended at 9 p.m. on March 11th, 2019, with new measures taken every three hours.

I then asked you to create a boxplot for the minimum temperatures found in Chicago throughout the period in the data frame. We can do this by running the following code:

```
df['mintemp'].plot.box()
```

A boxplot is supposed to visualize the "five-number summary": minimum, 0.25, median, 0.75, and maximum. The result shows us that most of the temperatures were between -20 and 5 degrees Celsius. However, we also see a number of circles at the bottom of the plot, indicating outlier values. In the pandas implementation of boxplots, outliers are defined as those 2.5 standard deviations above or below the mean. (NOTE: I have to double-check whether

it's  $2.5 \times \text{std}$  or  $1.5 \times \text{IQR}$  below/above 1Q/3Q.) Just to double check that the plot is showing them correctly, I asked you to find those values:

```
df.loc[df['mintemp'] < df['mintemp'].mean() - (df['mintemp'].std(
```

Sure enough, we see a number of temperature readings (on January 30th and 31st) in which the temperature was -27 and -28 degrees Celsius—not only cold, but unusually cold, even for a Chicago winter. Our boxplot was thus right to show them as outliers.

Next, I asked you to create a boxplot for Chicago's minimum temperatures in February. I solved this as follows:

```
df.loc['01-Feb-2019':'28-Feb-2019', 'mintemp'].plot.box()
```

My row selector was the slice from February 1st through February 28th. Here, I took advantage of the fact that our data frame's index contains date and time values, and that we can always use a slice to retrieve rows. I chose the `mintemp` column, and then fed the resulting 1-column data frame to `plot.box`. We can see that the median temperature during February 2019 was -5 degrees Celsius, which does indeed sound right for a Chicago winter.

Next, I asked you to create boxplots for both minimum and maximum temperatures in both February and March. Here, I again used a slice to select the appropriate rows, stretching from February 1st through March 30th:

```
df.loc['01-Feb-2019':'30-Mar-2019', ['mintemp', 'maxtemp']].plot.b
```

Once again, I selected rows using a slice. But the column selector needed to be a list of strings, the names of the columns that I wanted to plot. I then passed these to `plot.box`, and got two boxplots—displayed on the same scale, next to one another.

Having experienced, if only on paper, the cold Chicago winter, I thought that it would be nice to add data from two other cities. I thus asked you to read data from Los Angeles and Boston as well, creating a single data frame from all three of the CSV files. In order to distinguish data from the various cities, I asked you to add a "city" column with the city's name as you read them in. Since `df` already contains information for Chicago, I had to set that value

right away:

```
df['city'] = 'Chicago'
```

To load the other data, I decided to use a for loop—typical in day-to-day Python programming, but unusual in pandas. Here, the loop wasn't running over a series or data frame, but rather over a list of filenames containing city data:

```
for city_stem in ['los+angeles,ca', 'boston,ma']:
    new_df = pd.read_csv(f'../data/{city_stem}.csv',
                        usecols=[0, 1,2],
                        header=0,
                        names=['date_time', 'maxtemp', 'mintemp'],
                        parse_dates=['date_time'],
                        index_col=['date_time'])
    new_df['city'] = city_stem.split(',')[0].replace('+', ' ').title
    df = pd.concat([df, new_df])
```

Let's break down what I did here:

- First, I set up a list with the filenames (minus the 'csv' suffix) over which I wanted to run
- I used a for loop to iterate over those filenames.
- I re-used the `read_csv` call that we used earlier, passing the complete filename.
- As before, I selected specific columns, indicated that `date_time` should be parsed as a `datetime`, and should be set to the index.
- I then added a value to `city` for each of the loaded cities, using a bunch of string methods to convert `city_stem` into a useful string:
  - I used `str.split` on `city_stem`, getting a list—from which I took the initial part
  - I replaced the character '+' with a space, ' '
  - I invoked `str.title`, capitalizing each word

Finally, I used `pd.concat` to add the new data frame to the existing one. The end result is a single data frame with weather data from all three cities, and with the `city` column indicating the source of the data.

With this data loaded, I then asked you to get descriptive statistics for `mintemp` and `maxtemp`, grouped by city:

```
df.groupby('city')[['mintemp', 'maxtemp']].describe()
```

The data frame we get back has three rows, one for each city. The columns are in a multi-index, with all of the measurements for `mintemp` and then all of the measurements for `maxtemp`. But while these details might all be interesting and useful, they're not quite as compelling as a boxplot. I thus asked you to create a boxplot showing minimum and maximum temperatures for all three cities, grouped together. I solved it as follows:

```
df.plot.box(column=['mintemp', 'maxtemp'], by='city')
```

This produced two side-by-side boxplots, one for `mintemp` and the second for `maxtemp`. In each plot, we saw the five-number summary for each city, side by side. It wasn't a surprise to find that while Boston's winter months are a bit warmer than Chicago, Los Angeles is far warmer than either of them.

### 10.3.2 Solution

```
filename = '../data/chicago_il.csv'
df = pd.read_csv(filename,
                 usecols=[0, 1, 2], #1
                 header=0, #2
                 names=['date_time', 'maxtemp', 'mintemp'], #3
                 parse_dates=['date_time'], #4
                 index_col=['date_time']) #5

df['mintemp'].plot.box() #6

df.loc[df['mintemp'] < df['mintemp'].mean() - (df['mintemp'].std(
df.loc['01-Feb-2019':'28-Feb-2019', 'mintemp'].plot.box() #8
df.loc['01-Feb-2019':'30-Mar-2019', ['mintemp', 'maxtemp']].plot.b

df['city'] = 'Chicago' #10

for city_stem in ['los+angeles,ca', 'boston,ma']: #11
    new_df = pd.read_csv(f'../data/{city_stem}.csv', #12
                        usecols=[0, 1, 2],
                        header=0,
```

```

        names=['date_time', 'mintemp', 'maxtemp'],
        parse_dates=['date_time'],
        index_col=['date_time'])
new_df['city'] = city_stem.split(',')[0].replace('+', ' ').ti
df = pd.concat([df, new_df])#14

```

```

df.groupby('city')[['mintemp', 'maxtemp']].describe()#15
df.plot.box(column=['mintemp', 'maxtemp'], by='city')#16

```

### 10.3.3 Beyond the exercise

- Rather than starting with data from Chicago, start with an empty data frame, and use a for loop to load data from all three cities.
- For each city, calculate the mean and median for mintemp and maxtemp. Are they the same (or even close)? If they're different, in which direction were they pulled?
- Create a line plot showing the minimum temperatures in each city. The x axis should show dates, the y axis should show temperatures, and each line should represent a different city.

## 10.4 Exercise 45: Taxi fare breakdown

We've looked at New York City taxi fares a number of times in this book. This time, we're going to look at this data set visually, plotting the data from a variety of perspectives. It's hard to exaggerate not just how much of an impact a good plot can have when presenting it to others, but also to better understand the data set yourself. You'll see new relationships in the data, and know not only how to answer questions you already asked, but what new questions you should be asking.

I'd like you to do the following:

- Load data from all four NYC taxi files into a single data frame. We'll need a bunch of different columns: 'tpep\_pickup\_datetime', 'passenger\_count', 'trip\_distance', 'fare\_amount', 'extra', 'mta\_tax', 'tip\_amount', 'tolls\_amount', 'improvement\_surcharge', 'total\_amount', and 'congestion\_surcharge'.
- Create a bar plot showing how many rides took place during each month

- and year of our data set. (It's fine if there are "holes" in the bar plot.)
- Create a bar plot showing the total amount paid in taxi rides for every year and month of our data set.
  - Create a bar plot showing fare\_amount, extra, mta\_tax, tip\_amount, and tolls\_amount paid in taxi rides, per month + year—with the various components stacked in a single bar per year/month.
  - Create a bar plot showing fare\_amount, extra, mta\_tax, tip\_amount, and tolls\_amount paid in taxi rides, per number of passengers, stacked in a single bar per number of passengers.
  - Create a histogram showing the frequency of each tipping percentage between (and including) 0% and 50%

### 10.4.1 Discussion

This exercise was all about visualizing our taxi data, and in order to make the data more interesting and varied, I asked you to load all four of the CSV files I've made available—from January 2019, July 2019, January 2020, and then July 2020. I loaded them, as I've done before, most recently in Exercise 42, via a list comprehension:

```
filenames = ['../data/nyc_taxi_2019-01.csv', '../data/nyc_taxi_20
            '../data/nyc_taxi_2020-01.csv', '../data/nyc_taxi_20

all_dfs = [pd.read_csv(one_filename,
                    usecols=['tpep_pickup_datetime', 'passenger_count', 't
                        'fare_amount', 'extra', 'mta_tax', 'tip_amount',
                        'improvement_surcharge', 'total_amount', 'conge
                    parse_dates=['tpep_pickup_datetime'])
            for one_filename in filenames]
```

In this case, I passed usecols the list of columns that I had asked for in the question. I also passed parse\_dates a single value, the column tpep\_pickup\_datetime. (In this exercise, I didn't see a need for us to have the dropoff datetime.) This created a list of data frames, which I was then able to concatenate into a single data frame using pd.concat:

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

With our data frame in place, we can now begin to perform our analysis. I first asked you to create a bar plot showing how many rides there were in

each year and month of our data set. In order to do this, we'll run a groupby, grouping by two columns—first by year, and then by month:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,  
           df['tpep_pickup_datetime'].dt.month])
```

This, of course, gives us a groupby object on which we can perform the query. For this part of the exercise, I asked you to find the total amount paid in each year-month period of our data set. I ran the query as follows:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,  
           df['tpep_pickup_datetime'].dt.month])['total_amount']
```

This produces a numeric result, showing the total amount paid for each year-month combination of our data set. Given that we only loaded four months of data, it might seem strange that we have data from other months and years. Some of that data might not have been loaded when it was first created, leading to a delay. Or the data might be corrupt. Likely, it's a combination of both of these factors; even in a fully automated system, you shouldn't be surprised to have some bad data.

I then asked you to create a bar plot from this data:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,  
           df['tpep_pickup_datetime'].dt.month])['total_amount']
```

The call to `plot.bar` creates the bar plot based on the data frame that we got from the groupby. That data frame's index serves as the plot's x axis, while the values determine the axis, which I allowed to be generated automatically.

Next, I asked you to create a bar plot showing, again for every year-month combination, the number of taxi rides per month. Once again, we start with a groupby query:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,  
           df['tpep_pickup_datetime'].dt.month])
```

Here, we're not interested in totalling the receipts, but rather in just counting the rows. While we could run `count`, the aggregation method that counts rows, on the entire data frame, that'll give us an annoying and repeated count,

once per column. We don't really need that. So I chose to select a single column, `passenger_count`—although we really could have chosen any of them:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,
            df['tpep_pickup_datetime'].dt.month])['passenger_count']
```

Finally, I took this data frame and turned it into a bar plot. As before, I called `plot.bar` with a keyword argument of `figsize=(10, 10)`, ensuring the image would be a 10-inch (25 cm) square:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,
            df['tpep_pickup_datetime'].dt.month])['passenger_count']
```

While the x axis in both this plot and the previous one are the same, and while we see bars in the same places, the values are obviously different. Moreover, we can see that while July 2019 was the month with the greatest amount of revenue, it had the third-greatest number of rides. We can also see (as we've discussed in previous exercises) that in July, 2020—at the height of the pandemic—there were significantly fewer rides, and also significantly less taxi revenue.

Next: We've generally talked about the `total_amount` column when it comes to taxi revenue. But `total_amount` is the final dollar figure that a taxi passenger has to pay at the end of the ride. While passengers don't often think about this, that fare can be broken down into a number of different pieces. In this question, I asked you to create plot not only the amount of revenue that we got each month in the data set, but to break that bar down into segments, thus allowing us to see how much of each month's revenue came from each source.

Once again, I used `groupby` on the year and month columns:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,
            df['tpep_pickup_datetime'].dt.month])
```

Because I wanted to produce the plot with input from five columns—`fare_amount`, `extra`, `mta_tax`, `tip_amount`, and `tolls_amount`—I then named them in a list of column names after the `groupby`:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,
            df['tpep_pickup_datetime'].dt.month])[['fare_amount',
```

I then ran the sum method, which gives me a separate sum for each of these five columns, in each of the months for which we have data:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,
            df['tpep_pickup_datetime'].dt.month])[['fare_amount',
```

Finally, I asked pandas to create a bar plot:

```
df.groupby([df['tpep_pickup_datetime'].dt.year,
            df['tpep_pickup_datetime'].dt.month])[['fare_amount',
            'extra', 'mta_tax',
            'tip_amount', 'tolls_amount']].sum().plot.bar(stacked=True)
fig
```

However, there's a difference between my previous calls to `plot.bar` and this one: Normally, I would get a separate plot for each column, for each month. But because I specified `stacked=True`, I got all of the bars for a given month stacked on top of one another. Moreover, each portion of the bar was in a different color, and pandas provided a legend, as well. In this was, I was able to see, visually, not just how much revenue taxis brought in each month, but also how much of that revenue came from the fare itself, as opposed to taxes, tips, and tolls. We can see that while the fare is by far the greatest proportion of the total taxi revenue, tips constitute a fairly large proportion, as well, followed by extra charges, taxes, and tolls.

Next, I asked for a similar stacked bar plot, with the same five columns as components in each bar. However, rather than grouping by the year and month, I asked you to group by the `passenger_count` column. We can do that with a similar query to the previous one, grouping on `passenger_count` rather than by year-month combination:

```
df.groupby(df['passenger_count'])[['fare_amount', 'extra',
                                   'mta_tax', 'tip_amount', 'tolls_
```

Finally, I asked you to create a histogram showing the frequency of each tipping percentage between (and including) 0 and 50. In order to do this, I needed to find the tipping percentage for each ride, and then keep only those

between 0 and 50. I decided that the easiest thing would be to create a new column, `tip_percentage`, from dividing `tip_amount` by `fare_amount`. But the real world includes all sorts of surprises, not only including `NaN` values, but also records in which the `fare_amount` was equal to zero—thus giving us an infinite (known as `np.inf`) value.

To avoid this, I first got rid of any ride in which the fare was less than or equal to 0:

```
df = df[df['fare_amount'] > 0]
```

Then I created a new column, `tip_percentage`, knowing that I won't get any `np.inf` values:

```
df['tip_percentage'] = df['tip_amount'] / df['fare_amount']
```

Finally, I plotted all of the values than than or equal to 50%:

```
df.loc[df['tip_percentage'] <= .50, 'tip_percentage'].plot.hist()
```

The resulting histogram has a huge bar—the largest—for 0% tips, indicating that a plurality of New York taxi riders don't tip at all. But other than that bar, we see a fairly normal distribution, centered around 20 or 25%.

## 10.4.2 Solution

```
filenames = ['./data/nyc_taxi_2019-01.csv', './data/nyc_taxi_20  
              './data/nyc_taxi_2020-01.csv', './data/nyc_taxi_20
```

```
all_dfs = [pd.read_csv(one_filename,  
                      usecols=['tpep_pickup_datetime', 'passenger_count', 't  
                                'fare_amount', 'extra', 'mta_tax', 'tip_amount',  
                                'improvement_surcharge', 'total_amount', 'conge  
                      parse_dates=['tpep_pickup_datetime'])  
            for one_filename in filenames]
```

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

```
df.groupby([df['tpep_pickup_datetime'].dt.year,  
           df['tpep_pickup_datetime'].dt.month])['passenger_coun
```

```

df.groupby([df['tpep_pickup_datetime'].dt.year,
           df['tpep_pickup_datetime'].dt.month])['total_amount']

df.groupby([df['tpep_pickup_datetime'].dt.year,
           df['tpep_pickup_datetime'].dt.month])[['fare_amount',
           'mta_tax', 'tip

df.groupby(['passenger_count'])[['fare_amount', 'extra', 'mta_tax',

df = df[df['fare_amount'] > 0]
df['tip_percentage'] = df['tip_amount'] / df['fare_amount']
df.loc[df['tip_percentage'] <= .50, 'tip_percentage'].plot.hist()

```

### 10.4.3 Beyond the exercise

- Create a bar plot, showing the average distance traveled per day of the week in July 2020. The x axis should show the name of each day.
- Create a scatter plot with the taxi data from July 2020, comparing trip\_distance with total\_amount. Ignore all rides in which either value was less than or equal to 0, or greater than 500.
- Create a scatter plot with the taxi data from July 2020, comparing trip\_distance with passenger\_count. Ignore all rides in which trip\_distance was less than or equal to 0, or greater than 500.

**Correlation isn't causation. But what is it?**

No matter where you are in your data-analysis career, you're bound to hear someone say "correlation isn't causation." What does that mean? Moreover, what **is** correlation?

Loosely speaking, two measurements are correlated when movement in one is generally accompanied by movement in another. If the measurements rise and fall together, then they're considered "positively correlated." If one goes up when the other goes down (and vice versa), then they're said to be negatively correlated.

In addition to be positive or negative, correlation can be weak or strong. There's probably a strong correlation between your annual income and the size of your house. There's probably a weak correlation between your annual

income and your shoe size. (Although to be fair, higher income correlates with better nutrition and better health, so the correlation might be stronger than you'd expect.)

Let's take a simple example: The more electric power you use, the higher your electric bill will be. If you use more electricity, your bill goes up. If you use less electricity, your bill goes down. We can thus say that your electric consumption and your electric bill are positively correlated.

Here's another example: The wealthier you are, the more likely you are to own a private jet. If you're a multi-billionaire, then you probably have a jet, and probably several. (At least, that's what I've learned from watching "Succession.") So we can say that as your income goes up, the number of private jets you own goes up. And as your income goes down, the number of private jets you can afford to keep on hand will probably go down, as well.

It's very tempting, when we see data that is correlated, to say that one causes another. And in some cases, that's certainly true: We can safely say that your higher electric bill was caused by greater consumption.

But just because two data points are correlated doesn't mean that one causes the other. And even if one does, you have to be careful to determine just what causes what. For example, if there is a causal relationship between private-jet ownership and billionaire status, then perhaps I should buy a private jet. That'll raise the likelihood of my becoming a billionaire, right?

There are numerous examples of correlations without causation. For some terrific examples, check out the "Spurious Correlations" web site by Tyler Vigen: <https://tylervigen.com/spurious-correlations>

This difference between correlation and causation was most famously used by the tobacco industry. True, they said, people who smoke cigarettes are more likely to have cancer. But just because there's a correlation there doesn't mean that it's a causal effect. Can we really know whether cigarettes cause cancer? After many studies, and many years, it became clear that the answer is "yes": We can know, and the effect is causal.

Finding a causal relationship is hard, and generally requires doing an

experiment. You divide the population into two parts, giving one half the treatment and the other half no treatment (or a placebo). Then you measure the difference in effects on the two populations.

Fortunately, in the world of data analytics, we're often less interested in causation than in finding correlations. If I find that my online store gets more sales between 12 noon and 1 p.m., I don't really care what's causing it—but I do want to know about it, and take advantage of it.

This raises the question, though: What exactly does it mean for two sets of numbers values to be correlated?

Let's take two sets of numbers, the high and low temperatures for my city of Modi'in over the coming week:

```
df = DataFrame({'high': [19, 21, 24, 17, 14, 16, 16, 19, 16, 16, 15, 16, 18, 18, 18],
               'low': [12, 9, 11, 12, 11, 11, 10, 8, 10, 8, 8, 6, 6, 7]})
```

What would correlation mean?

- If the columns are positively correlated, then days with the highest high temperatures would also have the highest low temperatures. And the days with the lowest high temperatures would have the lowest low temperatures.
- If the columns are negatively correlated, then days with the highest high temperatures would have the lowest low temperatures. And the days with the lowest high temperatures would have the highest low temperatures.

If the two are strongly correlated, then a large change in one will be accompanied by a large change in the other. If they're weakly correlated, then a large change in one will be accompanied by a small change in the other.

The most common measurement for correlation, and what we'll use in this book, is called "Pearson's correlation coefficient," and is often abbreviated as "r". It's a number between -1 (indicating the strongest possible negative correlation) and 1 (indicating the strongest possible positive correlation), with 0 indicating no correlation. A correlation is always calculated between two data sets, which in the case of pandas, means two different columns.

We can find the correlation for the expected high and low temperatures with the `corr` method:

```
df.corr()
```

The result is a data frame in which each of our original column names appears both as a column and a row. Along the diagonal, where columns meet themselves, there will always be a value of 1.0, indicating (not very usefully) that a column has a perfect positive correlation with itself. More interesting is the intersection between different column names, showing the correlation between each of those pairs of columns. In this case, our data frame only has two columns, so the result will be a bit underwhelming:

```
          high      low
high  1.000000  0.105603
low   0.105603  1.000000
```

We see that there is a correlation of 0.105603 between our high and low temperatures. Meaning that there's a positive correlation between the two, but a very weak one. With more data over a longer period of time, we would probably find a higher correlation. In fact, we can do that by loading the weather data for New York City, with 728 weather measurements:

```
filename = '../data/new+york,ny.csv'

df = pd.read_csv(filename, usecols=[1, 2],
                 header=0,
                 names=['high', 'low'])
```

If I run `df.corr()` on this data frame, we see a different type of result:

```
          high      low
high  1.000000  0.874205
low   0.874205  1.000000
```

Here, we see a very strong positive correlation. This raises the question: How can it be that in one data set, the correlation is very strong, whereas in another one, it's very weak?

There are numerous possible answers: Perhaps Modi'in's temperatures are harder to predict. Perhaps the data that I input was from a particularly

turbulent time, with a high degree of variance. But I think that the real reason is that the sample from Modi'in was extremely small, with only 13 data points. It was hard to establish any correlations based on such a small sample.

Why are we interested in correlations? First and foremost, because it can inform our understanding, and thus our behavior. If we know that our store gets a huge number of requests at lunchtime each day, then perhaps we'll provision additional servers during that time. Or perhaps we'll offer discounts outside of that window, so as to encourage sales during otherwise dead times.

We can also use correlations to hint at underlying similarities and relationships in our data. If two things are correlated, then perhaps there's some behavior that explains the connection between the two. If that behavior or relationship isn't obvious, then it can point to a topic worth investigating or understanding better.

While correlations are normally measured mathematically, it's often possible to see correlations via a scatter plot. In such a plot, we choose one column as the "x" axis and a second column as the "y" axis. We then plot each of the points. We cannot expect to see a perfect diagonal line, but such a line starting at (0,0) and moving up and to the right points to a strong positive correlation in the columns. One that starts high up on the y axis and moves down to the right indicates a strong negative correlation. Using a scatter plot is a great way to better understand the data. In pandas, we can create such a plot based on a data frame with the `plot.scatter` method:

```
df.plot.scatter(x='high', y='low')
```

In this case, we'll indeed see a strong positive correlation, matching the numeric calculations that we performed earlier.

## 10.5 Exercise 46: Cars, Oil, and ice cream

In this exercise, we're going to try to answer a question that has probably occurred to you on many occasions: When the price of oil goes up, do people drive more or less in their cars? And while we're at it, we'll also try to answer another question, namely whether the price of ice cream is correlated

with the price of oil.

This exercise will not only try to identify these correlations, but will also use many of the techniques we've discussed in the book so far, including parsing dates, selecting appropriate rows and columns, removing bad data, and joining data frames together. Specifically, I want you to:

- Load the oil data (from Exercise 41) into a data frame. Set the names of the columns to be "date" and "oil", with the "date" column parsed as a date and set to be the index.
- Load historical ice-cream prices in the United States (for a half gallon, aka 1.9 liters) into a separate data frame, from the file `ice-cream.csv`. Set the column names to be "date" and "icecream". The "date" column should be parsed as a date and set to be the index.
- Set the `icecream` column to be a floating-point value, removing any rows that stop you from accomplishing that.
- Load historical US "miles traveled per month" data into a separate data frame. Name the columns `date` and `miles`, parsing `date` as a date, and setting it to be the index.
- Create a single data frame from these three data frames. The index should be the date, and the new data frame should have three columns—`oil`, `icecream`, and `miles`. Only dates that are common to all three should be included.

### 10.5.1 Discussion

In this exercise, we took three distinct data sets, merged them together to make a new data frame, and then found correlations among the various columns. And the results were... not what I was expecting, to say the least.

Before we can calculate the correlations, we first have to load the data. I always like to create separate data frames, and then join them together. This not only lets me do things step by step, but also ensures that I can debug, improve, and rerun my steps more easily.

The first data frame I asked you to create was similar to one we already looked at in Exercise 41. In order to make our `join` operation run more

smoothly later, I asked you to standardize some parts of the naming. For example, I wanted to parse the date column as a datetime, and also to set it to be the index. I also renamed the columns, calling them `date` and `oil`.

Most of the time, and especially when a CSV file has headers indicating the column names, I like to use those names in my call to `read_csv`. That makes the function call easier to read and debug. But when you want to rename the columns with the `names` parameter, you need to describe them numerically. Moreover, in order to avoid having the header row read as data, we need to indicate which row contains the header (0, in our case), effectively causing it to be ignored.

In the end, I loaded the oil data as follows:

```
oil_filename = '../data/wti-daily.csv'
oil_df = pd.read_csv(oil_filename,
                    parse_dates=[0],
                    header=0,
                    index_col=0,
                    names=['date', 'oil'])
```

A brief check (with `oil_df.head()` and `oil_df.dtypes` showed that we had successfully created the data frame, with the correct dtype. With the oil data in hand, it was time to create the next data frame, based on the monthly ice-cream price data that I got from the US government.

This file is in a very similar format to the oil data, in a CSV file containing two columns. The first column is a date—the final date of each month, when the ice-cream pricing data is recorded. We can thus load it with our usual combination of keyword arguments:

```
ice_cream_filename = '../data/ice-cream.csv'
ice_cream_df = pd.read_csv(ice_cream_filename,
                          parse_dates=[0],
                          index_col=0,
                          header=0,
                          names=['date', 'icecream'])
```

However, running `ice_cream_df.dtypes` shows that the `icecream` column didn't load as a floating-point value. Rather, it loaded as `object`. That's usually a good sign that one or more values tripped up the system that pandas

uses to identify and assign dtypes on our CSV files. I decided to see where the problem was by trying to turn the column into an `np.float64` value:

```
ice_cream_df['icecream'].astype(np.float64)
```

Sure enough, it failed, telling me that it choked on a line containing nothing more than `.`, instead of a price. I decided to gamble that any problematic lines would be like this, meaning that if I were only to keep lines containing digits, then I'd be able to convert the column into float values. I decided to use a regular expression for this, looking for `'\d'`, meaning "any digit," removing those rows that lack even a single digit:

```
ice_cream_df = ice_cream_df[ice_cream_df['icecream'].str.contains
```

Notice that I used a raw string (i.e., a string with an `r` before the opening quote). Raw strings are Python's way of automatically doubling backslashes, thus ensuring that Python doesn't pre-digest our backslashes before they get to the regular expression engine. I used `str.contains`, a method that we encountered earlier in this book, to find all rows that do indeed contain at least one digit; that returned a boolean series, which I used as a mask index and then assigned back to `ice_cream_df`.

With the bad row—in the end, only one lacked any digits—excluded, I was able to then convert the column to a float value:

```
ice_cream_df['icecream'] = ice_cream_df['icecream'].astype(np.flo
```

Next, I asked you to create a data frame containing the US government's report on total miles traveled during each calendar month. My naive assumption was that when oil prices are high, people will drive less, but that when they're low, they'll drive more. I created the new data frame using similar arguments as what we've already seen:

```
miles_filename = '../data/miles-traveled.csv'  
miles_df = pd.read_csv(miles_filename, parse_dates=[0],  
                      index_col=0,  
                      header=0,  
                      names=['date', 'miles'])
```

With these three data frames in place, it was time to join them together.

We've already seen how we can join two data frames together using the `join` method. But here, I asked you to join three data frames together. How can we do that?

The answer, once you see it, is straightforward: We join two data frames together, getting a new one. We then join this new data frame together with the third, to get a final, new one. So long as all of the data frames share an index, we should be fine:

```
df = oil_df.join(ice_cream_df).join(miles_df)
```

But if we do things this way, we'll discover that there's a bit of a hitch: Oil price data was recorded once per day, as opposed to the ice-cream and travel data, which were recorded once per month. Joining our data frames in this way will result in a new row for each index value in `oil_df`, and NaN values in all but one row per month.

There are a few ways to solve this problem. One is to perform the join as I did above, and then use `dropna` to remove all of the NaN-containing rows:

```
df = oil_df.join(ice_cream_df).join(miles_df).dropna()
```

A second method would be to perform the join on `ice_cream_df`, thus constraining the index values:

```
df = ice_cream_df.join(oil_df).join(miles_df)
```

But my preferred solution is to use an "inner" join, meaning that our index will only contain values that existed in all three data frames. I can do this by passing the keyword argument `how='inner'` to each call to `join`:

```
df = oil_df.join(ice_cream_df, how='inner').join(miles_df, how='i
```

The result is a data frame whose index contains 275 distinct values, from April 1986 through December 2021. With all of these values in place, we can (finally) start to look for correlations in our data. First, we can run `corr` on our data frame to find the correlations across all columns:

```
df.corr()
```

The resulting data frame has three columns (`oil`, `icecream`, and `miles`) and identical rows. The intersection of the column names give us the correlation, ranging from -1 to 1. We can see that oil prices and the number of miles traveled per month are positively correlated, with a value of 0.64. The correlation between gas prices and ice-cream prices is not only positive, but much larger, at 0.77.

But the biggest correlation of all is between the price of ice cream and the number of miles driven per month, with a value of 0.818. That's quite a large correlation factor, indicating that whenever ice-cream prices decline, people drive less, and vice versa.

Can we realistically say that there is a causal relationship here? I highly doubt it; I don't think that you are likely to drive more because you ate more ice cream, or that you eat more ice cream because you drove more. A more likely explanation, at least to me, is that people both drive more and eat more ice cream in the summer months, and that ice-cream prices rise when there's more demand. I haven't done any serious analysis to see if this is the case, but it seems more likely than either random chance or a causal effect.

Next, I asked you to produce two scatter plots, first between `oil` and `icecream`:

```
df.plot.scatter(x='oil', y='icecream')
```

The second scatter plot I asked you to make was between `oil` and `miles`:

```
df.plot.scatter(x='oil', y='miles')
```

While you might be able to identify, from these scatter plots, whether there is a positive correlation here, I think that the numbers give us a much clearer indication of the strength of that correlation.

Finally, I asked you to create a single "scatter matrix" plot, showing all of the numeric columns plotting against one another:

```
from pandas.plotting import scatter_matrix
scatter_matrix(df)
```

The scatter matrix is a great way to get a quick look at all of the correlations in your data set. The diagonal, which always contains 1.00 values in the call to `df.corr()`, is a histogram in the scatter matrix, indicating the distribution of values in each column.

## 10.5.2 Solution

```
oil_filename = '../data/wti-daily.csv'
oil_df = pd.read_csv(oil_filename,
                    parse_dates=[0],
                    header=0, #1
                    index_col=0, #2
                    names=['date', 'oil']) #3

ice_cream_filename = '../data/ice-cream.csv'
ice_cream_df = pd.read_csv(ice_cream_filename,
                          parse_dates=[0],
                          index_col=0,
                          header=0,
                          names=['date', 'icecream'])

ice_cream_df = ice_cream_df[ice_cream_df['icecream'].str.contains
ice_cream_df['icecream'] = ice_cream_df['icecream'].astype(np.flo

miles_filename = '../data/miles-traveled.csv'
miles_df = pd.read_csv(miles_filename, parse_dates=[0],
                      index_col=0,
                      header=0,
                      names=['date', 'miles'])

df = oil_df.join(ice_cream_df, how='inner').join(miles_df, how='i

df.corr() #7

df.plot.scatter(x='oil', y='icecream') #8
df.plot.scatter(x='oil', y='miles')

from pandas.plotting import scatter_matrix
scatter_matrix(df) #9
```

## 10.5.3 Beyond the exercise

- Is the month correlated with them at all?
- Create a scatter plot of icecream vs. miles, adding color using the

month the "Spectral" colormap.

- Instead of using an inner join, we could have removed all of the rows from `oil_df` that weren't on the final day of the month. How could we do that?

## Seaborn

Matplotlib is, without a doubt, the leading plotting system for Python. Many people have found it hard to learn and use, however, which has led to the creation of several alternatives to Matplotlib. One of the best-known alternatives, Seaborn ([seaborn.pydata.org/](http://seaborn.pydata.org/)), was written by data scientist Michael Waskom, and acts as an API on top of Matplotlib.

So far, this book has focused on the pandas plotting API, which (like Seaborn) uses Matplotlib to produce its plots. But the pandas API tries to simplify things, papering over much of the configuration that needs to happen in order to create a plot, but otherwise keeping Matplotlib's approach and API intact. By contrast, Seaborn rethinks how plotting should be done, replacing the original Matplotlib and pandas calls with a distinct set of functions and parameters.

Just as we typically import `numpy` as `np` and import `pandas` as `pd`, we also import Seaborn with an alias:

```
import seaborn as sns
```

Whereas pandas visualization is all done via the `plot` attribute, followed by the type of plot we want to create, Seaborn is organized more conceptually, around the different types of insights we might be trying to draw from our plots. We can choose from four different functions defined within `sns`:

- To visualize relationships among numeric columns, use `sns.relplot`.
- To visualize relationships that include categorical columns, use `sns.catplot`.
- To understand the distribution of data, use `sns.displot`.
- To visualize regression models, use `sns.regplot`.

In order to explore this more fully, let's load up our temperature and

precipitation data from our weather CSV files:

```
import glob

all_dfs = []

# all_filenames = glob.glob('../data/*/*.csv')
all_filenames = ['../data/chicago,il.csv']

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

We've already seen how line and scatter plots can give us insights into the relationship between two numeric columns. Seaborn puts both of them in its `relplot` function. Let's first look at how we can create a scatter plot for min vs. max temperatures in Chicago:

```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df.loc[df['city'] == 'Chicago'])
```

Our call to `sns.relplot` includes three mandatory keyword arguments:

- `x` indicates which column from our data frame will be used for the x axis
- `y` indicates which column from our data frame will be used for the y axis
- `data` is a data frame containing both of those columns

In this case, I decided to provide only a subset of the data from `df`, so that we would only see Chicago weather. But what if I want to see all of the data, from all of the cities?

```
sns.relplot(x='max_temp',
            y='min_temp',
```

```
data=df)
```

The good news is that this is much easier to write. But the bad news is that it's not nearly as useful. Now we've mixed together all of the weather reports from all of the cities! Fortunately, Seaborn provides us with a number of different ways to make the data more useful and interesting.

For example, we can ask Seaborn to use a different color for each city by passing a column name to the hue keyword argument:

```
sns.relplot(x='max_temp',  
            y='min_temp',  
            data=df,  
            hue='city')
```

We can have each city's dots use a different marker, as well, by giving the same city argument to the style parameter:

```
sns.relplot(x='max_temp',  
            y='min_temp',  
            data=df,  
            hue='city', style='city')
```

We don't have to use the same categorical data for hue and style. For example, we can set the hue per state:

```
sns.relplot(x='max_temp',  
            y='min_temp',  
            data=df,  
            hue='state', style='city')
```

However, it's a bit messy to see all of these plots on the same axes. We can ask Seaborn to do the visual equivalent of a groupby, with one plot per value of city. There are two different ways to do this, actually, by setting row (i.e., each row is a different value for the named column) or col (i.e., each column is a different value for the named column). For example:

```
sns.relplot(x='max_temp',  
            y='min_temp',  
            data=df,  
            hue='state',  
            row='city')
```

While scatter plots are extremely useful, we can also see the relationship between two numeric columns with line plots. The most obvious difference between the two kinds of plots is that Seaborn draws a line between the dots. For example:

```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df,
            hue='state', kind='line')
```

The above call is fine, **except** that it won't work. In my case, I got both a warning from pandas and an error message from Seaborn. Both of them told me that they cannot handle my data frame as it stands, because its index contains non-unique values.

I can fix this easily with `reset_index`:

```
df.reset_index(drop=True)
```

Note that I passed `drop=True` to avoid having the old index added as a column to the data frame. I'm happy to throw out the old index and replace it with a new one, so I pass `drop=True`.

With a new index in place, we can again ask Seaborn to create our line plot:

```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df,
            hue='state', kind='line')
```

The good news is that we see all of the values, and (thanks to our value for `hue`, we have a different-colored line for each state. The bad news is that two of the cities in our data set are from the same state. And besides, it's a bit hard to read this plot, with all of the data squashed together.

We can once again ask Seaborn to put each city in a separate row:

```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df,
            hue='state',
```

```
kind='line',  
row='city')
```

Seaborn supports a wide variety of other plots, as well. For example, what if we want to see all of the values of `max_temp` for a given city? You can think of this as a set of one-dimensional scatter plot, and we can see it as follows:

```
sns.catplot(x='city', y='max_temp', data=df)
```

Notice how the x axis is for the categories, while the y axis describes which value we'll be seeing. This plots each of the values in the data set. If we instead want to summarize our data, we can ask to have a box plot, instead:

```
sns.catplot(x='city', y='max_temp', data=df, kind='box')
```

This shows a box plot for each of the cities' values of `max_temp`, all side-by-side on the same y axis.

Finally, Seaborn also offers us the chance to create histograms. Because histograms allow us to understand the distribution of our data, we use the `sns.displot` function. For example, we can get a histogram of all maximum temperatures:

```
sns.displot(x='max_temp', data=df)
```

This, of course, shows the distribution of all values of `max_temp`. We can, once again, give each city its own colored bars by setting `hue`:

```
sns.displot(x='max_temp', data=df, hue='city')
```

And we can see them in a single column, with only one city per row, by saying:

```
sns.displot(x='max_temp', data=df, hue='city', row='city')
```

These are just some of Seaborn's many capabilities. If you're interested in seeing everything that Seaborn can do, I strongly recommend you check out the documentation at [seaborn.pydata.org/](https://seaborn.pydata.org/). I've grown to really like the Seaborn approach to visualization—not only does it produce very nice-looking plots, but I find the API easier to understand and work with than

many others.

## 10.6 Exercise 47: Seaborn taxi plots

In this exercise, we're going to revisit our New York City taxi data from 2020, creating some visualizations with Seaborn, rather than with the built-in pandas plotting system. Specifically, I want you to:

- Load data from NYC taxis in 2020, only loading columns `tpep_pickup_datetime`, `passenger_count`, `trip_distance`, and `total_amount`.
- Add columns "month" and "year", from `tpep_pickup_datetime`. Keep only those data points in which the year is 2020 and the month is either January or July.
- Set a new numeric range index, numbered starting at 0
- Assign `df` to a random sample of 1% of the elements in the original `df`
- Using Seaborn, create a scatter plot in which the x axis shows `trip_distance` and the y axis shows `total_amount`, with the plot colors set by `passenger_count`. Use the 1% sample of the data.
- Why do we have colors for `passenger_count` of 1.5, 4.5, and 7.5?
- Create a line plot showing the distance traveled on each day of January and July. The x axis should be the day of the month, and the y axis will be the average trip distance. There should be two lines, one for each month.
- Using Seaborn, show the number of trips taken on each day (1-31) of both months (January and July). The x axis should refer to the day of the month, and the y axis should show the number of trips taken.
- Using Seaborn, create a box plot of `total_amount`, with one plot for each month.

### 10.6.1 Discussion

In this exercise, I asked you to create plots of 2020 New York City taxi data, from January and July, and then to use Seaborn to plot that data. We started off by creating a data frame based on the 2020 taxi files, loading four of our favorite columns:

```

filenames = ['../data/nyc_taxi_2020-01.csv', '../data/nyc_taxi_20
all_dfs = [pd.read_csv(one_filename,
                      usecols=['tpep_pickup_datetime', 'passenger_count', 't
                      'total_amount'],
                      parse_dates=['tpep_pickup_datetime'])
           for one_filename in filenames]

df = pd.concat(all_dfs)

```

Notice that I once again used `parse_dates` to turn the `tpep_pickup_datetime` column into a `datetime` column, leaving the three others to be detected as floating-point values. This code again creates a list of data frames using a list comprehension. The list is then passed to `pd.concat`, which returns a new data frame that combines all of the input data frames.

I then asked you to create three new columns from various parts of each row's date:

```

df['year'] = df['tpep_pickup_datetime'].dt.year
df['month'] = df['tpep_pickup_datetime'].dt.month
df['day'] = df['tpep_pickup_datetime'].dt.day

```

I then asked you to ensure that all of the data we look at is from January or July of 2020. As we've seen, the taxi data is a bit "dirty," including a number of rows from other years and months. To avoid having our plots come out odd looking, I thought it would be wise to remove rows that aren't from January and July of 2020. We can do that by using a combination of mask indexes:

```

df = df.loc[(df['month'].isin([1, 7])) & (df['year'] == 2020)]

```

Next, I asked you to ensure that the new data frame's index doesn't contain duplicate values—something that is almost certainly the case at this point, given that we created `df` from two previous data frames. You can actually check to see if a data frame's index contains repeated values with the code

```

df.index.is_unique

```

If this returns `True`, then the values are already unique. If not, then some Seaborn plots will give you errors. We could renumber the index on our own,

but why work so hard, when pandas includes this functionality? We can just say:

```
df = df.reset_index(drop=True)
```

Yes, this is the same `reset_index` that we've used before to get rid of a "special" index that we've created, such as from a data column. By passing `drop=True`, we tell `reset_index` not to make the just-ousted index column a regular column in the data frame, but rather to drop it entirely.

We could then start to plot our data. But the data set is rather large, with many millions of data points. To speed up our plotting, albeit at the cost of some accuracy, I asked you to keep a random 1 percent of the original `df`'s values, and then to assign it to `df`:

```
df = df.sample(frac=0.01)
```

We're now finally ready to plot our data with Seaborn. First, I asked you to create a scatter plot comparing `trip_distance` (x axis) with `total_amount` (y axis) on the `df` containing 1 percent of our original data:

```
sns.relplot(x='trip_distance', y='total_amount', data=df,
            hue='passenger_count')
```

The `relplot` function is there to show us relationships among numeric columns, and the default way to do that is with a scatter plot. Here, we tell `relplot`:

- The x axis should use values from the `trip_distance` column
- The y axis should use values from the `total_amount` column
- We should use `df` as our data frame
- We'll use `passenger_count` as the basis for coloring the lines and dots

Sure enough, this works, giving us a nice scatter plot.

Next, I asked you to show a line plot in which the x axis indicates days of the month (1-31), and the y axis shows the value of `trip_distance` on that date. Once again, we'll use `relplot` to get that plot:

- The x axis will be from the day column
- The y axis will be from the trip\_distance column
- We have to indicate that kind='line', to get the line plot
- We say that data comes from the df data frame
- We color each of the lines by month

```
sns.relplot(x='day', y='trip_distance', kind='line',
            data=df, hue='month')
```

By asking Seaborn to have separate colors for each value of month, we were able to get two different lines plotted on the same chart.

Notice, though, that there are some gray lines around each of our plots. Those indicate the "confidence interval" for each calculation. Confidence intervals are a statistical tool to indicate how likely a value is to fall within a certain range. We can disable the confidence intervals by passing ci='None' on a relplot:

```
sns.relplot(x='day', y='trip_distance', kind='line',
            data=df, hue='month', ci=None)
```

Next, I asked you to show the number of trips taken on each day of these months. This will require another line plot:

- The x axis will be the day of each month
- The y axis will reflect how many trips were taken on that day
- We have to indicate that kind='line' to get a line plot

But wait a second: How will we get the number of trips taken each day? In order to do that, we'll need to use the count aggregation method. And indeed, here I suggest not getting data back from df, but rather from the result of a groupby on df. If we count by both month and day, and count in the year column, we'll have access to month and day, but also to the number of rides there were per day. (Using year is a bit weird, since we aren't counting the year at all—but we need to pick some column.) After performing this groupby, I'll then reset the index, making month and day back into regular columns, from which they can be retrieved:

```
sns.relplot(x='day', y='year', hue='month', kind='line',
```

```
data=df.groupby(['month', 'day'])[['year']].count().r
```

This is a complex query, and it's then used in a complex plot. So let's walk through this once again, a step at a time:

- We want to know how many rides there on each day of each month. That requires `groupby(['month', 'day'])`.
- We then run the `count` aggregation method on the `groupby` object
- The result gives us a count for each remaining column in the data frame. We only need one, and I choose `year`.
- I then run `reset_index` to take `month` and `day`, which are part of the index of the aggregation data frame, and put them back into the main data frame.
- I then pass the result from `reset_index` as the argument to `data`, in my call to `replot`
- I tell `replot` that the x axis should be based on `day` and the y axis should be based on `year`, the count of rides
- I tell `replot` to distinguish between months by color
- I ask for a line plot
- Finally, I ask for `ci='None'` to avoid any confidence intervals from being shown

We then see, rather dramatically, that there were fewer rides per day in July (in the middle of the pandemic) than there were in January (before it started).

Finally, I ask to see a box plot of the `total_amount` column, separated by month.

Box plots are, in the world of Seaborn, categorical plots, because they allow us to compare the distribution of values across multiple categories. We thus need to use the `catplot` function:

- The x axis will be the categories we're comparing, thus `month`
- The y axis will be the values we want to see graphically, thus `total_amount`
- We're looking at data from `df`
- We want to see a box plot, and thus specify `kind='box'`

The code is thus:

```
sns.catplot(x='month', y='total_amount', data=df, kind='box')
```

We see that the mean values for `total_amount` weren't that different in January and July of 2020. And sure enough, we can see this numerically:

```
df.groupby('month')['total_amount'].mean()
```

## 10.6.2 Solution

```
filenames = ['./data/nyc_taxi_2020-01.csv', './data/nyc_taxi_2020-07.csv']
all_dfs = [pd.read_csv(one_filename,
                      usecols=['tpep_pickup_datetime', 'passenger_count', 'total_amount'],
                      parse_dates=['tpep_pickup_datetime'])
           for one_filename in filenames]

df = pd.concat(all_dfs)

df['year'] = df['tpep_pickup_datetime'].dt.year
df['month'] = df['tpep_pickup_datetime'].dt.month
df['day'] = df['tpep_pickup_datetime'].dt.day

df = df.loc[(df['month'].isin([1, 7])) & (df['year'] == 2020)]

df = df.reset_index(drop=True)

df = df.sample(frac=0.01)

sns.relplot(x='trip_distance', y='total_amount', data=df,
            hue='passenger_count')

sns.relplot(x='day', y='trip_distance', kind='line',
            data=df, hue='month', ci=None)

sns.relplot(x='day', y='year', hue='month', kind='line',
            data=df.groupby(['month', 'day'])[['year']].count().r

sns.catplot(x='month', y='total_amount', data=df, kind='box')
```

## 10.6.3 Beyond the exercise

- Load NYC taxi data from both 2019 and 2020, January and July. Remove data from outside of those years and months. Now display the number of trips on each day of the month in four separate graphs—the top row in 2019, and the bottom row in 2020, the left column for January and the right column for July.
- Create a histogram, showing how many rides took place per day of each month (January and July). Each month should appear in a different color, and they should appear side-by-side, with January on the left and July on the right.
- Create a bar plot, showing how many rides took place in each hour (0-24) in each month (January and July). Each month should appear in a different color, and they should appear side-by-side, with January on the left and July on the right.

## 10.7 Summary

Visualization is a key part of data science. We often think of it as a way to help non-experts to better understand our data, but it's also a powerful way to better understand our own data, getting insights from a new perspective. In this chapter, we not only saw a number of the ways that pandas can itself perform visualizations, using a simplified API to Matplotlib. We also saw, in the final exercise, how the Seaborn package can create attractive plots using our data frames, using its own, separate API on top of Matplotlib.

# 11 Performance

## 11.1 Useful references

Table 11.1. What you need to know

Concept	What is it?	Example
<code>df.info</code>	Get information about a data frame, including its memory usage	<code>df.info()</code>
<code>df.memory_usage</code>	Get information about a data frame's memory usage	<code>df.memory_usage(deep=True)</code>
Categorical data	pandas documentation for categorical data	<code>df['a'].astype('categorical')</code>
<code>df.to_feather</code>	Write a data frame to feather format	<code>df.to_feather('mydata.feather')</code>

<code>pd.read_feather</code>	Create a data frame, based on a feather-formatted file on disk	<code>df = pd.read_feather('mydata.feather')</code>
<code>pd.read_csv</code>	returns a new data frame based on CSV input	<code>df = df.read_csv('myfile.csv')</code>
<code>pd.read_json</code>	returns a new data frame based on JSON input	<code>df = df.read_json('myfile.json')</code>
<code>time.perf_counter</code>	Get the number of seconds, useful for timing programs	<code>time.perf_counter()</code>
<code>df.query</code>	Write an SQL-like query	<code>df.query('v &gt; 300')</code>
<code>df.eval</code>	Perform actions and queries on a data frame	<code>df.eval('v + 300')</code>

<code>pd.eval</code>	Perform a variety of pandas actions in an evaluated string	<code>pd.eval('df.v &gt; 300')</code>
<code>timeit</code>	Python module for benchmarking code speed, and a Jupyter "magic command" for invoking it	<code>%timeit 3+2</code>
<code>isin</code>	Method to check if a value is in a Python sequence.	<code>df['a'].isin([10, 20, 30])</code>

### Saving memory with categories

Let's say that I want to work with data from our Olympics CSV file:

```
filename = '../data/olympic_athlete_events.csv'
df = pd.read_csv(filename)
```

How much memory does this data set consume? That's an important question when working with pandas, because all of our data needs to fit into memory. We can find out by running the `memory_usage` method on our data frame:

```
df.memory_usage()
```

This returns a series telling us how many bytes are consumed by each column. (The column names from `df` constitute the index of the returned series.) We can get the total memory usage by summing the values:

```
df.memory_usage().sum()
```

On my computer, this comes up as 32,534,048 bytes, or just over 31 MB of RAM.

But you know what? This number is completely wrong. That's because pandas, by default, ignores the size of any Python objects contained in our data frame. Given that these objects are generally strings, and that they can be of any length, the difference between the actual memory usage and what is reported here can be quite big.

We can tell pandas to include all of the objects in its size calculation by passing the `deep=True` keyword argument:

```
df.memory_usage(deep=True).sum()
```

On my computer, the same data frame gives me a result of 186,408,012 bytes, or about 182 MB of RAM—five times the originally calculated amount.

But wait: This is a lot of memory, and the data set is relatively small. A much larger data set will obviously consume much more memory, potentially more than I can fit into my computer. How can I cut down the size of the data set, thus allowing me to potentially work with more data? We've already talked about several of them in past chapters:

- Limit which columns are imported, by passing a value to `usecols`
- Explicitly specify the `dtype` for each column, allowing you to choose types with fewer bits while simultaneously speeding up the loading of data

But as we saw in our calculations, the majority of the memory is being used by strings. This means that we need to somehow reduce the size or number of the text strings in our data frame.

One way to do this is with a special pandas data type known as a "category." In the case of a category, each distinct string value is stored a single time, and then referred to multiple times. However, this replacement is completely transparent to us, as users of the data frame: We can continue to pretend that the column contains strings, including use of the `str` accessor to apply string methods to every element of the column.

We've often used `astype` to create a new series based on an existing one. We can do the same thing to create a new categorical column based on one containing text strings:

```
df['Games'].astype('category')
```

However, this doesn't do anything useful, because it doesn't store our new series anywhere. It's often easiest to just assign the newly created, categorical series, back to the original column, replacing it with an equivalent-but-slimmer version:

```
df['Games'] = df['Games'].astype('category')
```

How much memory does that action save us? We can find out by running `memory_usage` once again:

```
df.memory_usage(deep=True).sum()
```

Sure enough, memory usage has gone down to 168,248,812 bytes, or a bit more than 160 MB. In other words: We've trimmed 15 MB of storage from our data frame, simply by turning the `Games` column from a string to a category.

Which columns should we attack first? Well, we want those in which the same strings are often repeated. Consider this code:

```
(df.count() / df.nunique()).sort_values(ascending=False)
```

Here, we divide the number of non-null rows in each column by the number of distinct values in that column. The higher the number, the more times the same string is repeated, and thus the greater the memory savings we can achieve by switching the column to a category. I then used

`sort_values(ascending=False)` to sort them in order of priority.

I decided to choose all categories with a dtype of object, in which a value was repeated at least 100 times. This led me to the following code:

```
for column_name in ['Sex', 'Season', 'Medal', 'City', 'Games', 'S
    print(column_name)
    df[column_name] = df[column_name].astype('category')
```

The result? A data frame that's just over 33 MB in size. Which means that after only a handful of lines of code that took several seconds to execute, I've cut the memory requirement to about 20% of its original value. That seems like an extremely worthwhile use of my time.

But wait a second: This method creates the category based on the data that's already in the series. What if I know that the series might include other values in the future, even if they're not in the original data set? Here's a simple example:

```
s = Series(['a', 'b', 'c', 'a', 'b', 'c', 'c', 'c']).astype('cate
```

I'll now try to set one of the values to 'd':

```
s.loc[7] = 'd'
```

This fails with a `TypeError` exception, telling me that we cannot set a value that wasn't included in the category.

We can solve this problem by creating the category before creating the series (or column of the data frame), including all of the possible values it might contain. Then we can ask pandas not to create the category with `astype`, but rather to assign the specific category type that we've defined, with all of its values. Let's first see how this might work with our above series:

```
abcd_category = pd.CategoricalDtype(['a', 'b', 'c', 'd'])
s = Series(['a', 'b', 'c', 'a', 'b', 'c', 'c', 'c']).astype(abcd_
s.loc[7] = 'd' # Success!
```

In the above code, we created a new category, with all of its values, by calling `pd.CategoricalDtype`. Then, when we called `astype`, we passed the

category that we had created, rather than asking pandas to create a new, anonymous category. We can do the same in our Olympics data frame:

```
medals_category = pd.CategoricalDtype(['Gold', 'Bronze', 'Silver'])
df['Medal'] = df['Medal'].astype(medals_category)
```

Another potential benefit of creating this category type is that if we need the same category for multiple columns, we can save even more memory by sharing the category.

## 11.2 Exercise 48: Categories

We've explored New York City's parking tickets on several previous occasions in this book, but we were always concerned by how much memory the full data set would require. Indeed, if I load the entire data set onto my computer, it uses a **lot** of memory—about 18 GB. I'd like to crunch that down to a much smaller number by turning many of the columns into categories.



Note

Because I realize that not everyone reading this book has many gigabytes of RAM to spare, I'm asking you to limit the number of columns you load for this exercise. If you are fortunate enough to have such a computer, though, I encourage you to load the entire data set into memory, and to pare the columns down using the same techniques. If you're like me, you'll be amazed by how much memory categories can save you.

Similarly, if your computer cannot load the columns that I have specified for this exercise, feel free to cut them down, so that things will fit into memory.

- Read the NYC parking violations data into a data frame. Only load the following columns: 'Plate ID', 'Registration State', 'Vehicle Make', 'Vehicle Color', 'Vehicle Make', 'Violation Time', 'Street Name', and 'Violation Legal Code'.
- How much memory is being used by the data frame you've created?
- Turn each column into a category.

- What types are your columns now?
- How much memory does your data consume now?
- How much memory have you saved thanks to using categories?

## 11.2.1 Discussion

This exercise has fewer steps than many of the recent ones we've done, for two reasons: First, I wanted to show you how easily we can create and work with categories. Second, when we're dealing with large amounts of memory, even the fastest and most tricked-out computers can take a while to calculate things.

With that in mind, let's go through the code, and see what we can do. First, we'll load the data set, limiting ourselves to the eight columns that I asked for:

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

df = pd.read_csv(filename, usecols=['Plate ID', 'Registration St
                                   'Vehicle Make', 'Vehicle Color', 'Vehicle
                                   'Violation Time', 'Street Nam
                                   'Violation Legal Code'])
```

You'll likely get a `DtypeWarning` from pandas, indicating that because a column has mixed types, something that we've seen before. That's fine; we'll soon be turning it into a category, in any event.

Next, I asked you to calculate how much memory the data frame is using, total. There are actually two ways to do this. The first is to run the method that I showed to you, `memory_usage`, passing the keyword argument `deep=True`. Running this method gives us a series, in which the index contains the data frame's column names, and the values show how much memory is being used by each column:

```
df.memory_usage(deep=True)
```

On my computer, I get the following result:

Index	128
Plate ID	798282162

```
Registration State      737248306
Vehicle Body Type      758166224
Vehicle Make           768611575
Violation Time         774726961
Street Name            879156216
Violation Legal Code   515644296
Vehicle Color          735089399
dtype: int64
```

According to this report, each of the columns in my data frame requires more than half a gigabyte of RAM. Even in our modern era of cheap, plentiful RAM, this is still a large data set—and given the alternative, there’s no reason for us to use all of this memory.

I want to remind you that it’s important to always use `deep=True` if you truly want to know the size of your data frame. If we hadn’t passed `deep=True`, then we would have gotten something like this:

```
Index                  128
Plate ID               99965872
Registration State     99965872
Vehicle Body Type     99965872
Vehicle Make          99965872
Violation Time        99965872
Street Name           99965872
Violation Legal Code  99965872
Vehicle Color         99965872
dtype: int64
```

Notice how all of the columns, aside from the index, have the same size, namely 99,965,872 bytes—basically 100 MB. Not a small amount of memory, but it’s far less than the actual size of our data in memory, whose size we calculated using `deep=True`.

Why does pandas not run `deep=True` all of the time? Because it takes substantially longer

Of course, this just gives us the size of each individual column. But since we got the results back as a series, we can add the values together using `sum`:

```
df.memory_usage(deep=True).sum()
```

On my computer, the result is 5,966,925,267, or about 6 GB.

Next, I asked you to turn each of these columns into a category. Remember that given a column named `colname`, you can turn it into a category with:

```
df['colname'] = df['colname'].astype('category')
```

When you do this, pandas removes NaN values in the column, looks at the remaining unique values, builds a new category object from it, and then uses that category to assign values. While it still appears as though the values are there, as before, pandas has actually replaced them with much-smaller integers, storing each of the strings a single time.

How can I perform this transformation on each column? My suggestion is to use a `for` loop. You might be surprised to see this suggestion, given that I often point out that if you're using a `for` loop in pandas, you're almost certainly doing something wrong. But that's if you're trying to perform a calculation on each row; for such purposes, pandas has a lot of functionality that'll generally be faster than any loop you run. Because so much of the back-end data uses NumPy, pulling the data into Python data structures will use significantly more memory than taking advantage of its vectorized, compiled, and optimized systems.

But this case is quite different: Here, I'm interested in performing one vectorized operation per column. There isn't any vectorizing to be done across the columns. For this reason, a `for` loop is perfectly reasonable. The index object that we get back from `df.columns` is iterable, allowing us to get each column name, one at a time. I thus wrote:

```
for one_colname in df.columns:
    print(f'Categorizing {one_colname}...')
    df[one_colname] = df[one_colname].astype('category')
    print('\tDone.')
```

Notice that I put two calls to `print` inside of the `for` loop, once before starting the transformation and once after. This is because the creation of a category can take some time, and I thought it would be useful to know when pandas was starting to work on a column, and when it had finished with it. In addition, if something goes wrong while creating the columns, I'll know

exactly where we were when the problem took place.

After performing this transformation, I wanted to get confirmation that things had changed. By running `dtypes` on our data frame, we can see precisely what type each column has:

```
df.dtypes()
```

Sure enough, pandas showed me that all of the columns had been changed to have category types. But what impact did that have on the memory usage? We can check, once again, by asking for a deep memory check:

```
orig_mem = df.memory_usage(deep=True).sum()
```

This time, on my computer, I get the value 574,455,678—still half a gigabyte of RAM, but that's a far cry from the original value we got, namely 6GB. In other words, it would seem that we have cut down our memory usage by about 90 percent! And indeed, if we perform a quick calculation:

```
new_mem / orig_mem
```

We get a result of 0.096, meaning that we are indeed using approximately 10 percent of the original data frame's memory, while still using the same data and enjoying the same benefits from it.



#### Note

The `df.info` method returns a summary of information about the data frame, including the total memory usage. By default, it doesn't do a "deep" memory check; in such cases, and if there are object columns, the memory will be returned with a + sign following the number. You can avoid the +, and get a precise calculation, by passing `memory_usage='deep'` as a keyword argument to `info`:

```
df.info(memory_usage='deep')
```

This will give you a summary of the total memory used.

## 11.2.2 Solution

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

df = pd.read_csv(filename,
                 usecols=['Plate ID', 'Registration State',
                          'Vehicle Make', 'Vehicle Color',
                          'Vehicle Body Type', 'Violation Time',
                          'Street Name', 'Violation Legal Code'])

orig_mem = df.memory_usage(deep=True).sum() #1

for one_colname in df.columns: #2
    print(f'Categorizing {one_colname}...')
    df[one_colname] = df[one_colname].astype('category') #3
    print('\tDone.')

df.dtypes() #4

new_mem = df.memory_usage(deep=True).sum() #5

print(new_mem / orig_mem) #6
```

## 11.2.3 Beyond the exercise

- Without calculating: Of the columns we loaded, which would make less sense to turn into categories? Once you've thought about it, calculate how many repeated values there are in each column, and determine (more formally) which would give the biggest ROI in using categories.
- In Exercise 25, we saw that the vehicle makes and colors were far from standardized, with numerous misspellings and variations. If we were to standardize the spellings before creating categories, would that make any effect on the memory savings we gain from categorization? Why or why not?
- Read only the first 10,000 lines from the CSV file, but all columns. Show the 10 columns that will most likely benefit greatest from using categories?

### Apache Arrow

There's no doubt that CSV files are convenient to work with. Not only does

every programming language and data-analysis system knows how to read from and write to them, but they're readable by people, too. Heck, we can even go in and edit CSV files by hand, when we need to. The same could be said for JSON, which has also become popular in the last few years. JSON can handle more complex data structures than CSV, but is still a text-based format.

The fact that CSV and JSON are human readable almost inherently makes them less efficient for computers: They take up more space on disk than binary formats, and also take longer to read and write. A number of binary formats exist, but none has pushed out the others as a standard.

Fortunately, we have an increasingly viable binary option—Apache Arrow, along with its file format, known as Feather. Apache Arrow is meant to be a new backend in-memory storage system not just for pandas, but also for other data-analysis libraries and languages, including the popular R language and the Spark distributed library. Arrow is designed to handle the data types, and data-storage needs, associated with data analysis. It can handle the data types that we're used to, from integers and floats to strings and dictionaries. It can even handle categories, of the sort that we saw in Exercise 48. It uses a number of tricks to use less memory than the standard NumPy back end.

Beyond the work being done to make Apache Arrow a fast and useful in-memory database, there is also an on-disk serialization format for that database, known as "feather." Feather files take advantage of the data types in Arrow, along with compression and binary storage. They are thus smaller in size, and faster to both read and write, than either CSV or JSON.

To write a pandas data frame to a feather-formatted file, you can use the `to_feather` method, which works similarly to `to_csv` and `to_json`:

```
df.to_feather('mydata.feather')
```

You can similarly read from a feather-formatted file into a data frame using the `pd.from_feather` method, which works similarly to `from_csv` and `from_json`:

```
df = pd.from_feather('mydata.feather')
```

Because pandas can read from and write to feather files faster than CSV or JSON, your projects would likely benefit from turning files from CSV and JSON into feather at the first opportunity. The larger the data set, the greater the benefit you'll have from working in this way.

Feather provides another potential way to speed up your work in pandas, even if you don't use Arrow or the feather format directly: When reading from a CSV file, you can traditionally choose from two different parsing engines, one written in Python and the other written in C. The Python engine is more flexible and offers more features, but it's slower than the C-language parser. By default, pandas tries to use the C parser, unless you specify an option that it cannot handle, in which case it switches to C. You can explicitly specify the engine you want when reading the CSV file:

```
df = pd.read_csv('mydata.csv', engine='python')
```

While it is still considered experimental as of this writing, you can now pass a third value to the engine keyword argument, namely the string 'pyarrow'. If you do that, then pandas will use the Arrow architecture and library to read the CSV file, potentially speeding up its reading into memory. If you find that the data is loading slowly, you might want to try this option. In my experience, this can significantly improve the rate at which CSV data is read into a data frame.

Over the coming years, I expect that the Arrow backend, as well as the feather format, will become increasingly common among users of pandas and other open-source data analysis tools. CSV and JSON aren't going away, but as data sets grow in size, feather's faster speed and cross-platform compatibility will become more dominant, and should assume a greater role in your arsenal.

### **11.3 Exercise 49: How much faster?**

Each file format has its own advantages and disadvantages, among them being the speed with which you can read and write data. Given a data frame, is it faster to write it as a CSV, JSON, or feather file? (If you read the above sidebar on Apache Arrow and feather, you might have a good sense of the

answer.) How much of a difference is there? And is there a significant difference in speed when reading CSV, JSON, and feather files into a data frame?

In order to understand that, I'm asking you to do the following:

- Load the New York parking data CSV file into a data frame.
- Write that data frame out to the filesystem, in each of three different formats—CSV, JSON, and feather. Time the writing of each format, and print the format along with the number of seconds it took to write to it.
- How big are the files you've created?
- Then, go through each file you just created, reading it into a data frame. Once again, time how long the loading takes, and print that timing alongside the format name.



#### Note

The New York parking data set is quite large, and might overwhelm computers with less than 32 GB of free memory. If you're working on such a computer, then I encourage you to use the `usecols` keyword parameter to reduce the number of columns read into the data frame at the start of the exercise. You might see less of a difference between writing to, and then reading from, the various formats—but at least you'll be able to finish the exercise.

### 11.3.1 Discussion

The first thing I asked you to do was load the New York parking violations data set for 2020. I'm going to assume that your computer has enough memory to load the entire thing, which I did as follows:

```
filename = '../data/nyc-parking-violations-2020.csv'  
df = pd.read_csv(filename, low_memory=False)
```

Notice that I passed the `low_memory=False` keyword argument. This told pandas that I had enough RAM on my computer that it could look through all of the rows in the data set, when trying to determine what dtype to assign to

each column.

With my data frame in place, I can thus begin writing to different formats, timing how long each takes.

But of course, that means we'll need some way to keep track of time. Python's `time` module, part of the standard library, provides a number of different methods that could theoretically be used, but it's generally considered best to use `time.perf_counter()`. This function uses the highest-resolution clock available, and returns a float indicating a number of seconds. The number returned by `perf_counter` should not be relied upon for calculating the current date and time—but if used within the same program, it can be used to measure the passage of time, which is precisely what we'll want to do.



#### Note

Python's standard library also includes the `timeit` module ([docs.python.org/3/library/timeit.html?#module-timeit](https://docs.python.org/3/library/timeit.html?#module-timeit)), which includes a number of utilities for benchmarking. I decided not to use `timeit`, in no small part because that module runs code several times. Each of our runs will take long enough that this didn't seem like a good fit. But `timeit` is a good tool to know about for benchmarking your code, and we'll use it in Exercise 50.

I wanted to try writing to CSV, JSON, and feather formats. In theory, I could have written code that looked like this:

```
df.write_json('parking-violations.json')
df.write_csv('parking-violations.csv')
df.write_feather('parking-violations.feather')
```

But of course, I wanted to find out how long each one took. So I could add some benchmarking code above and below each of our formats:

```
start_time = time.perf_counter() #1
df.write_json('parking-violations.json')
end_time = time.perf_counter()
total_time = end_time - start_time
print(f'\tWriting JSON: {total_time=}')
```

```

start_time = time.perf_counter() #2
df.write_csv('parking-violations.csv')
end_time = time.perf_counter()
total_time = end_time - start_time
print(f'\tWriting CSV: {total_time=}')

start_time = time.perf_counter() #3
df.write_feather('parking-violations.feather')
end_time = time.perf_counter()
total_time = end_time - start_time
print(f'\tWriting feather: {total_time=}')

```

The above code will work, and will do the job. But it also violates an important rule of programming: "Don't Repeat Yourself," often abbreviated as "DRY." We're basically doing the same thing three times. If we can consolidate that code into a loop, our code will be cleaner, easier to read, easier to debug, and easier to extend. But how can we do that? After all, we're calling three different methods.

This is where some knowledge and understanding of Python, and not just pandas, comes in handy: We can create a dictionary in which the keys are the file formats, and the values are the methods we want to use to write the data frame. That's right—you can store any Python object, including a function or method—as the value in a dict. I can thus say:

```

root = 'parking-violations'
write_methods = {'JSON': df.to_json,
                 'CSV': df.to_csv,
                 'feather': df.to_feather
                }

for one_format, method in write_methods.items():
    print(f'Saving in {one_format}')
    start_time = time.perf_counter()
    write_methods[one_format](f'parking-violations.{one_format}.lo
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tWriting {one_format}: {total_time=}')

```

My for loop here iterated over the dict, getting each key (a string, stored in `one_format`) and value (method, containing the method to be run) from the

`write_methods` dict. I printed the current format, just for debugging purposes, and then ran `time.perf_counter()`, getting back the current time (more or less) in seconds.

I then retrieved the write method with `write_methods[one_format]`, invoking it on the filename, via an f-string.

After writing the file to disk, I then called `time.perf_counter()` again, storing the difference in `total_time`, which I then printed.

On my computer, I got the following results from running the above code:

```
Saving in JSON
    Writing JSON: total_time=46.29149689315818
Saving in CSV
    Writing CSV: total_time=114.35314526595175
Saving in feather
    Writing feather: total_time=7.929971480043605
```

In other words, it took about 114 seconds (nearly two minutes) to write our data frame to a CSV file. It took 46 seconds to write the same data to JSON. But it took just under 8 seconds—or about 14 times faster!—to write the same data to feather. If that doesn't convince you to consider using feather, I'm not sure what will.

Notice that in order for this code to work, I had to define `df`, the data frame, **before** the `write_methods` dictionary was defined. I also used `one_format.lower()` to take the format name and ensure that it was only in lowercase letters.

How big were the files that I created? Here, I again decided to rely on Python's standard library. We've already seen the `glob.glob` function in previous exercises; here I use it to retrieve all of the filenames that start with the value of our `root` variable. But I then want to get the size of each file, something I can do easily with `os.stat`. This function returns a special data structure that's modeled on Unix's `stat` functionality. In Python, we can get the size of the file, in bytes, by retrieving the `st_size` attribute from the value we get back from `os.stat`:

```
for one_filename in glob.glob(f'{root}*'):
```

```
print(f'{one_filename:27}: {os.stat(one_filename).st_size:,}')
```

Inside of the f-string, I used two tricks to adjust the way in which the values were formatted:

- I told the f-string to pad `one_filename` with spaces, such that each filename would use 27 characters. This helped to ensure that the results would line up.
- I told the f-string to add commas before every three digits in the integer it was displaying, thus making them more readable.

The result, on my computer, was:

```
parking-violations.json      : 8,820,247,015
parking-violations.csv       : 2,440,860,181
parking-violations.feather   : 1,466,535,674
```

We can see here that the CSV file was about 2 GB in size, the JSON file was about 8 GB (!) in size, and Apache Arrow's feather format was just over 1 GB in size. This isn't the only reason why writing feather files is faster, but it's certainly one of them; at the end of the day, pandas had to write one eighth as much data to disk.

However, I also wanted you to benchmark reading these files back from the filesystem. I decided to use the same technique as before, namely creating a dictionary (this time, called `read_methods`), containing the file extensions and the methods I'd want to run. The code then was as follows:

```
read_methods = {'JSON': 'read_json',
                'CSV': 'read_csv',
                'feather': 'read_feather' }

for one_format, method in read_methods.items():#1
    print(f'Reading from {one_format}')
    start_time = time.perf_counter()
    df = read_methods[one_format](
        f'parking-violations.{one_format.lower()}') #2
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tReading {one_format}: {total_time=}')
```

Once again, I iterated over a dictionary, getting each key (a string, stored in `one_format`) and value (method) from the `read_methods` dict. I printed the current format, and then ran `time.perf_counter()`. I then retrieved the appropriate read method with `read_methods[one_format]`, and invoked the method that I got on the appropriate filename. After reading the file into a data frame, I then called `time.perf_counter()` again, storing the difference in `total_time`, which I then printed.

If you're like me, then you'll likely get the `DtypeWarning` we've previously discussed. For our purposes, I decided to ignore it, in no small part so that I could avoid having to worry about which method, and which format, was being read. But the benchmarking results were as follows:

```
Reading from JSON
    Reading JSON: total_time=469.92014819500037
Reading from CSV
    Reading CSV: total_time=35.20077076088637
Reading from feather
    Reading feather: total_time=9.132312984904274
```

This time, the JSON file took the longest to read into memory, at a hefty 469 seconds, or nearly 8 minutes. In second place, and taking less than 10% of the time, was CSV, at 35 seconds. But the speed champion remained feather, taking just over 9 seconds.

From this simple demonstration, it seems pretty clear that Apache Arrow, and its feather format, are significantly faster for reading and writing than both CSV and JSON. Which doesn't mean that you can or should move everything to feather—but that it has a number of clear advantages, both in terms of speed and in its footprint on the filesystem.

### 11.3.2 Solution

```
import glob
import os

filename = '../data/nyc-parking-violations-2020.csv'
df = pd.read_csv(filename, low_memory=False)

root = 'parking-violations'
write_methods = {'JSON': 'to_json',    #1
```

```

        'CSV': 'to_csv',
        'feather': 'to_feather' }

for one_format, method in write_methods.items(): #2
    print(f'Saving in {one_format}')
    start_time = time.perf_counter()
    write_methods[one_format]( #3
        f'parking-violations.{one_format.lower()}') #4
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tWriting {one_format}: {total_time=}')

for one_filename in glob.glob(f'{root}*'): #5
    print(f'{one_filename:27}: {os.stat(one_filename).st_size:,}')

read_methods = {'JSON': 'read_json',
                'CSV': 'read_csv',
                'feather': 'read_feather' }

for one_format, method in read_methods.items():#7
    print(f'Reading from {one_format}')
    start_time = time.perf_counter()
    df = read_methods[one_format](
        f'parking-violations.{one_format.lower()}') #8
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tReading {one_format}: {total_time=}')

```

### 11.3.3 Beyond the exercise

- If we read the CSV file using the "pyarrow" engine, do we see any speedup? That is, can we read CSV files into memory any faster if we use a different engine?
- If we specify the dtypes to read\_csv, does it take more time, or less, than without doing so?
- How much memory does our data frame take in as a pandas data frame? How much memory does it require as an Arrow table?

#### Speeding things up with eval and query

Over the course of this book, I've emphasized a number of techniques that

you should use to speed up your pandas performance:

- Never use standard Python iterations (for loops and comprehensions) on a series or data frame
- Take advantage of broadcasting
- Use the `str` accessor for anything string related
- Use the smallest dtype you can, without sacrificing accuracy
- Avoid double square brackets when setting and retrieving values
- Load only those columns that you really need for your analysis
- Columns with repeated values should be turned into categories
- Use a binary format, such as feather, for data you'll repeatedly save or load

Even after using all of these techniques, you might find that your queries are still running slowly, or using lots of memory. This often occurs when performing an arithmetic operation on two columns, each of which contains many rows. A related problem is when you're broadcasting an operation on a scalar and a series. While pandas takes advantage of the high-speed calculations in NumPy, much of the work is still being done within the Python language, which is slower to execute than C.

Another problem occurs when you're creating a boolean series, for use as a mask index, based on several conditions. It's certainly convenient to use `&` and `|` to combine your conditions with logical "and" and "or", but behind the scenes, pandas has to create multiple boolean series, which are then combined. If you have 1 million rows in your original column, then combining three conditions will end up creating at least 3 million rows in temporary series, before combining and applying them together.

We can avoid these problems, as well as make our queries more readable, using the `query` method that I introduced back in chapter 2, as well as two versions of the more general `eval` method. These reduce the memory needed in queries using `|` and `&`, and can often execute expressions in a library known as `numexpr`. The combination of reduced memory and increased speed can sometimes give dramatically faster results, while also using fewer resources.

However, it's important to understand that these methods are not cure-alls for your performance problems:

- Using them on small data frames, with fewer than 10,000 rows, will often result in slower performance, not in faster performance.
- Often, the bottleneck in your performance is in assignment or retrieval of elements, not in the calculation. Which means that there won't be a speed boost in such cases.
- You'll need to install the `numexpr` package from PyPI, and then explicitly tell pandas to use it. If you don't make this explicit, then pandas will use its default Python-based engine for parsing the query string, resulting in no speedup.
- Anything which doesn't involve calculations, comparisons, and boolean operators will either raise an exception or run at the standard (non-enhanced) speed.

Let's start by looking at the query for data frames. We'll then talk about two versions of `eval`, which are part of the same family.

Given a data frame `df`, the method `df.query` allows you to describe which rows you want to get back from `df`. The description is passed as an SQL-like string in which the columns can be named as if they were variables. The result of the query will be a data frame, a subset of `df`, with all of the columns from `df` and those rows for which the comparison returned a `True` value. For example, given a data frame `df` with numeric columns `a` and `b`, in which we want all rows where `a` is greater than 100 and `b` is equal to 50, we would normally say:

```
np.random.seed(0)
df = DataFrame(np.random.randint(0, 1000, [5,5]),
               index=list('vwxyz'),
               columns=list('abcde'))

df.loc[((df['a'] > 100) &
        (df['b'] < 700))]
```

But using `df.query`, we can instead write:

```
df.query('a > 100 & b < 700')
```

The version using `query` might run faster. But in almost all cases, it'll use less memory, because it won't have to create two separate, temporary boolean series: One for `a > 100` and another for `b == 50`. We might not see

these boolean series when running a traditional query, but they're there, and can use a great deal of memory without you realizing it.

I should add that some people prefer to use `df.query` for all of their pandas work, because of its readability and reduced memory use.

A related data frame method is `df.eval`, which allows us not only to retrieve from a data frame (as in `df.query`), but also to perform other actions, including broadcasting and assigning. For example:

```
df.eval('(a + b) * 3')
```

The above code adds columns `a` and `b` together, then multiplies the new series by 3, via broadcasting. The returned value is a series. What if we were to pass the same code as we used before, with `df.query`?

```
df.eval('a > 100 & b < 700')
```

The above returns a boolean series. Whereas `df.query` applies that boolean series to `df`, `df.eval` returns the boolean series itself, and allows us to apply it if and when we want to do so. We can even add a new column (or update an existing one) by assigning to a column name:

```
df.eval('f = d + e - c')
```

Using a triple-quoted string, you can perform multiple assignments (not conditions) `df.eval`:

```
df.eval('''
f = d + e - c
g = a * 2
h = a * b
''')
```

The third, and final method that allows you to use less memory, speed up computation, and write more readable queries is `pd.eval`. Notice that this is a top-level function in the `pd` namespace, rather than a method we run on a specific data frame. We can use `pd.eval` instead of `df.eval`, although we'll need to explicitly state the name of the data frame we're working on. For example, we can say:

```
pd.eval('df[df.a > 100 & df.b < 700]')
```

Notice that when using `pd.eval`, you'll almost certainly need to use the dot syntax with columns, rather than the square-bracket syntax that I have generally used in this book—so to retrieve column `a` from data frame `df`, you'll need to say `df.a`, rather than `df['a']`. As a result, this also means that your column names cannot contain spaces in them.

The above code will return all of the rows of `df` in which `a` is greater than 100 and `b` is less than 700, as before. However, we have written the query as a string, which is passed to `numexpr`. That package will, as we've seen, use less memory and (usually) result in better performance. Note that a call to `df.eval` is translated into a call to `pd.eval`, which means that you can probably get better performance if you just call `pd.eval`. That said, the convenience of the syntax in `df.eval` is hard to beat.

As with `df.eval`, you can assign to one or more columns in the string you pass to `pd.eval`. But because we're invoking `pd.eval`, the data frame on which the assignment should take place isn't known to the system. You must set it by passing the `target` keyword argument. The assignment is reflected in the data frame that is returned:

```
pd.eval('f = df.d + df.e - df.c', target=df)
```

So, when should you use each of these? Again, the biggest wins are likely to be with compound queries (using `&` and `|`), on large data frames. The larger the data frame and the more complex the query, the bigger the speed boost that you might see—but even if you don't, you'll almost certainly be using less memory.

Meanwhile, here's a quick recap on each of these three functions:

- To retrieve selected rows from a data frame, use `df.query`
- To assign multiple columns, or to perform either queries or assignments on a data frame, use `df.eval`.
- To work on multiple data frames, use `pd.eval`. It doesn't handle multiline assignments, and the syntax makes it a bit uglier, though.

## 11.4 Exercise 50: query and eval

In this exercise, we'll look through New York parking tickets one final time, running queries using the traditional `df.loc` accessor, and then also using `df.query` and `df.eval`. For each of these questions, I'd like you to run the query via `timeit`, allowing us to compare the executing time needed for the various types of queries. Specifically, I'd like you to:

- Load the New York parking data CSV file into a data frame. We'll only need the following columns: `Plate ID`, `Registration State`, `Plate Type`, `Feet From Curb`, `Vehicle Make`, and `Vehicle Color`.
- Rename the columns to `pid`, `state`, `ptype`, `make`, `color`, and `feet`. (This will make it easier to use `df.eval`.)
- Find all of the cars whose registration state is from New York, New Jersey, or Connecticut, using `.loc`.
- Find all of the cars whose registration state is from New York, New Jersey, or Connecticut, using `df.query`.
- How much faster was it to use `query`? # Use `isin` to search for the states. How does this technique compare?
- Perform each of the following queries using `df.loc`, `df.query`, and `df.eval`, all within `timeit`. In each case, which type of query ran the fastest?
  - Find cars from New York.
  - Find cars from New York with passenger (PAS) plates.
  - Find white cars from New York with passenger (PAS) plates.
  - Find white cars from New York with passenger (PAS) plates that were parked > 1 foot from the curb.
  - Find white Toyota-brand cars from New York with passenger (PAS) plates that were parked > 1 foot from the curb.
- Which type(s) of query would appear to run the fastest?

### 11.4.1 Discussion

In this exercise, I wanted you to learn several things: \* How to formulate the same query using `.loc`, `df.query`, and `df.eval` \* How to use `timeit` to time

your queries, and thus compare their relative speeds \* What might lead a query to be slower \* Some of the syntactic issues associated with alternative query mechanisms

The first thing I asked you to do was load a number of columns from the New York parking-ticket dataset, much as we've often done in this book:

```
df = pd.read_csv(filename,
                 usecols=['Plate ID', 'Registration State',
                         'Plate Type', 'Feet From Curb',
                         'Vehicle Make', 'Vehicle Color'])
```

There is nothing inherently wrong with loading the data in this way. However, when we use `pd.query` and `pd.eval`, it's often annoying to have column names that includes spaces in them. Yes, we can use backticks, but it's more convenient to give them names that'll allow us to treat them as variables inside of the query string. So while there's nothing technically wrong with loading the data as I've done here, I'll then want to set the headers to be single-word names. I can do that by assigning a list of strings to `df.columns`:

```
df.columns = ['pid', 'state', 'ptype', 'make', 'color', 'feet']
```

Now, you might be thinking that it would be more effective to set these names as part of the call to `read_csv`. After all, `read_csv` has a `names` parameter, which takes a list of strings that are assigned to the newly created data frame. However, things get tricky if we want to rename the columns (with `names`) and also load a subset of the columns (with `usecols`). That's because passing a value to `names` means that you need to use those names, rather than the original ones from the file, when choosing columns in `usecols`. And you can only do that if you name all of the columns, which is rather annoying.

Actually, there is another way to do it: You can specify which columns you want by passing a list of integers to `usecols`. pandas will select the columns at those indexes. You can then assign them names by passing a value to the `names` parameter. Here's how I would do that:

```
df = pd.read_csv(filename,
                 usecols=[1, 2, 3, 7, 33, 37],
```

```
names=['pid', 'state', 'ptype', 'make', 'color',
```

Will this work? Yes, it will, and in many cases, it might be the preferred way to go. However, I have two problems with it: First, I find it somewhat annoying to find the integer positions for the columns we want to load. And secondly, when I ran this code on my computer, I got the "low memory" warning that we've sometimes seen in previous examples. I thus decided to avoid the annoyance of finding the desired columns' numeric locations and the low-memory warning, and to use the two-step column renaming that appears in the solution.

With our data frame in place, we can start to perform some queries. One of the main points of this exercise is to get comfortable timing queries, in order to find out how quickly they run. Python provides the `timeit` module, which you can use in standard programs, but Jupyter provides a special `%timeit` magic method that can be used inline, inside of Jupyter cells. You can say:

```
%timeit myfunc(2, 3, 4)
```

In this example, `timeit` will run `myfunc(2, 3, 4)` a number of times, reporting the mean execution time along with the variation that it detected. Just how many loops `timeit` runs is determined by the code speed; something that takes a fraction of a second might run hundreds or even thousands of times, whereas something that takes more than a few seconds might be run only a handful of times.



#### Note

When using the `%timeit` magic command in Jupyter, don't forget:

- Your code must be written on a single line, just after the `%timeit` magic command. If you have more than one line, then wrap it into a function, and invoke that function.
- If you're timing a function, don't forget to put `()` after the function's name.

For the first task, I asked you to find all of the rows in `df` that were for parking tickets issued in New York ('NY'), New Jersey ('NJ'), or

Connecticut ('CT'), using both the traditional `.loc` accessor and using the query method. I also asked you to time each of these, for comparison.

I started with the traditional `.loc` accessor, combining three separate queries:

```
%timeit df.loc[(df['state'] == 'NY') | (df['state'] == 'NJ') | (
```

On my computer, this query took 1.84 seconds.

Consider everything that pandas had to do for this query:

- Compare each element in `df['state']` with 'NY'
- Compare each element in `df['state']` with 'NJ'
- Compare each element in `df['state']` with 'CT'
- Perform an "or" operation on the first two (New York and New Jersey) boolean series
- Perform an "or" operation on the result of the above "or" and the Connecticut series
- Apply that final boolean series to `df.loc` as a mask index

There's no doubt that with so many rows, each of these comparisons will take some time. Moreover, the "or" operations, resulting in a single boolean series, will also take quite a while. Using the query method won't help with the first part; we'll still need to perform the comparisons. However, by using query, we can dramatically reduce the number of "or" operations involved. That's because query uses the `numexpr` backend to perform such operations, which does them far more efficiently. How much more? Here's how I rewrote things to use query:

```
%timeit df.query("state == 'NY' or state == 'NJ' or state == 'CT'
```

On my computer, using query took only 1.03 seconds, about 0.8 seconds (or 45%) less than the original query. That's a pretty dramatic speed improvement, and points to how much query can improve our performance for certain queries.

However, the comparisons with each of the three state abbreviations also takes quite some time. Can we cut down on the number of comparisons? Yes, if we use the `isin` method on our column to search for a match within a

Python list:

```
%timeit df.loc[df['state'].isin(['NY', 'NJ', 'CT'])]
```

This query took even less time than the previous one, clocking in at 0.77 seconds on my computer. That represents a 58% speedup from the original query.

But wait: Maybe we can enjoy an even greater speedup if we use `query` and `isin` together. Let's give it a try:

```
%timeit df.query('state.isin(["NY", "NJ", "CT"])')
```

Unfortunately, this didn't seem to improve things; it took 0.80 ms on my computer—still better than the original queries, but not as good as simply using `isin`.

From this small comparison, we see that optimization of queries is rarely a matter of always using one particular technique. It requires a bit of thinking about what you're doing, considering what pandas is doing behind the scenes, and then performing some tests to check your assumptions. That said, we can conclude at least two things from these queries: First, that if you're combining queries with `|` or `&`, you'll likely get a decent improvement by using `query` rather than `loc`, thanks to the speedups provided by `numexpr`. Second, using `isin` will almost always be faster than combining multiple queries, because we're making a single comparison per row, rather than three.

Following this first set of queries, I asked you to perform a number of increasingly complex queries, each in three different ways: First using the traditional `loc` accessor, then by using `df.query`, and finally by using `df.eval`. I did this not only to give you some practice building queries in different ways, and in comparing the time that each takes, but also to see that the improvements you see using `query` and `eval` become more pronounced as the query becomes more complex.

For starters, I asked you to find all of the parking tickets given to cars with New York license plates. Here are the three queries, all together:

```
%timeit df.loc[(df['state'] == 'NY')]
```

```
%timeit df.query('state == "NY"')
%timeit df[df.eval('state == "NY"')]
```

On my computer, these gave me timings of 903 ms, 733 ms (19% faster), and 758 ms (17% faster), respectively. We thus already see that `loc` is the slowest of the three, with the use of `df.query` and `df.eval` coming in at almost the same.

But wait—the result of `df.eval` is a boolean series, one which we then apply to `df`. Perhaps, instead of using a mask index on `df`, I should do so on `df.loc`? Using `%timeit`, I can find out pretty quickly:

```
%timeit df.loc[df.eval('state == "NY"')]
```

Sure enough, I got the fastest result, albeit by just a hair, when using `df.loc` here, at 729 ms. In other words, it would seem that selecting via `df.loc` and a mask index will have better performance than just `df` and a mask index—something that I've seen elsewhere, too. The rest of my solutions in this exercise will all use `df.loc`, for a fairer comparison.

Next, I asked you to find passenger cars (i.e., with `ptype` equal to 'PAS') from New York. Here are my three solutions:

```
%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS'))]
%timeit df.query('state == "NY" & ptype == "PAS"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS"')]
```

Here, I got timings of 1.27 seconds for the traditional use of `df.loc`, vs. 965 ms for `df.query` (24% faster) and 924 ms for `df.eval` (27% faster). Here, we used `&` to combine the two boolean series that we got back from each comparison. Whereas we were able to speed up our "or" query above by using `isin`, there isn't an exact equivalent for "and" queries.

Next I asked you to expand the query further, thus narrowing the potential results, looking for white passenger cars from New York that had been ticketed. Again, we can compare my queries:

```
%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
%timeit df.query('state == "NY" & ptype == "PAS" & color == "WHIT
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" & color ==
```

This time, I got timings of 1.34 seconds, 728 ms for `df.query` (45% faster), and 727 ms for `df.eval` (also 45% faster). We can see that adding another condition slows down the traditional query a bit, but actually results in faster queries when using the numexpr backend.

Next, I asked to find tickets for white passenger cars from New York that were parked more than 1 foot away from the curb. Here are my queries:

```
%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') & (df['feet'] > 1))]
%timeit df.query('state == "NY" & ptype == "PAS" &
                color == "WHITE" & feet > 1')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" &
                       color == "WHITE" & feet > 1')]
```

In this case, I got timings of 1.31 seconds for the traditional query, 712 ms for `df.query` (45% faster), and 706 ms for `df.eval` (46% faster). Again, we can see that when the queries are complex, using the numexpr backend gives us a big speed advantage.

Finally, I asked you to find tickets given to white Toyota passenger cars, with license plates from New York state, that were parked more than 1 foot away from the curb. Here is how I wrote those queries:

```
%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') & (df['feet'] > 1) &
                (df['make'] == 'TOYOT'))]
%timeit df.query('state == "NY" & ptype == "PAS" &
                color == "WHITE" & feet > 1 &
                make == "TOYOT"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" &
                       color == "WHITE" & feet > 1 &
                       make == "TOYOT"')]
```

Here, I got timings of 1.75 seconds for the traditional query, 896 ms for `df.query` (49% faster), and 899 ms for `df.eval` (48% faster) for `df.eval`. The added condition slowed down all of the queries, but our numexpr backend continued to prove its worth, giving us the same answer at nearly twice the speed.

Does this mean that it's always worth using `df.query` or `df.eval`? I know

that there are pandas users who would say "yes," given that even in the simplest of cases, we saw a speedup. And in the most complex cases, the speedup was quite dramatic. So you could argue that since it doesn't matter much for simple queries on a short data set, but it matters a lot for complex queries on large ones, you should always use these techniques.

However, focusing on speed before you've really thought hard about the problem, and where the bottlenecks are, can be misleading. Remember that `df.query` returns all of the columns from a data frame—so if your data frame contains more columns than you'll want to get back, it might end up using lots of memory unnecessarily. By contrast, `df.loc` provides you not only with a row selector, but also with a column selector, for more flexibility. I thus tend to use `df.loc` for my queries while I'm still putting them together. When I'm done, I can then experiment with these techniques to see how to reduce memory and speed things up.

```
filename = '../data/nyc-parking-violations-2020.csv'
df = pd.read_csv(filename,
                 usecols=['Plate ID', 'Registration State', 'Plate
                          Vehicle Make', 'Vehicle Color'])
df.columns = ['pid', 'state', 'ptype', 'make', 'color', 'feet']

%timeit df.loc[(df['state'] == 'NY') | (df['state'] == 'NJ') | (
%timeit df.query("state == 'NY' or state == 'NJ' or state == 'CT'")
%timeit df.loc[df['state'].isin(['NY', 'NJ', 'CT'])]

%timeit df.loc[(df['state'] == 'NY')]
%timeit df.query('state == "NY"')
%timeit df.loc[df.eval('state == "NY"')]

%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS'))]
%timeit df.query('state == "NY" & ptype == "PAS"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS"')]

%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE'))]
%timeit df.query('state == "NY" & ptype == "PAS" &
                color == "WHITE"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" &
                color == "WHITE"')]

%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') & (df['feet'] > 1))]
```

```

%timeit df.query('state == "NY" & ptype == "PAS" &
                color == "WHITE" & feet > 1')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" &
                      color == "WHITE" & feet > 1')]

%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') & (df['feet'] > 1) &
                (df['make'] == 'TOYOT'))]
%timeit df.query('state == "NY" & ptype == "PAS" &
                color == "WHITE" & feet > 1 &
                make == "TOYOT"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" &
                      color == "WHITE" & feet > 1 &
                      make == "TOYOT"')]

```

## 11.4.2 Beyond the exercise

In this exercise, we

- In `df.query`, we can use the words `and` and `or`, rather than the symbols `&` and `|`, thanks to the `numexpr` library. Rewrite our final query using the words. Does this change the speed at all?
- I prefer measuring distance in meters, rather than in feet. I thus want to find all of the cars that were ticketed when they were more than 1 meter from the curb. Perform this query using the traditional `df.loc` and also using `df.query`. Which one runs faster?
- What if we modify our query, such that we look for cars that are  $> 1$  meter from the curb and the state is New York? Which query runs faster, and by how much?

## 11.5 Summary

Calculations and analysis with pandas are much faster than they would be in pure Python. Even so, when you're working with a large data set, you'll often want or need to reduce the memory footprint of your data frame, and use techniques that can improve the performance. In this chapter, we reviewed a number of techniques that you can use to speed up your queries and also use fewer resources, including:

- Choosing columns carefully

- Reducing memory usage with categories
- Reading data from feather format, rather than CSV
- Speeding complex queries with `df.query` and `df.eval`

If you've reached this part of the book, congratulations! You've successfully gone through all 50 exercises! I have no doubt that if you've made it this far, you have a much better, deeper understanding of pandas, what it can do, and how you can use it in a variety of situations. You should feel good about yourself, and confident about your ability to use pandas at work.

But wait, before you close the book forever: The next chapter contains a large project, in which I'll ask you to use all of the techniques from this book to analyze a large, real-world data set. I hope that you'll take the time to do the project, which will help to cement the lessons that you've learned, and help you to use pandas even more effectively in the future.

# 12 Final project

Congratulations! You've finished all of the exercises in this book. If you've gone through each one—and especially if you've gone through the "beyond the exercise" questions, as well—then I'm sure you have improved your Pandas skills quite a lot.

But before you go, I want to give you a final project. We'll explore the "college scorecard," a data set assembled by the US Department of Education about post-secondary (i.e., after high school) educational programs. The college scorecard allows us to see what programs schools offer, how many students they admit, what those students pay in tuition and fees, how many students graduate, and how much they can expect to earn after graduation. From looking at this data, we can better understand many different aspects of university education in the United States. I should add that with a data set this large and rich, there are many different questions that you could ask. After you finish answering the questions that I pose here, I strongly suggest that you try to explore the data set on your own, as well, asking (and answering) questions that you think are interesting and relevant.

## 12.1 Problem

Here is what I'd like you to do:

- Create a data frame (`institutions_df`) from the college scorecard cohorts-institutions CSV file. You'll only need to load the following columns: `OPEID6`, `INSTNM`, `CITY`, `STABBR`, `FTFTPCTPELL`, `TUITIONFEE_IN`, `TUITIONFEE_OUT`, `ADM_RATE`, `NPT4_PUB`, `NPT4_PRIV`, `NPT41_PUB`, `NPT41_PRIV`, `NPT45_PUB`, `NPT45_PRIV`, `MD_EARN_WNE_P10`, and `C100_4`.
- Load the CSV file for fields of study into another data frame (`fields_of_study_df`). Here, load the columns `OPEID6`, `INSTNM`, `CREDESC`, `CIPDESC`, and `CONTROL`.
- What state has the greatest number of universities in this database?
- What city, in which state, has the greatest number of universities in this

database?

- How much memory can we save if we set the CITY and STABBR columns in institutions\_df to be categories?
- Create a histogram showing how many bachelor programs universities offer.
- Which university offers the greatest number of bachelor programs?
- Create a histogram showing how many graduate (master's and doctoral) programs universities offer.
- Which university offers the greatest number of different graduate (master + doctoral) programs?
- How many universities offer bachelor's degrees, but not master's or doctorates?
- How many universities offer master's and doctoral degrees, but not bachelors?
- How many institutions offer bachelor's degrees whose name contains the term "Computer Science"?
- The CONTROL field describes the types of institutions in the database. How many of each type offer a computer-science program?
- Create a pie chart showing the different types of institutions that offer CS degrees
- What are the minimum, median, mean, and maximum tuitions for an undergrad CS degree? (We'll define this as a bachelor's program with the phrase "Computer Science" in the name.) When comparing tuition, use TUITIONFEE\_OUT for all schools.
- Describe the tuition again, but grouped by the different types of universities ("CONTROL")
- What is the correlation between admission rate and tuition cost? How would you interpret this?
- Create a scatter plot in which tuition is on the x axis, and admission rate is on the y axis, the median earnings after 10 years are used for colorizing, and we use the "Spectral" colormap. Where do the lowest-paid graduates show up on the graph?
- Which universities are in the top 25% of tuition, and also the top 25% of percentage with Pell grants? Print only the institution name, city, and state, ordered by institution name.
- NPT4\_PUB indicates the average net price for public institutions (in-state tuition) and NPT4\_PRIV for private institutions. NPT41\_PUB and

NPT45\_PUB show the average price paid by people in the lowest income bracket (1) vs. the highest income bracket (5) at public institutions. NPT41\_PRIV and NPT45\_PRIV show the average price paid by people in the lowest income bracket (1) vs. the highest income bracket (5) at private institutions. In how many institutions does the bottom quintile receive money (i.e., is the value negative)?

- What is the average proportion that the bottom quintile pays vs. the top quintile, in public universities?
- What is the average proportion that the bottom quintile pays vs. the top quintile, in private universities?
- Let's try to figure out which universities offer the best overall ROI (across all disciplines). What schools are in the cheapest 25%, but 10 years after graduation, students have the top 25% of salaries?
- How about private institutions?
- Is there a correlation between admission rates and completion rates? If a school is highly selective, are students more likely to graduate?
- Ten years after graduating, from what kinds of schools (private, for-profit, private non-profit, or public) do people earn, on average, the greatest amount?
- Do people who graduate from "Ivy Plus" schools earn more than the average private-school university graduate? If so, then how much more?
- Do people studying at universities in particular states earn, on average, more after 10 years?
- Create a bar plot for the average amount earned, per state, sorted by ascending pay
- Create a boxplot for the earnings by state.

### 12.1.1 Discussion

As I wrote above, the college scorecard data set includes a large number of facts and figures about American higher education, describing both the institutions and the students who learn there. To answer this set of questions, we'll only need to look at two CSV files: Information about the most recent cohorts of students who enrolled at and graduated from these institutions, as well as the fields of study that each institution offers. Some of our questions will require just one of these data sources, but others will require that we combine them into a single data frame.

To start, I asked you to load each of the CSV files into a data frame. You might have noticed that the files I've provided both have a `.csv.gz` suffix. This means that they are compressed with "gzip"—but you don't need to uncompress them before loading, because Pandas is smart enough to automatically do so when we run `read_csv`. I loaded the first data frame as follows, defining `institutions_df`:

```
institutions_filename = '../data/Most-Recent-Cohorts-Institution.
institutions_df = pd.read_csv(institutions_filename,
                             usecols=['OPEID6',
                                       'INSTNM', 'CITY', 'STABBR',
                                       'FTFTPCTPELL', 'TUITIONFEE_IN',
                                       'TUITIONFEE_OUT', 'ADM_RATE',
                                       'NPT4_PUB', 'NPT4_PRIV',
                                       'NPT41_PUB', 'NPT41_PRIV',
                                       'NPT45_PUB', 'NPT45_PRIV',
                                       'MD_EARN_WNE_P10', 'C100_4'])
```

I then loaded the other CSV file, with information about fields of study for the last few years, as follows, assigning it to `fields_df`:

```
fields_filename = '../data/FieldOfStudyData1718_1819_PP.csv.gz'
fields_of_study_df = pd.read_csv(fields_filename,
                                 usecols=['OPEID6', 'INSTNM',
                                          'CREDESC', 'CIPDESC', 'CONTROL'])
```

With these two data frames defined and in memory, I decided to start performing some queries. First, I wanted to know which state has the greatest number of universities in this database. This is a classic example of when to use grouping. We can group on the `STABBR` ("state abbreviation") column, running the `count` method. This will tell us how often each state appears in the data set. We also have to provide a second column, which is where the count will be reported. The choice doesn't matter, so I decided to go with `OPEID6`, the unique ID used for each institution:

```
institutions_df.groupby('STABBR')['OPEID6'].count()
```

The above query will tell me how often each state appears in the data set. But I was interested in finding which of the states had the greatest number of universities. To find that, I sorted the series by the values we got back, in descending order. The first row in this series was, by definition, the state with

the largest number—which I retrieved using `head(1)`:

```
institutions_df.groupby('STABBR')[  
    'OPEID6'].count().sort_values(  
    ascending=False).head(1)
```

According to this data, California has the greatest number of universities—quite a large number at 705. But then I decided to ask a slightly different question: Which city has the greatest number of universities?

It might seem, at first glance, that this query will be identical to the previous one, grouping by the `CITY` column rather than `STABBR`. But that would combine cities of the same name in different states. The solution requires that we group by two columns—first `STABBR` and then `CITY`. The combination will allow us to find which city, in which state, has the greatest number of universities:

```
institutions_df.groupby(['STABBR', 'CITY'])[  
    'OPEID6'].count().sort_values(  
    ascending=False).head(1)
```

Once again, we ask for the `count` method to be run on `OPEID6`, because we need to count on a non-grouping column. Also once again, we'll sort in descending order, and grab the top value. The answer is New York City, with 81 institutions of higher learning.

Considering that both state and city names are text data, and that they repeat so often, it makes sense to consider how much memory we might save by turning the `STABBR` and `CITY` columns into categories. But as always when trying to optimize, we should measure our results, to know whether our efforts were worthwhile.

I thus asked you to find out how much memory our data frame was already using. The easiest way to find this is to by running `memory_usage` on a data frame. Don't forget to pass the `deep=True` keyword argument. This will return the total memory usage of each column, including the objects to which it refers. (As we saw in Chapter 11, that argument can make a huge difference!) Here's how I can calculate that, and then print it:

```
pre_category_memory = institutions_df.memory_usage(deep=True).sum
```

```
print(f' {pre_category_memory:,}')
```

First, I calculate the total memory usage, and assign it to `pre_category_memory`. Then, in order to print the number with commas between the digits—and yes, I’m showing off a bit here—I print it within an f-string, using a single comma (,) as the format specifier, after the colon (:).

I then turn both the `STABBR` and `CITY` columns into categories:

```
institutions_df['CITY'] = institutions_df[
    'CITY'].astype('category')
institutions_df['STABBR'] = institutions_df[
    'STABBR'].astype('category')
```

Now that this has been done, how much memory did we save?

```
post_category_memory = institutions_df.memory_usage(
    deep=True).sum()

savings = pre_category_memory - post_category_memory
print(f' {savings:,}')
```

On my computer, the savings is calculated as 579,371 bytes—meaning, we managed to reduce memory usage by approximately one third by turning these two columns into categories. Not a bad gain for a few seconds of coding, I’d say.

Next, I asked you to create a histogram indicating how many programs are offered by each university. That is, I’d like to know how many universities offer 10 programs, how many offer 20, how many offer 30, and so forth.

In order to create such a histogram, we first need to count the number of different programs that each university offers. We start by looking at `fields_of_study_df`, and retrieving only those rows for which the `CREDESC` value is 'Bachelors Degree':

```
fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree']
```

With that in hand, we can then run `groupby` on the `INSTNM` (institution name) column. This means that our aggregation method (`count`, in this case) will

run once for each distinct value of INSTNM. In order to avoid getting a result for each column, I restrict our output to the CIPDESC column:

```
fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree'].groupby('INSTNM')[
    'CIPDESC'].count()
```

This returns a series in which the index contains the institution name, and the value contains the number of bachelor-level degrees offered by each institution. Finally, we can feed that into the histogram-plotting method:

```
fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree'].groupby('INSTNM')[
    'CIPDESC'].count().plot.hist()
```

The result shows us that a very large number of institutions (more than 1400!) offer fewer than 20 bachelor-level programs, fewer than 600 institutions offer between 20 and 50 programs, and 200 or fewer institutions offer more than 50 programs.

Now that we've counted the number of programs offered by each institution in this data set, we can ask which universities offer the greatest number of programs. We already have their counts, thanks to the groupby we ran before. We can thus rerun that query, sorting the resulting values in descending order, and keeping only the top 10 results:

```
fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree'].groupby('INSTNM')[
    'CIPDESC'].count().sort_values(ascending=False).head(10)
```

When I ran this, I found that the institution with the greatest number of programs was Westminster College (with 165 bachelor-level programs), followed by Pennsylvania State University's main campus (141) and the University of Washington's Seattle campus (137).

Now that we've counted bachelor's programs, how about graduate programs, offering either a master's or doctoral degree? That query is a bit trickier, because we can no longer compare CREDESC with a single string. Rather, we need to check if the value is one of two different strings. For that, I decided to use the `isin` method, which takes a list of strings and returns True if the

value in that row matches one or more of the values in the list.

To start, we can get all of the schools that offer master's and doctoral degrees:

```
fields_of_study_df.loc[fields_of_study_df['CREDESC'].isin(
    ["Master's Degree", "Doctoral Degree"])]
```

With that in hand, we can then repeat our groupby query, using count as our aggregation method:

```
fields_of_study_df.loc[fields_of_study_df['CREDESC'].isin(
    ["Master's Degree", "Doctoral Degree"])].groupby('INSTNM')[
    'CIPDESC'].count()
```

Finally, having grouped by INSTNM using count, and knowing how many programs each institution offers, we can create the histogram:

```
fields_of_study_df.loc[fields_of_study_df['CREDESC'].isin(
    ["Master's Degree", "Doctoral Degree"])].groupby('INSTNM')[
    'CIPDESC'].count().plot.hist()
```

Here, we see that the vast majority of schools offer fewer than 25 different graduate programs, with more offering fewer than 50. The number of schools offer more than 50 master's and doctoral degrees declines even more precipitously, although a handful offer more than 200.

I then asked you to find just which schools offer the greatest number of different graduate programs. Once again, this meant sorting the results from our groupby and count:

```
fields_of_study_df.loc[fields_of_study_df['CREDESC'].isin(
    ["Master's Degree", "Doctoral Degree"])
].groupby('INSTNM')['CIPDESC'].count(
).sort_values(ascending=False).head(10)
```

We see that the University of Washington's Seattle campus has the most programs (237), followed by Penn State's main campus (230), and New York University (226).

I should note that the number of programs that a university offers at any level

shouldn't be taken as an indication of how good the university is, or whether the program is appropriate for you. Especially when it comes to graduate studies, the important thing is whether the specific program is good for you, and (perhaps even more importantly) whether your advisor is someone you can trust to help you through the program. So don't take these questions as anything other than a numeric exercise; I'm certainly not trying to imply that the more programs a university offers, the better it is.

While the universities I attended all offered degree programs at all levels, some focus exclusively on either undergraduate or graduate education. I asked you to find how many universities offer bachelor's degrees, but not master's or doctorates, followed by the reverse—how many offer master's or doctoral degrees, but not bachelor's.

To answer these questions, I first wanted to find all of the schools offering bachelor's programs, and those offering master's and doctoral programs. These queries were identical to what we did before. However, here I stored them in two separate variables, so that we could then make calculations based on them:

```
ug_schools = fields_of_study_df.loc[
    fields_of_study_df['CREDESC'] ==
    'Bachelors Degree', 'INSTNM']
grad_schools = fields_of_study_df.loc[
    fields_of_study_df['CREDESC'].isin(
    ["Master's Degree", "Doctoral Degree"]), 'INSTNM']
```

Both `ug_schools` and `grad_schools` are Pandas series, with the values containing the names of the universities. However, because we retrieved the university names from `fields_of_study_df`, there will be plenty of repeats, with one row for each program offered by the institution. I decided to leave things as they are, rather than apply the `unique` method here, because `apply` returns a NumPy array, and I wanted to make use of some additional Pandas functionality.

Now that I have defined these two series, how can I determine which schools offer bachelor's degrees, but not master's or doctoral degrees? We can once again rely on `isin`. That is, to find all of the undergraduate institutions that are also graduate schools, I can say:

```
ug_schools.isin(grad_schools)
```

The above code will return a boolean series. But I want the opposite of this, namely the undergraduate schools that are **not** graduate schools, so I use `~` to flip the logic:

```
~ug_schools.isin(grad_schools)
```

This gives me the opposite boolean series of what I had before. If I apply that boolean series to `ug_schools`, I get the rows corresponding to undergraduate schools which aren't graduate schools:

```
ug_schools[~ug_schools.isin(grad_schools)]
```

However, there is a problem with this result, namely that the school names are repeated. This is where we can use the `unique` method, getting distinct values back:

```
ug_schools[~ug_schools.isin(grad_schools)].unique()
```

**drop\_duplicates** is better!

The result is a NumPy array, from which we can retrieve the `size` attribute:

```
ug_schools[~ug_schools.isin(grad_schools)].unique().size
```

The database has 923 undergraduate schools that don't offer graduate degrees. We can apply similar logic to flip the question around:

```
grad_schools[~grad_schools.isin(ug_schools)].unique().size
```

The result is 404 institutions that offer master's and doctoral degrees, but which don't offer bachelor's degrees.

Next, I thought that it would be interesting to find out how many institutions offer bachelor's degrees in computer science. Now, every institution calls its department and degree something slightly different, which means that we'll likely miss many possibilities. But if we look for programs containing the term 'Computer Science', how many will we find?

First, we'll need to find all of those rows in which CIPDESC contains the string 'Computer Science':

```
fields_of_study_df['CIPDESC'].str.contains('Computer Science')
```

But that won't be enough, because we're specifically looking for bachelor's programs in computer science. We thus need to have two conditions, joined together with &:

```
fields_of_study_df['CIPDESC'].str.contains('Computer Science') &  
    fields_of_study_df['CREDESC'] == 'Bachelors Degree'
```

This combined query will return a boolean series. We can then apply that boolean series to fields\_of\_study\_df with .loc:

```
fields_of_study_df.loc[(fields_of_study_df[  
    'CIPDESC'].str.contains('Computer Science')) & (fields_of_study_  
    'CREDESC'] == 'Bachelors Degree')]
```

The thing is, we're not interested in all of the columns. Rather, we just want to see the institution names, so that we can count them. We can do this by adding a column selector to .loc, indicating that we want to see INSTNM:

```
fields_of_study_df.loc[(fields_of_study_df[  
    'CIPDESC'].str.contains('Computer Science')) & (fields_of_study_  
    'CREDESC'] == 'Bachelors Degree'), 'INSTNM']
```

This returns a series of 824 institution names. But as before, the names aren't necessarily unique, given that there might be more than one degree program with "Computer Science" in its name. For this reason, we'll take the results, invoke unique() on them, and then get the size of the resulting array:

```
fields_of_study_df.loc[(fields_of_study_df[  
    'CIPDESC'].str.contains('Computer Science')) &  
    (fields_of_study_df['CREDESC'] ==  
    'Bachelors Degree'), 'INSTNM'].unique().size
```

The result, on my system, is 762.

The college scorecard data set puts each university into one of four categories, listed in the CONTROL column: Public, private and non-profit,

private and for-profit, or foreign. In my next question, I asked you to show how many institutions of each type offer computer science as a bachelor's-level degree.

I started with our previous query, before the call to unique:

```
fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
     'Bachelors Degree'), 'INSTNM']
```

I then decided to run a groupby on CONTROL, since we want to know how many institutions of each type offer undergraduate CS programs. In order for this to work, our column selector will need to include not just INSTNM as before, but also CONTROL:

```
fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
     'Bachelors Degree'), ['CONTROL', 'INSTNM']].groupby('
```

The above query will give us a groupby object, on which we can then invoke count:

```
fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
     'Bachelors Degree'),
    ['CONTROL', 'INSTNM']].groupby('CONTROL').count()
```

I find, with this query, that there are 32 foreign universities, 18 private for-profit universities, 501 private non-profit universities, and 273 public universities, all offering undergraduate CS programs.

Seeing this information in a table, however accurate, isn't quite as striking as a graphical display would be. I thus asked you to take these results and put them into a pie chart. Fortunately, that's quite easy. We start with the above query, and then retrieve only the INSTNM column:

```
fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
```

```

    'Bachelors Degree'),
    ['CONTROL', 'INSTNM']].groupby('CONTROL').count()['IN

```

That returns a single series of values, along with the index (i.e., the different institution categories). We can then turn that into a pie chart by invoking `.plot.pie()` to the end:

```

fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
    'Bachelors Degree'),
    ['CONTROL', 'INSTNM']].groupby(
    'CONTROL').count()['INSTNM'].plot.pie()

```

Next, I wanted to start looking at the cost of getting a computer science degree from an American university. In order to do this, I first needed to find all of the universities at which computer science is taught at the undergraduate level. This query will be identical to one we've already seen, except that I'm going to look for three different columns—OPEID6 (a unique ID number for each university in the system), CONTROL (the category of institution we've already seen) and INSTNM (the name of the institution):

```

comp_sci_universities = fields_of_study_df.loc[
    (fields_of_study_df['CIPDESC'].str.contains('Computer Science
    (fields_of_study_df['CREDESC'] == 'Bachelors Degree'),
    ['OPEID6', 'CONTROL', 'INSTNM'])

```

The good news is that we now have these rows, and have put them into a new data frame, `comp_sci_universities`. However, the index contains the same values as we had in `fields_of_study_df`. This isn't inherently bad, except that in order to answer our questions, we're going to need to join this data frame with `institutions_df`. Joining requires that the indexes match up. For that reason, I'll modify our creation of `comp_sci_universities`, such that it sets the index to be OPEID6:

```

comp_sci_universities = fields_of_study_df.loc[
    (fields_of_study_df['CIPDESC'].str.contains('Computer Science
    (fields_of_study_df['CREDESC'] == 'Bachelors Degree'),
    ['OPEID6', 'CONTROL', 'INSTNM']).set_index('OPEID6')

```

Now let's make sure that `institutions_df` has the index we'll need in order

to join them:

```
institutions_df[['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6')
```

Note that this doesn't change `institutions_df`, but rather returns a new data frame with `OPEID6` as its index.

Now that we have two data frames with a common index, we can join them together:

```
comp_sci_universities.join(institutions_df[  
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6'))
```

But the above query gives us the entire new data frame. We don't really want that; rather, we only want the `TUITIONFEE_OUT` column:

```
comp_sci_universities.join(institutions_df[  
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6'))['TUITIONFEE_OUT']
```

The result of this query, short as it is, packs a real punch: We have retrieved the tuition at each university with an undergraduate CS program in the data set.

I asked you to find the minimum, median, mean, and maximum tuitions for the tuition. We could, of course, calculate each of these individually. But when you want to perform a number of aggregate calculations, the easiest thing to do is invoke `describe`, which gives them all to you:

```
comp_sci_universities.join(institutions_df[  
    ['OPEID6', 'TUITIONFEE_OUT']].set_index(  
        'OPEID6'))['TUITIONFEE_OUT'].describe()
```

Next, I asked you to describe the tuition again, but grouped by the different types of universities (i.e., the `CONTROL` column). We can accomplish this by invoking `groupby('CONTROL')` on the result of the join, then retrieving `TUITIONFEE_OUT`, and then invoking `describe` on the result:

```
comp_sci_universities.join(institutions_df[  
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6')).groupby(  
    'CONTROL')['TUITIONFEE_OUT'].describe()
```

However, I found two problems with this result: First, foreign-owned universities gave us results of 0 or NaN for each of the columns. Second, it was a bit weird to have the university types in the index, and the results from describe in the columns. Both of these are issues of aesthetics, but if we're already playing with the data, let's see how we can clean it up.

First, I can use dropna to remove the Foreign row, the only one in which we have any NaN values:

```
comp_sci_universities.join(institutions_df[
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6')).groupby(
    'CONTROL')['TUITIONFEE_OUT'].describe().dropna()
```

What about my preference that the values of describe be in the rows, rather than the columns? We can transpose rows and columns in a Pandas data frame with the transpose method:

```
comp_sci_universities.join(institutions_df[
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6')).groupby(
    'CONTROL')['TUITIONFEE_OUT'].describe().dropna().transpose
```

However, because this is used so often, you can instead invoke it with T:

```
comp_sci_universities.join(institutions_df[
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6')
).groupby('CONTROL')['TUITIONFEE_OUT'].describe().dropna()
```



#### Note

Whereas transpose is a method, and needs to be invoked with parentheses after its name, T is a Python property, and should **not** have parentheses. Using T() will result in an error.

We often hear that the most expensive universities are also the hardest to get into. Is this true? Do we see a correlation in the data? To find out, we can invoke corr on institutions\_df, looking at the ADM\_RATE and TUITIONFEE\_OUT columns.

```
institutions_df[['ADM_RATE', 'TUITIONFEE_OUT']].corr()
```

As always, a correlation of 0 means that there's no correlation between the two values, whereas 1 means that they're perfectly aligned and -1 means that they're completely opposite. In this case, we see a correlation of -0.3, slightly negative. This means that as the tuition goes up, the admission rate goes (slightly) down—something that does indeed describe many American universities. Another way to say state this is that the universities that are hardest to get into are, in general, also momore expensive.

We can see this graphically using a scatter plot. I asked you to create such a scatter plot, with the admission rate on the y axis and the tuition fee on the x axis:

```
institutions_df.plot.scatter(x='TUITIONFEE_OUT', y='ADM_RATE')
```

We can see that the plot (overall) starts in the top left, and moves toward the bottom right. This aligns with our numeric correlation finding, that higher admission rates are associated with lower tuitions, and vice versa.

However, I was intrigued by the fact that the college scorecard includes a column MD\_EARN\_WNE\_P10, which shows the median income for graduates from each school, 10 years following graduation. This allows us to ask, and answer, a number of different questions. For example, we have now seen that more-expensive schools are also harder to get into. However, is there a tangible benefit for that additional cost? Specifically, if you attend a more exclusive school, can you expect to earn more after graduation?

I thus asked you to modify this scatter plot, colorizing it using the "Spectral" colormap, drawing upon the values in the MD\_EARN\_WNE\_P10 column:

```
institutions_df.plot.scatter(x='TUITIONFEE_OUT', y='ADM_RATE',  
                             c='MD_EARN_WNE_P10',  
                             colormap='Spectral')
```

The "Spectral" colormap put earnings of \$20,000/year in red, \$120,000/year in blue, and everything else in between. The closer to blue the dots are colored, the higher the income. It's not a huge surprise that we see a great deal of red in the top-left corner (i.e., less expensive, lower-admission schools with lower earnings), whereas yellows, greens, and blues are in the lower-right corner (i.e., more expensive, higher-admission schools with

higher earnings). On average, it would seem, graduates from more exclusive schools do earn more.

I decided to probe expensive, exclusive schools a bit further. First, I asked you to find schools that charge in the top 25% of tuition (i.e., the most expensive universities) that are also in the top 25% of schools offering Pell grants. Pell grants are awarded to students on the basis of financial need, and provide us with a rough estimate of how many less-wealthy students are studying somewhere. I'm thus looking to find, in simple terms, expensive schools that have a relatively high proportion of non-wealthy students.

In order to do that, we'll need to use `quantile(0.75)` on the `TUITIONFEE_OUT` column, to find what the top quartile of tuition is. We'll similarly need to run `quantile(0.75)` on the `FTFTPCTPELL` column, which contains the percentage of Pell-grant recipients at each school. We can then compare each institution's value for `TUITIONFEE_OUT` and `FTFTPCTPELL` against that 0.75 quantile, retrieving institutions that are above those thresholds in both:

```
institutions_df.loc[(institutions_df['TUITIONFEE_OUT'] >
                    institutions_df['TUITIONFEE_OUT'].quantile(0.75)) &
                  (institutions_df['FTFTPCTPELL'] >
                    institutions_df['FTFTPCTPELL'].quantile(0.75))]
```

This returns all of the rows in `institutions_df` where both `TUITIONFEE_OUT` and `FTFTPCTPELL` are above the 75th percentile. But we aren't really interested in all of the columns; I asked you to show the institution name, along with its city and state. For that, we'll need to include a column selector in our call to `.loc`:

```
institutions_df.loc[(institutions_df['TUITIONFEE_OUT'] >
                    institutions_df['TUITIONFEE_OUT'].quantile(0.75)) &
                  (institutions_df['FTFTPCTPELL'] >
                    institutions_df['FTFTPCTPELL'].quantile(0.75)),
                  ['INSTNM', 'CITY', 'STABBR']]
```

Finally, I asked you to sort the output by institution name, which we can do by calling `sort_values` and specifying the `INSTNM` column:

```
institutions_df.loc[(institutions_df['TUITIONFEE_OUT'] >
                    institutions_df['TUITIONFEE_OUT'].quantile(0.75)) &
                  (institutions_df['FTFTPCTPELL'] >
```

```
institutions_df['FTFTPCTPELL'].quantile(0.75),  
['INSTNM', 'CITY', 'STABBR']].sort_values(by='INSTNM')
```

Now let's look at university tuition from another perspective: The college scorecard tracks the net price for four-year public and private institutions (NPT4\_PUB and NPT4\_PRIV, respectively). It then breaks the tuition payments down even further in additional columns, showing (for example) the average price paid by people in the lowest income bracket at public (NPT41\_PUB) and private (NPT41\_PRIV) universities.

At how many institutions, both public and private, does the average lowest-income-bracket student receive money, rather than spend money?

If we were merely interested in public institutions, we could find all of those where the value of NPT41\_PUB is less than 0, and then show their names:

```
institutions_df.loc[institutions_df['NPT41_PUB'] < 0,  
                    'INSTNM'].count()
```

Or if we're interested in private institutions, we would have to do the same for NPT41\_PRIV:

```
institutions_df.loc[institutions_df['NPT41_PRIV'] < 0,  
                    'INSTNM'].count()
```

We could use | for an "or" condition, thus getting the values where either of these is less than 0.

```
institutions_df.loc[((institutions_df['NPT41_PUB'] < 0) |  
                    (institutions_df['NPT41_PRIV'] < 0)),  
                    'INSTNM'].count()
```

This gave me an answer of 12. However, there's another way to do this: We can add the values in NPT41\_PRIV to those in NPT41\_PUB, with a fill\_value of 0. Then we can simply check to see where NPT41\_PUB is < 0:

```
institutions_df.loc[institutions_df['NPT41_PUB'].add(  
                    institutions_df['NPT41_PRIV'], fill_value=0) < 0,  
                    'INSTNM'].count()
```

This gives the same answer, and while I'm not convinced it's a better way to

solve the problem, it shows that in Pandas, there's always more than one option, and they often look quite different from one another.

I then asked you to show, for public universities, the average proportion that the bottom quintile pays vs. the top quintile. To calculate this, we'll divide NPT41\_PUB (the bottom quintile) into NPT45\_PUB (the top quintile), and then take the mean:

```
(institutions_df['NPT41_PUB'] / institutions_df['NPT45_PUB']).mean
```

I get a result of about 52 percent. We can then repeat this for private universities:

```
(institutions_df['NPT41_PRIV'] / institutions_df['NPT45_PRIV']).mean
```

It turns out that people in the bottom quintile at private universities pay about 71 percent of what the top quintile do. So not only is tuition higher at private universities, but the poorest students also end up paying the greatest proportion of their tuition!

In looking at this data, we've seen that, overall, the schools with the highest-paid alumni are also the most expensive and the hardest to get into. But of course, that's only overall, in the aggregate. I thus asked you to find the schools that offer the greatest return on investment—whose tuitions are in the lowest 25%, but whose 10-year alumni are in the highest 25% of salaries.

First, let's look at public institutions:

```
institutions_df.loc[(institutions_df['NPT4_PUB'] <=
    institutions_df['NPT4_PUB'].quantile(0.25)) &
    (institutions_df['MD_EARN_WNE_P10'] >=
    institutions_df['MD_EARN_WNE_P10'].quantile(0.75)),
    ['INSTNM', 'STABBR', 'CITY']].sort_values(by=['STABBR', 'CITY'])
```

The above query is a variation on what we've already done, looking for those public universities whose tuition is in the lowest quartile, but whose 10-year alumni are earning in the highest quartile. In our column selector, we ask for only three columns, namely institution name, state, and city. That allows us to then sort the results, first by state and then by city. The result is a data frame with 22 rows, whose universities are in California, Florida, New York,

and (in one case) New Mexico.

What about private universities? We can run a similar query, but using NPT4\_PRIV rather than NPT4\_PUB:

```
institutions_df.loc[(institutions_df['NPT4_PRIV'] <=
    institutions_df['NPT4_PRIV'].quantile(0.25)) &
    (institutions_df['MD_EARN_WNE_P10'] >=
    institutions_df['MD_EARN_WNE_P10'].quantile(0.75)),
    ['INSTNM', 'STABBR', 'CITY']].sort_values(by=['STABBR', 'CITY
```

This query returned 30 universities, spread out across a variety of states. Some well-known universities (e.g., Harvard, Stanford, and Princeton) were in there, along with some smaller and lesser-known ones.

Next, I wanted to know if we could find any correlation between admission rates and completion rates. That is, if a school is highly selective, then are its students more likely to graduate? I ran the following query:

```
institutions_df[['C100_4', 'ADM_RATE']].corr()
```

Sure enough, we see a moderate negative correlation. That is: A school which accepts more people has a lower graduation rate. That shouldn't hugely surprise us; after all, for a school to accept more people, it likely has to take people who are bigger risks in terms of not finishing.

Next, I asked whether, on average, people earn more after graduating from a public or private university. That is, on average, how much do people earn for each value of the CONTROL column? Once again, we find ourselves joining institutions\_df with fields\_of\_study\_df—but only after doing a groupby on fields\_of\_study\_df:

```
institutions_df[['OPEID6', 'MD_EARN_WNE_P10']].set_index(
    'OPEID6').join(
    fields_of_study_df.groupby('OPEID6')['CONTROL'].min()
    ).groupby('CONTROL').mean()
```

The result aligned with my expectations, namely that people who attend private-for-profit universities end up earning less than those who attend public universities, who in turn end up earning less than those who attend

private universities. Obviously, this is an aggregate measure—and I definitely know high earners who attended public universities, and low earners who attended private ones. But data analytics is all about making generalizations, drawing conclusions that are incorrect for any individual, but correct for the overall population.

Let's take this question of private universities to an extreme: People often want to get into the best-known universities, on the assumption that they'll be able to earn more later on in life. Is this true? Does going to a famous, exclusive university mean you'll have a more lucrative career? I asked you to check the mean salary for graduates from what are sometimes known as "Ivy Plus" schools—the Ivy League, as well as MIT, Stanford, and the University of Chicago.

To do this, I used `isin` in my column selector. Note that the universities' formal names were a bit tricky to figure out, especially for "Columbia University in the City of New York." But here's the query that I ended up writing:

```
institutions_df.loc[institutions_df[
    'INSTNM'].isin(
    ['Harvard University',
     'Massachusetts Institute of Technology',
     'Yale University',
     'Columbia University in the City of New York',
     'Brown University',
     'Stanford University',
     'University of Chicago',
     'Dartmouth College',
     'University of Pennsylvania',
     'Cornell University',
     'Princeton University']),
    'MD_EARN_WNE_P10'].mean()
```

The answer to this query was just over \$91,806/year, more than twice the average salary earned by all graduates of private universities. Which was, as we saw, greater still than the amount earned by graduates of public or for-profit institutions.

Finally, I wanted to compare post-graduation salaries, 10 years out, by state. That is: Will your future earnings depend, in part, on the state in which you

studied? For starters, I performed a groupby on the states (STABBR), looking at the mean salary of 10-year graduates:

```
institutions_df.groupby('STABBR')['MD_EARN_WNE_P10'].mean()
```

This gave me the overall answer I wanted, but understanding such data is always easier when the data is sorted. I thus asked you to sort the values, in descending order:

```
institutions_df.groupby('STABBR')[  
    'MD_EARN_WNE_P10'].mean().sort_values(ascending=False)
```

Following this, I asked you to create a bar plot from the per-state salary averages:

```
institutions_df.groupby('STABBR')[  
    'MD_EARN_WNE_P10'].mean().sort_values().plot.bar(figsize=(20,10)
```

By sorting the values, we got (I believe) a more aesthetically pleasing, easy to read, plot than would otherwise have been the case. We can easily see that there is a big difference between how much people earn after graduating from schools in Massachusetts and Rhode Island, as opposed to Arkansas and Mississippi. However, before we make a claim regarding the quality of universities in these respective states, we have to find out how many people still live in the states where they studied. After all, the cost of living in New England is significantly higher than Arkansas and Mississippi, so it stands to reason that people living there will earn more—regardless of what university they attended.

Finally, I asked you to create a box plot based on the per-state salary data, so that we can easily see the spread in visual form:

```
institutions_df.groupby('STABBR')['MD_EARN_WNE_P10'].mean().plot.
```

The plot shows us that most annual salaries are between \$25,000 and \$50,000, with the median being just under \$40,000.

## 12.1.2 Solution

```
institutions_filename = '../data/Most-Recent-Cohorts-Institution.
```

```

institutions_df = pd.read_csv(institutions_filename,
                             usecols=['OPEID6',
                                       'INSTNM', 'CITY', 'STABBR',
                                       'FTFTPCTPELL', 'TUITIONFEE_IN',
                                       'TUITIONFEE_OUT', 'ADM_RATE',
                                       'NPT4_PUB', 'NPT4_PRIV',
                                       'NPT41_PUB', 'NPT41_PRIV',
                                       'NPT45_PUB', 'NPT45_PRIV',
                                       'MD_EARN_WNE_P10', 'C100_4'])

fields_filename = '../data/FieldOfStudyData1718_1819_PP.csv.gz'
fields_of_study_df = pd.read_csv(fields_filename,
                                  usecols=['OPEID6', 'INSTNM',
                                            'CREDESC', 'CIPDESC', 'CONTROL'])

institutions_df.groupby('STABBR')['OPEID6'].count().sort_values(
    ascending=False).head(1)

institutions_df.groupby(['STABBR', 'CITY'])['OPEID6'].count().sort_values(
    ascending=False).head(1)

pre_category_memory = institutions_df.memory_usage(deep=True).sum
print(f'{pre_category_memory:,}')

institutions_df['CITY'] = institutions_df[
    'CITY'].astype('category')
institutions_df['STABBR'] = institutions_df[
    'STABBR'].astype('category')

post_category_memory = institutions_df.memory_usage(
    deep=True).sum()

savings = pre_category_memory - post_category_memory
print(f'{savings:,}')

```

Next, we look at institutions and the degrees they offer:

```

fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree'].groupby('INSTNM')['CIPDESC'].count().plot.hist()

fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree'].groupby('INSTNM')['CIPDESC'].count().sort_values(ascending=False).head(10)

```

```

fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'].isin(["Master's Degree", "Doctoral Degree"])]>.gro
    'INSTNM')['CIPDESC'].count().plot.hist()

fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'].isin(["Master's Degree", "Doctoral Degree"])]>.gro
    'INSTNM')['CIPDESC'].count().sort_values(ascending=False).h

ug_schools = fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'] == 'Bachelors Degree', 'INSTNM']
grad_schools = fields_of_study_df.loc[fields_of_study_df[
    'CREDESC'].isin(["Master's Degree", "Doctoral Degree"]), 'IN

ug_schools[~ug_schools.isin(grad_schools)].unique().size

grad_schools[~grad_schools.isin(ug_schools)].unique().size

fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) & (fields_of
    'CREDESC'] == 'Bachelors Degree'), 'INSTNM'].unique().

fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
        'Bachelors Degree'), ['CONTROL',
        'INSTNM']].groupby('CONTROL').count()

fields_of_study_df.loc[(fields_of_study_df[
    'CIPDESC'].str.contains('Computer Science')) &
    (fields_of_study_df['CREDESC'] ==
        'Bachelors Degree'), ['CONTROL', 'INSTNM']
    ].groupby('CONTROL').count()['INSTNM'].plot.pie()

comp_sci_universities = fields_of_study_df.loc[
    (fields_of_study_df['CIPDESC'].str.contains('Computer Science
    (fields_of_study_df['CREDESC'] == 'Bachelors Degree'),
    ['OPEID6', 'CONTROL', 'INSTNM']).set_index('OPEID6')

comp_sci_universities.join(
    institutions_df[['OPEID6', 'TUITIONFEE_OUT']
    ].set_index('OPEID6'))['TUITIONFEE_OUT'].describe()

comp_sci_universities = fields_of_study_df.loc[
    (fields_of_study_df['CIPDESC'].str.contains('Computer Science
    (fields_of_study_df['CREDESC'] == 'Bachelors Degree'),
    ['OPEID6', 'CONTROL', 'INSTNM']).set_index('OPEID6')

```

```
comp_sci_universities.join(institutions_df[
    ['OPEID6', 'TUITIONFEE_OUT']].set_index('OPEID6')
).groupby('CONTROL')['TUITIONFEE_OUT'].describe().dropna().T
```

Next, let's look at admission rates, tuition, financial aid, and after-graduation earnings:

```
institutions_df[['ADM_RATE', 'TUITIONFEE_OUT']].corr()
```

```
institutions_df.plot.scatter(x='TUITIONFEE_OUT', y='ADM_RATE',
                             c='MD_EARN_WNE_P10',
                             colormap='Spectral')
```

```
institutions_df.loc[(institutions_df['TUITIONFEE_OUT'] >
    institutions_df['TUITIONFEE_OUT'].quantile(0.75)) &
    (institutions_df['FTFTPCTPELL'] >
    institutions_df['FTFTPCTPELL'].quantile(0.75)),
    ['INSTNM', 'CITY', 'STABBR']].sort_values(by='INSTNM')
```

```
institutions_df.loc[institutions_df['NPT41_PUB'].add(
    institutions_df['NPT41_PRIV'], fill_value=0) < 0,
    'INSTNM'].count()
```

```
(institutions_df['NPT41_PUB'] / institutions_df['NPT45_PUB']).mea
```

```
(institutions_df['NPT41_PRIV'] / institutions_df['NPT45_PRIV']).m
```

```
institutions_df.loc[(institutions_df['NPT4_PUB'] <=
    institutions_df['NPT4_PUB'].quantile(0.25)) &
    (institutions_df['MD_EARN_WNE_P10'] >=
    institutions_df['MD_EARN_WNE_P10'].quantile(0.75)),
    ['INSTNM', 'STABBR', 'CITY']].sort_values(by=['STABBR', 'CITY
```

```
institutions_df.loc[(institutions_df['NPT4_PRIV'] <=
    institutions_df['NPT4_PRIV'].quantile(0.25)) &
    (institutions_df['MD_EARN_WNE_P10'] >= institutions_df[
    'MD_EARN_WNE_P10'].quantile(0.75)),
    ['INSTNM', 'STABBR', 'CITY']].sort_values(by=['STABBR', 'CITY
```

```
institutions_df[['C100_4', 'ADM_RATE']].corr()
```

```
institutions_df[['OPEID6',
    'MD_EARN_WNE_P10']].set_index('OPEID6').join(
    fields_of_study_df.groupby('OPEID6')['CONTROL'].min()
).groupby('CONTROL').mean()
```

```
institutions_df.loc[institutions_df['INSTNM'].isin(
```

```

['Harvard University',
 'Massachusetts Institute of Technology',
 'Yale University',
 'Columbia University in the City of New York',
 'Brown University',
 'Stanford University',
 'University of Chicago',
 'Dartmouth College',
 'University of Pennsylvania',
 'Cornell University',
 'Princeton University']], 'MD_EARN_WNE_P10'].mean()

institutions_df.groupby('STABBR')['MD_EARN_WNE_P10']
    .mean().sort_values(ascending=False)

institutions_df.groupby('STABBR')['MD_EARN_WNE_P10']
    .mean().sort_values().plot.bar(figsize=(20,10))

institutions_df.groupby('STABBR')['MD_EARN_WNE_P10']
    .mean().plot.box()

```

## 12.2 Summary

You've now come to the true and actual end of the book.

Thanks for joining me on this journey. I hope that the exercises in this book, including all of the extra "beyond the exercise" questions, helped you to improve your understanding of Pandas and how to load, clean, and analyze data in a variety of ways.

Beyond the specific techniques that I covered in this book, I hope that you also started to internalize the Pandas perspective on data analysis. Pandas is a huge (and constantly growing) library, and there's no way for someone to know all of it. Understanding how Pandas works means that when you're faced with a new problem, you can guess how to solve it, even predicting what methods Pandas will provide in order to do so.

I wish you the best of success in your use of Pandas to analyze data, in whatever you're doing. And I hope that this book helped you to advance your skills in that area.