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 want to share our analysis by saving data to files in those (or other) formats.

In this chapter, we explore how to import data from various formats, emphasizing CSV files because they're so common. We look at ways in which we can not only read from such files but also customize the reading either to improve the quality of our data or optimize the process.

Table 3.1 What you need to know

Concept	What is it?	Example	To learn more
`pd.read_csv`	Returns a new data frame based on CSV input	`df = pd.read_csv('myfile.csv')`	http://mng.bz/wvl7
`df.to_csv`	Writes a data frame to a CSV-formatted file or string	`df.to_csv('myfile.csv')`	http://mng.bz/7Dzx
`pd.read_json`	Returns a new data frame based on JSON input	`df = pd.read_json('myfile.json')`	http://mng.bz/mV4n
`df.corr`	Shows the correlations among the columns	`df.corr()`	http://mng.bz/6DQG
`df.dropna`	Returns a new data frame without any `NaN` values	`df.dropna()`	http://mng.bz/o1PN
`df.loc`	Retrieves selected rows and columns	`df.loc[df['trip_distance'] > 10, 'passenger_count']`	http://mng.bz/nWPv

`pd.read_html`	Returns a list of data frames based on HTML input	`df = df.read_html('https://a-site.com')`	http://mng.bz/vnxx
`s.value_counts`	Returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	http://mng.bz/yQyJ
`s.round`	Returns a new series based on `s` in which the values are rounded to the specified number of decimals.	`s.round(2)`	http://mng.bz/QPym

`df.memory_usage`	Indicates how many bytes are being used by a data frame and its associated data	`df.memory_usage()`	http://mng.bz/XNPY
`pd.Series.idxmin`	Returns the index of the lowest value in a series	`s.idxmin()`	http://mng.bz/ZR6Z
`pd.Series.idxmax`	Returns the index of the highest value in a series	`s.idxmax()`	http://mng.bz/RmrP
`pd.DataFrame.agg`	Invokes one or more aggregation methods on a data frame	`df.idxmax(['min', 'max'])`	http://mng.bz/27QX

CSV, the nonstandard standard

Computer scientist Andrew S. Tanenbaum once said, "The good thing about standards is that there are so many to choose from." In many ways, the same can be said for files in comma-separated values (CSV) format, which are the overwhelming favorite in the world of data. Sure, plenty of people use Excel and relational databases, but if you download a data set from the internet, odds are it's 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, meaning you can read (and edit) them without special tools.

For all its popularity, CSV doesn't have a formal specification. There is a Request for Comments (RFC; 4110, available at **https://datatracker.ietf.org/doc/html/rfc4180**), but it's informational, from 2005. And although we can generally agree on what constitutes legal CSV, many variants and gray areas 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`), we can choose from many, *many* parameters, each of which can affect how it is written. Among the most common are these:

- `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
- `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, we can say

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

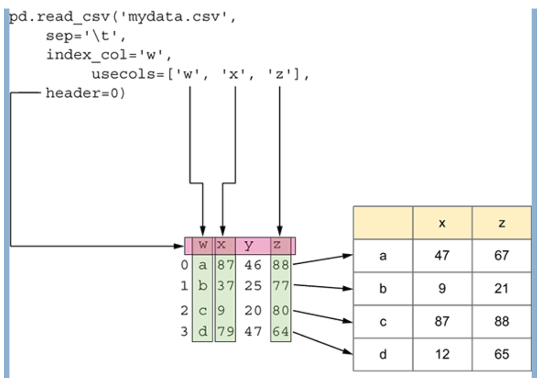
① Names the CSV file to read

② Uses a tab as a separator character

③ Sets the index to the w column

④ Only reads columns w, x, and z from the file

⑤ The file's first line contains the column names.



```
pd.read_csv('mydata.csv',
    sep='\t',
    index_col='w',
        usecols=['w', 'x', 'z'],
    header=0)
```

	x	z
a	47	67
b	9	21
c	87	88
d	12	65

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 several of these parameters in this book, but many that we don’t cover may 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 know as much about your data as possible before willy-nilly reading it into memory. You probably don’t want to load all the columns into pandas. And you may want to specify the type of data 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.

xercise 15 • Weird taxi rides

When I was growing up, taking a taxi in New York City was pretty simple: you hailed a cab and told the driver where you wanted to go. When you got there, you paid whatever was on the meter, added a tip, and got a receipt. Of course, the payment was in cash.

Nowadays, things are a bit different: New York taxis 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. The screens are also computers, storing information about the trip and sending it to the Taxi and Limousine Commission (TLC), 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 use it quite a bit in this book. In particular, we'll look 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 tip.
- `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 exercise, I want you to create a data frame from the CSV data for January 2019:

1. Load the CSV file into a data frame using only the four columns mentioned earlier: `passenger_count`, `trip_distance`, `payment_type`, and `total_amount`.
2. How many taxi rides had more than eight passengers?
3. How many taxi rides had zero passengers?
4. How many taxi rides were paid for in cash and cost over $1,000?
5. How many rides cost less than $0?
6. 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.

Working it out

To solve this problem, we first need to create a new data frame from the CSV file. Fortunately, the data is formatted in such a way that `pd.read_csv` works 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 reduce the memory footprint a lot. (I found that loading only the columns I asked for reduced 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, which can either contain integers (indicating the numeric index of each column) or strings representing the column names. I generally prefer to use strings because 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, we can start answering the questions asked by this exercise.

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

We can even run the `count` method on every column in the data frame:

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

Counting (and ignoring) NaN

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, we get back a series in which the index represents the data frame's columns and the numbers indicate how many non-`NaN` values are in each column. For example:

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

The result of this code is 4. However, the result of the following code is a series:

```
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 series shows the number of non-`NaN` values 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`:

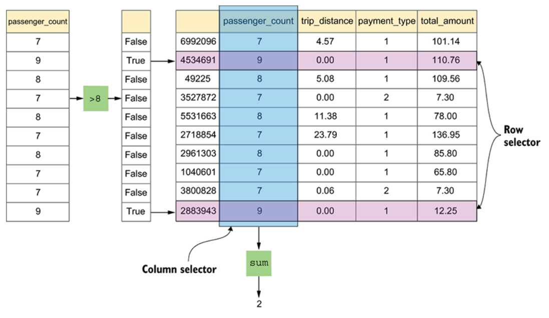
```
df.loc[df['passenger_count'] > 8,     ①
    'passenger_count'                 ②
    ].count()                         ③
```

① Row selector: only rows with more than nine passengers

② Column selector: only the passenger_count column

③ How many non-NaN values are there?

Sure enough, this tells us that in January 2019, there were nine trips with more than eight people, as we can see in figure 3.1. (I hope they took place in larger-than-usual taxis.)



passenger_count			passenger_count	trip_distance	payment_type	total_amount
7	False	6992096	7	4.57	1	101.14
9	True	4534691	9	0.00	1	110.76
8	False	49225	8	5.08	1	109.56
7	False	3527872	7	0.00	2	7.30
8	False	5531663	8	11.38	1	78.00
7	False	2718854	7	23.79	1	136.95
8	False	2961303	8	0.00	1	85.80
7	False	1040601	7	0.00	1	65.80
7	False	3800828	7	0.06	2	7.30
9	True	2883943	9	0.00	1	12.25

Figure 3.1 Graphical depiction of selecting rows where `passenger_count` > 8 and then invoking `sum`.

Next, how many taxi rides in January 2019 had zero passengers? I would guess that if there aren't any passengers, 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
```

① Row selector, looking for zero passengers

② Column selector: just passenger_count

③ How many non-NaN values are there?

There were 117,381 such rides. This sounds like a lot, but it turns out to be only 1.5% of all rides taken that month.

Although it’s true that most people pay for taxi rides using credit cards, some still pay cash for various reasons. How many rides that month were paid for in cash and had a `total_amount` of more than $1,000?

This question is a bit harder to answer because we 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 will find `total_amount` greater than 1,000. We can then join the two together using `&`:

```
(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 apply it to the data frame using `loc`, retrieving the `total_amount` column via the second argument and then calling `count` on it (figure 3.2):

```
df.loc[(df['payment_type'] == 2) & (df['total_amount'] > 1000),    ①
        'passenger_count'                                          ②
      ].count()                                                    ③
```

① Row selector: looking for cash payments of at least $1,00

② Column selector: looking for passenger count

③ How many non-NaN values are there?

I got a result of 5. I may be extreme in using very little cash, but I was still shocked to discover that there were *any* rides paid for in cash with 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.

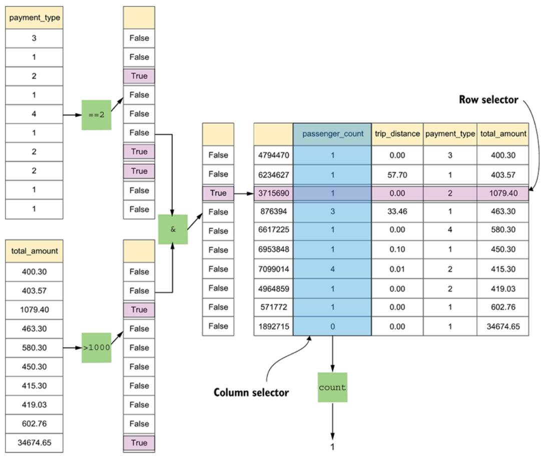


Figure 3.2 Graphical depiction of selecting rows where `payment_type` == 2 and `total_amount` > 1000, and counting the elements of `passenger_count`

Next, I asked you to find rides that cost less than $0. This would presumably mean the rider got a refund, but there could be other reasons, such as overpayment for a previous ride. How many such rides took place in January 2019?

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

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

We 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 only .01% of all taxi rides gave money back. These 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 solve this, we once again need to find all the trips that traveled a below-average distance:

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

Then we find all the trips that cost an above-average amount:

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

We 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 use `loc` on this boolean series, applying it to `trip_distance` and then counting the results (figure 3.3):

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

① First part of row selector: trip_distance is less than the mean

② Second part of row selector, total_amount is more than the mean

③ column selector, trip_distance column

④ Count the non-NaN value

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

Solution

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

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

You can explore this in the Pandas Tutor at **http://mng.bz/0lvN**.

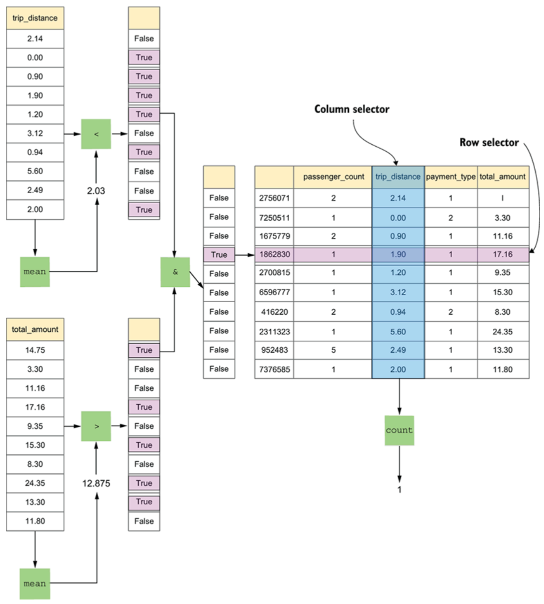


Figure 3.3 Graphical depiction of counting rows where `trip_distance` is less than the mean but `total_amount` is greater than the mean

Beyond the exercise

- Repeat this exercise using the `query` method rather than a boolean index and `loc`.
- How many rides that cost less than $0 involved either a dispute (`payment_type` of 4) or a voided trip (`payment_type` of 6)?
- I stated earlier that most people pay for taxi rides using a credit card. Show this, and find what percentage normally pays in cash versus a credit card.

xercise 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 look at how we can load data from multiple files into a single data frame and then make 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 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 one 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 date-time parsing that would make it easier to solve these problems. We'll discuss those techniques in chapters 6, 7, and 10, respectively. For now, see if you can answer the questions without such assistance.

Working it out

There are countless ways to measure the pandemic's effect on our lives and our world. I find that this data set provides some interesting insights.

For starters, I wanted you to take information from two different files and join them into a single data frame. We saw in Chapter 1 how to use `pd.concat` to combine two existing series objects into a single series. It turns out you can also use `pd.concat` on data frames, which is what we want to do here. We can 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 is to add a new column to each of the file-based data frames and then concatenate them (figure 3.4):

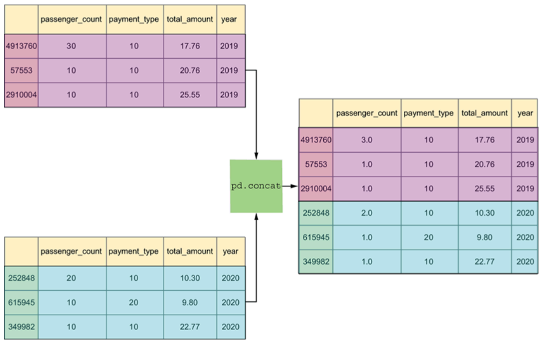
```
df_2019_jul = pd.read_csv('../data/nyc_taxi_2019-07.csv',
                usecols=['passenger_count',
                         'total_amount', 'payment_type'])
df_2019_jul['year'] = 2019                                ①

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
```

① Adds a year column with a value of 2019 to all rows from 2019

② Adds a year column with a value of 2020 to all rows from 2020

③ Creates df, combining both data frames



	passenger_count	payment_type	total_amount	year
4913760	30	10	17.76	2019
57553	10	10	20.76	2019
2910004	10	10	25.55	2019

	passenger_count	payment_type	total_amount	year
252848	20	10	10.30	2020
615945	10	20	9.80	2020
349982	10	10	22.77	2020

	passenger_count	payment_type	total_amount	year
4913760	3.0	10	17.76	2019
57553	1.0	10	20.76	2019
2910004	1.0	10	25.55	2019
252848	2.0	10	10.30	2020
615945	1.0	20	9.80	2020
349982	1.0	10	22.77	2020

Figure 3.4 Concatenating two data frames into a single one

Once we have done that, we have a single data frame, `df`, to ask questions. For starters, we want to know how many rides were taken in 2019 versus 2020. That can be done by invoking `count` on any of our columns, subtracting the 2020 count from the 2019 count (figure 3.5):

```
(
    df.loc[df['year'] == 2019, 'total_amount'].count() -    ①
    df.loc[df['year'] == 2020, 'total_amount'].count()      ②
)
```

① Number of rides in 2019

② Number of rides in 2020

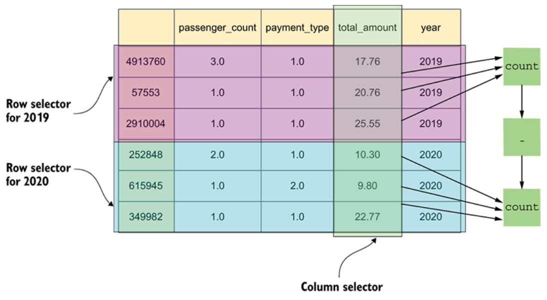


Figure 3.5 Comparing the number of rides in 2019 with 2020

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

```
(
    df.loc[df['year'] == 2019, 'total_amount'].sum() -     ①
    df.loc[df['year'] == 2020, 'total_amount'].sum()       ②
)
```

① Total earned in 2019

② Total earned in 2020

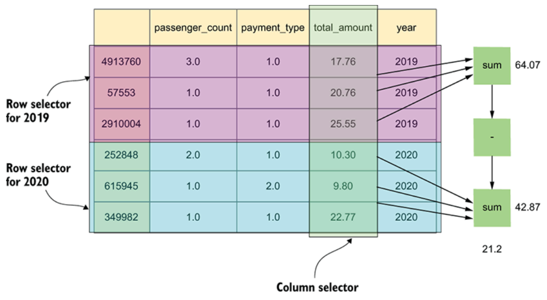


Figure 3.6 Comparing the total amount earned in 2019 with 2020

The answer that I get is 108848979.24000001, or more than $108 million. I don't know about you, but I look at that huge number and am simply astonished. (You can see a graphical depiction of this in figure 3.6.)

Rounding floats

If you’re bothered by the long number of numbers after the decimal point, you can use the `round` method on a series to limit it 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 may 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 large degree. As a result, we may wonder whether people were less likely to take taxis with other people. The next question asked you to compare the proportion (not a raw number) of multiperson taxi rides in 2019 with those in 2020. To do that, we can divide the number of multiperson rides by the number of overall rides. Here’s how we do that:

```
df.loc[
    (df['year'] == 2019) &
    (df['passenger_count'] > 1), 'passenger_count'].count() /     ①
        df.loc[df['year'] == 2019, 'payment_type'].count()        ②

df.loc[
    (df['year'] == 2020) &
    (df['passenger_count'] > 1), 'passenger_count'].count() /    ③
        df.loc[df['year'] == 2020, 'payment_type'].count()       ④
```

① Number of rides in 2019 with > 1 passenger

② Total number of rides in 2019

③ Number of rides in 2020 with > 1 passenger

④ Total number of rides in 2020

I get about 28% in 2019 and 21% in 2020, meaning 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, we want to know whether 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 (figure 3.7):

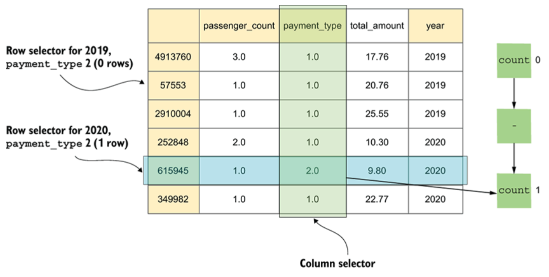
```
df.loc[
    (df['year'] == 2019) &
    (df['payment_type'] == 2), 'payment_type'].count() /      ①
       df.loc[df['year'] == 2019, 'payment_type'].count()     ②

df.loc[
    (df['year'] == 2020) &
    (df['payment_type'] == 2), 'payment_type'].count() /      ③
       df.loc[df['year'] == 2020, 'payment_type'].count()     ④
```

① Number of rides paid for with cash in 2019

② Total number of rides in 2019

③ Number of rides paid for with cash in 2020

④ Total number of rides in 2020


Row selector for 2019, payment_type 2 (0 rows)
Row selector for 2020, payment_type 2 (1 row)
count 0
-
count 1
Column selector

	passenger_count	payment_type	total_amount	year
4913760	3.0	1.0	17.76	2019
57553	1.0	1.0	20.76	2019
2910004	1.0	1.0	25.55	2019
252848	2.0	1.0	10.30	2020
615945	1.0	2.0	9.80	2020
349982	1.0	1.0	22.77	2020

Figure 3.7 Comparing the number of cash payments in 2019 with 2020

Here, the answer is a bit surprising. In July 2019, about 29% of the trips were paid for in cash. But in July 2020, that number went up to 32%—exactly the opposite direction of what I expected, given that many people preferred contactless payment. 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 the reason, the numbers bear out the increased use of cash.

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 http://mng.bz/g7jE.

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 two 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 versus 2020.

Data frames and dtype

In Chapter 1, we saw that every series has as `dtype` describing the type of data 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 retrieving the `dtypes` (notice the plural!) attributes from a data frame, we can determine the `dtype` of each column. This information and additional details about the data frame are also available by invoking the `info` method on our data frame.

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

- If the values can all be turned into integers, it chooses `int64`.

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

However, there are several problems with letting pandas analyze and choose the data this way. First, although 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, it must examine all the values in that column. But if a column has millions of rows, 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 it's happening; pandas does this to save memory.

This can potentially lead to a problem, if pandas finds (for example) values that look like integers at the top of the file and values that look like strings at the bottom. In such a case, you end up with a `dtype` of `object` and with values of different types. This is almost certainly bad, and pandas warns you about it with a

`DtypeWarning`. If you load the New York City taxi data from January 2020 into pandas without specifying `usecols`, you may 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 okay to examine all 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 I've described here. But remember that setting `low_memory` to `False` may use lots of memory, a potentially big problem if your data set 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`, pandas will simply use `np.int64` or `np.float64`. And if you specify `str`, 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 an integer `dtype`. But remember that if the column contains `NaN`, 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.

xercise 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 the data is in the most appropriate and compact form it can be and will use as little memory as possible when being loaded. So, I want you to do the following:

- 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.

Working it out

Although this exercise is ostensibly about setting the `dtype` when reading from files, there is much more to it—in particular, we begin to see that cleaning data and setting appropriate data types can be a multistep process.

We start by reading the data from January 2020, much as we did before, with `read_csv`. However, this time I want you to specify the `dtype` of each column. In theory, the best choices for the `dtype` assignments are `int8` for both `passenger_count` and `payment_type`, because both are integers that won't ever go above 128.

But if we try to set the `dtype` for `passenger_count` and `payment_type` to `int8`, we quickly discover a problem: pandas raises an error, indicating that there are `NaN` values in those columns. Because `NaN` is a float that cannot be converted into an integer, we need to keep those columns as floats. So, we can use `float32` for now and then switch it back to `int8` when we're done removing `NaN` values.

It may seem odd 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?

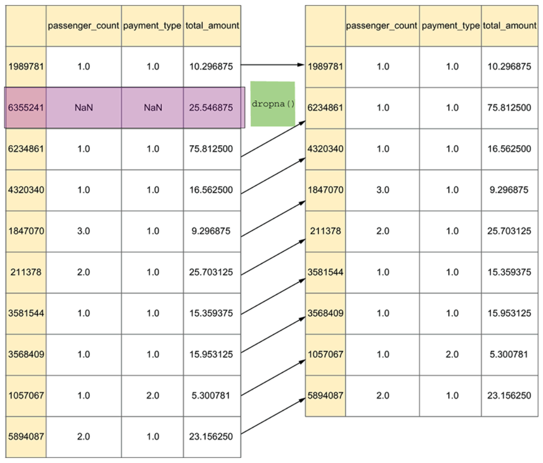
Because in a large data set, we risk having multiple `dtype` values for a single column. That's a result of pandas reading our file in chunks and choosing a `dtype` for each chunk. If all chunks have the same `dtype`, the entire column matches. If not, the column is set to a `dtype` of `object`, meaning 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 12.

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%. It doesn't seem unreasonable for drivers to neglect to indicate the number of passengers in 1 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` (figure 3.8):

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



	passenger_count	payment_type	total_amount
1989781	1.0	1.0	10.296875
6355241	NaN	NaN	25.546875
6234861	1.0	1.0	75.812500
4320340	1.0	1.0	16.562500
1847070	3.0	1.0	9.296875
211378	2.0	1.0	25.703125
3581544	1.0	1.0	15.359375
3568409	1.0	1.0	15.953125
1057067	1.0	2.0	5.300781
5894087	2.0	1.0	23.156250

	passenger_count	payment_type	total_amount
1989781	1.0	1.0	10.296875
6234861	1.0	1.0	75.812500
4320340	1.0	1.0	16.562500
1847070	3.0	1.0	9.296875
211378	2.0	1.0	25.703125
3581544	1.0	1.0	15.359375
3568409	1.0	1.0	15.953125
1057067	1.0	2.0	5.300781
5894087	2.0	1.0	23.156250

Figure 3.8 Removing rows containing `NaN` with `dropna`

Even though `df.dropna()` returns a new data frame, its data may be shared with other data frames for the sake of efficiency. Modifying our data frame may 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`, you may get the warning, which may be harmless, but it also may mean any changes you make won't stick.

Now that we have removed all the `NaN` values, we can finally assign the `dtype` values we wanted to use all along:

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

=== Solution
```

```
df = pd.read_csv('../data/nyc_taxi_2020-01.csv',
                usecols=['passenger_count',
                        'total_amount' , 'payment_type'],
                dtype={'passenger_count':float32,
                       'total_amount':float32,
                       'payment_type':float32})                ①

df.count() 2((CO11-2))                                         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)
```

① We use float32 for all columns because two of them contain NaN values

② Uses df.count to determine which columns may contain NaN

③ Removes all rows containing even one NaN, copies into a new data frame, and assigns back to df

④ Uses the loc assignment with : to indicate "all rows"

You can explore this in the Pandas Tutor at **http://mng.bz/eEKv**.

Beyond the exercise

- Create a data frame from four other columns (`VendorID`, `trip_distance`, `tip_amount`, and `total_amount`), specifying the `dtype` for each. What types are most appropriate? Can you use them directly, or must you first clean the data?

- Instead of removing `NaN` values from the `VendorID` column, set it to a new value: 3. How does that affect your specifications and cleaning of the data?
- We'll talk more about this in 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.

xercise 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 that fits the format fine: the Unix `passwd` file. This file, which is standard on Unix and Linux systems, contains usernames and passwords. Over the years, it has evolved such that it no longer contains the actual passwords. Although MacOS is based on Unix, it doesn't use the `passwd` file for most user logins.

Specifically, do the following:

1. 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 `:`.
2. Add column names: `username`, `password`, `userid`, `groupid`, `name`, `homedir`, and `shell`.
3. Make the `username` column 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.

Working it out

For this exercise, we pull out all the stops, passing more arguments to `read_csv` than ever before. Each is necessary to parse the `passwd` file correctly, turning it into a data frame we can query. Over time, you'll discover that certain parameters to `read_csv` are used in nearly every project, making it easier to remember them.

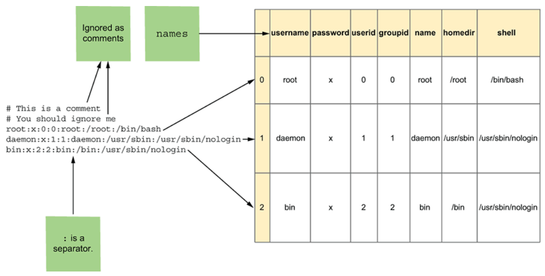
Let's review each keyword argument that we pass to `read_csv`, look at what it does, and see how the value we pass allows us to read `passwd` into a data frame. For starters, CSV files are named for the default field separator, the comma. By default, pandas assumes that we have comma-separated values. It's fine if we want to use another character, but then we need to specify that in the `sep` keyword argument. In this case, our separator is `:`, so we pass `sep=':'` to `read_csv`.

Next, we 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 in their `passwd` files, but given that some do, we should handle them. And `read_csv` does this elegantly, letting us specify the string that marks the start of a comment line. By passing it `comment='#'`, we indicate that the parser should 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 its first line, we can set `header` to an integer value, indicating 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, we can tell `read_csv` that there is no header with `header=None`.

What about the blank lines? We get off easy here because `read_csv` ignores blank lines by default. If we want to treat blank lines as `NaN` values, we can pass `skip_blank_lines=False` rather than accepting the default value of `True`.

The final keyword argument we pass is `names`. If we don't give any names, 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. Besides, it's easy to pass the names we want to give our columns as a list of strings. Here, we pass the same list of strings we described in the exercise description (figure 3.9).



```
# This is a comment
# You should ignore me
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

	username	password	userid	groupid	name	homedir	shell
0	root	x	0	0	root	/root	/bin/bash
1	daemon	x	1	1	daemon	/usr/sbin	/usr/sbin/nologin
2	bin	x	2	2	bin	/bin	/usr/sbin/nologin

Figure 3.9 Turning the `passwd` file into a data frame

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

Field separators and regular expressions

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

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

Solution

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

You can explore an abridged version of this in the Pandas Tutor at **http://mng.bz/G9lv**.

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 exercises to further help you understand how to massage `passwd` file into various types of data frames:

- Ignore the `password` and `groupid` fields so they don't appear in the data frame.
- Unix systems typically reserve user IDs below 1000 to special accounts. Show the nonspecial 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?

xercise 19 • Bitcoin values

When we think about CSV files, it's often in the context of data that has been collected once and that we now want to examine and analyze. But there are numerous examples of computer systems that publish updated data regularly and 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 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

This last case is the most interesting and will be the focus of this exercise. We can pass a URL to `read_csv`, and assuming 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 we're reading from a URL rather than from a file on a filesystem.

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

Using "requests"

In many cases, CSV files published to a URL 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 can't retrieve directly via `read_csv`. Rather, you 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 can 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
```

① Example URL

② Turns the content into a string and uses it to create a StringIO

③ Passes the StringIO to read_csv, returning a data frame

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 Bitcoin's price history in CSV format at 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.

Working it out

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.

We can read the CSV file into memory by passing a URL to `pd.read_csv`. There are only two columns to read, but there are no headers—so we have to say `header=None`. Then we give names to the columns, `date` and `value`:

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

Once we have created our data frame, we want to retrieve the closing price for the most recent day. Given that this kind of program can be run daily 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 the data from the most recent update. Rather, we only want to see the `value`. How can we get that? By realizing that we get a data frame back from `df.tail(1)`. We can request a particular column from that data frame: `value`.

Want just the value?

`df.tail(1)` returns the final row of `df`, which contains both the `date` and `value` columns. What if we only want `value`?

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 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 list only one column?

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

The result is 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, it doesn't matter.

Next, I asked you to find the minimum and maximum values and to show the corresponding `date` and `value`. We can use a boolean index to find the rows—or, more likely, a 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 we look for the minimum value of `value` and then find all the rows equal to that, effectively finding the row with the `min` value. In theory, two rows may both have the same value, in which case we show both of them. We then repeat this for the `max` value (figure 3.10):

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

However, there is another, more elegant approach: if we turn the `date` column into the data frame's index, we can then invoke `idxmin` and `idxmax` on the data frame. These method calls return not just the minimum/maximum values but also the indexes associated with these values—that is, the dates:

```
df.set_index('date').idxmin()
df.set_index('date').idxmax()
```

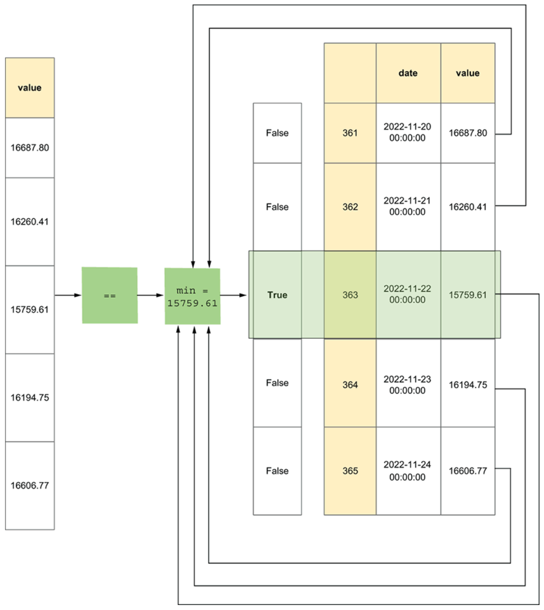


Figure 3.10 Selecting the minimum value from a data frame with a mask index

But why stop there? We can use the `agg` method to invoke more than one aggregation method on a data frame, passing the methods as a list of strings. We can set the data frame's index to be the `date` column and then run `agg` for both `idxmin` and `idxmax` in a single line:

```
df.set_index('date').agg(['idxmin', 'idxmax'])
```

Solution

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

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

df.tail(1)[['value']]                            ②
df.set_index('date').agg(['idxmin', 'idxmax'])   ③
```

① Names the columns date and value

② Retrieves the value column from the final row of df

③ Sets date to be the index and then find the rows (index + value) with the min and max values

You can explore an abridged version of this in the Pandas Tutor at **http://mng.bz/YRXB**.

Beyond the exercise

Pandas is full of amazing functionality that lets us retrieve data from the internet in various 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, you 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 that 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 will encounter HTML-formatted text containing at least one table. It turns each table into a data frame and then returns a list of those data frames. With this in mind, retrieve one 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. Note that Yahoo seems to look at the `User-Agent` header in the HTTP request, which cannot be set in `read_html`. So you'll need to use `requests` to retrieve the data, setting `User-Agent` to a string equal to `'Mozilla 5.0'`. Turn the content of the result into a `StringIO`, and then feed that to `read_html` and retrieve the data.
- 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.

xercise 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 various programming languages. Because it's easier to work with, smaller than XML, and more expressive than CSV, it's no surprise that JSON has become a common format for storing and exchanging data. JSON has also become the standard format for internet APIs, allowing us to access various 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 you?
- Along these lines, if you 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 on this list?
- Which state has the smallest number of cities on this list?

Working it out

Reading a JSON file into a data frame doesn't have to be difficult; in this case, it's easy. That's partly because this particular JSON file is an array of objects, or what Python people call a "list of dicts." When `read_json` sees this file, it sees each dict 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.

First, I asked you to compare the mean and median city populations. We can do that with `describe` in the `population` column, which returns a series. Because 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%']]
```

The mean population is 131,132, and the median is 68,207. This means a few big values pull the mean higher than the median. And indeed, the United States has a few very large cities and many medium- and small-size cities. By definition, half of these 1,000 cities have populations less than 68,207.

The next question asks, what if we ignore the 50 largest cities? What does that do to the mean and median? For that, we 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 indicate that we want all the rows starting with index 50. And we only want one column: `population`. Once again, we run `describe` and then grab only the mean and median values. We find that the mean has dropped a lot, to 87,027, and 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 the maximum positive value for `latitude`. We can find that by getting the max latitude and finding which rows of `df` have that value. Once again, we use `loc` to retrieve only those rows and then pass a list of columns to retrieve only those values:

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

The result, not surprisingly, is Anchorage, Alaska, which is the 63rd largest city in the United States—a much higher rank than I expected!

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, is 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 the next-most-common state doesn't have the same value. (As far as I know, there isn't a good way to avoid such problems.)

What about the states with the fewest cities on this list? We use `tail(10)` to look at the 10 lowest-ranked states and find that the bottom 5 states (including Washington, DC) all have a single city in the list:

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

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', 'rank']]   ③
```

```
df['state'].value_counts().head(1)    ④
df['state'].value_counts().tail(5)    ⑤
```

① Grabs just the mean and 50% values for the population descriptive statistics

② Grabs just the mean and 50% values for the population descriptive statistics for rows 50 and up

③ Finds the maximum latitude value and gets only the city, state, and rank for it

④ One state has the most cities, which we can see here.

⑤ Five states have only 1 city in the top 1,000.

You can explore an abridged version of this in the Pandas Tutor at **http://mng.bz/z0oB**.

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 has no recorded growth, set it to 0.
- How many cities had positive growth in this period, and how many had negative growth?
- Find the city or cities with latitudes more than two standard deviations from the mean.

ummary

In this chapter, we started to work with real-world data. We read data from CSV, JSON, and even HTML tables and saw how pandas provides 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 to learn how to read data from them—specifying the `dtype` for each column and even which columns we want to see.