11 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 understanding large collections of numbers. We're generally much better at comprehending visual depictions of numbers, especially if we're trying to understand relationships among our data. So, although 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 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.

In this chapter, we'll look at how to visualize data using the pandas wrapper for Matplotlib. We'll explore a number of different plots that can help make your data come alive.

We'll also spend some time looking at Seaborn, a popular alternative to Matplotlib. There are a number of such alternatives; some (like Seaborn) are wrappers around the Matplotlib library, and others are full-blown alternatives written from the ground up. It's worth learning what your options are so you can find a system with which you feel comfortable. I've grown to like Seaborn's API as well as its ability to create attractive plots with little or no customization.

This chapter also provides you with the opportunity to explore one of Jupyter's best features: the fact that it keeps images inline (figure 11.1). 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.

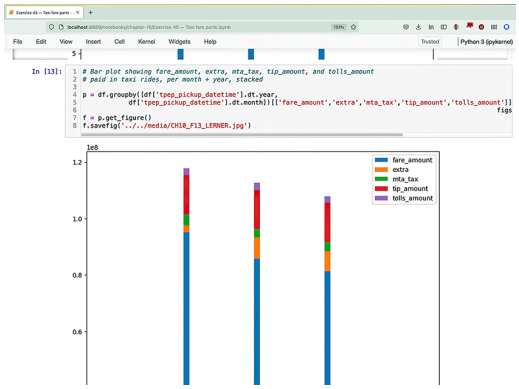


Figure 11.1 Screenshot of a Jupyter notebook combining code and plots

Table 11.1 What you need to know

Concept	What is it?	Example	To learn more
`pd.read_csv`	Returns a new data frame based on CSV input	`df = df.read_csv('myfile.csv')`	**http://mng.bz/a1az**
`df.groupby`	Allows us to invoke one or more aggregate methods for each value in a particular column	`df.groupby('year')`	**http://mng.bz/gBGl**
`df.loc`	Retrieves selected rows and columns	`df.loc[:, 'passenger_count'] = df['passenger_count']`	**http://mng.bz/pPNG**
`df.plot`	Plotting object for a data frame	`df.plot.box()`	**http://mng.bz/Ox8n**
`df.corr`	Produces a data frame describing correlations among each pair of numeric columns	`df.corr()`	**http://mng.bz/Y1oN**

`s.quantile`	Gets the value at a particular percentage of the values	`s.quantile(0.25)`	http://mng.bz/GyYq
`df.join`	Joins two data frames based on their indexes	`df.join(other_df)`	http://mng.bz/zXva
`pandas.plotting.scatter_matrix`	Creates scatter plots comparing every pair of numeric columns	`pandas.plotting.scatter_matrix`	http://mng.bz/0Kqx
Matplotlib	Python library for plotting data	`import matplotlib.pyplot as plt`	http://mng.bz/9D6l
Seaborn	Python library for plotting data	`import seaborn as sns`	http://mng.bz/jPKx
`df.reset_index`	Gets a data frame identical to our current one, but with a new numeric	`df.reset_index(drop=True)`	http://mng.bz/Wz50

	index starting at 0		
`pd.concat`	Returns a list of data frames combined as a single new data frame	`df = pd.concat (df1, df2)`	http://mng.bz/8r5P

Exercise 43 • Cities

Back in exercise 20, we worked with a JSON file describing the 1,000 largest cities in the United States. In this exercise, we look at the same file—but instead of printing the analysis as a bunch of numbers, we visualize some of the most interesting numbers and trends in the file. Specifically, I want you to

1. Load data from cities.json into a data frame.
2. Create a bar plot showing how many of the top 1,000 cities are in each state. There should be one vertical bar per state (with a few extra for nonstates such as Washington, DC). The plot should be ordered such that the state with the fewest cities in this list is on the left and the state with the most cities is on the right.

3. Create a bar plot comparing the growth of all cities in the state of Pennsylvania. There should be one vertical bar per city, ordered with the slowest-growing city on the left and the fastest-growing city on the right.
4. Create a pie plot showing how much each Massachusetts city in the list contributes to the overall population. (And no, I'm not trying to say that 100% of the population of that state resides in large cities.) There should be one pie segment per city in the list, and its size should indicate how much it contributes to the total.
5. Create a scatter plot of the cities, putting the longitude on the *x* axis and latitude on the *y* axis. What does the resulting plot look like?

Working it out

Matplotlib offers a wide variety of plotting formats, and we use this exercise to explore a number of them, trying different techniques to understand our data in a variety of ways. Visualization isn't just about choosing a type of plot; we often need to clean, arrange, and modify the data before we 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 we create from the JSON has several columns, one of which is `state`. We 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 a result for every column in the data frame. Because we're only interested in the number of cities, we can choose a single column—in this case, the `city` column:

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

With that in place, we can create a bar plot. But wait: the question asks for the bar to be sorted from the smallest value to the largest. This means before producing the plot, we need to sort the values in the series returned by the `groupby` call. Fortunately, sorting a series is easily done with `sort_values`:

```
(
    df
    .groupby('state')['city'].count()
    .sort_values()
)
```

With that in place, we can produce our 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 small. We thus pass the `figsize` keyword argument to `bar`, which is in turn passed 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:

```
(
    df
    .groupby('state')['city'].count()
    .sort_values()
    .plot.bar(figsize=(10,10))
)
```

It's probably not particularly surprising that California has the most large cities, but the sheer number (and thus very tall bar in

our plot) was still striking to me when producing this plot (figure 11.2).

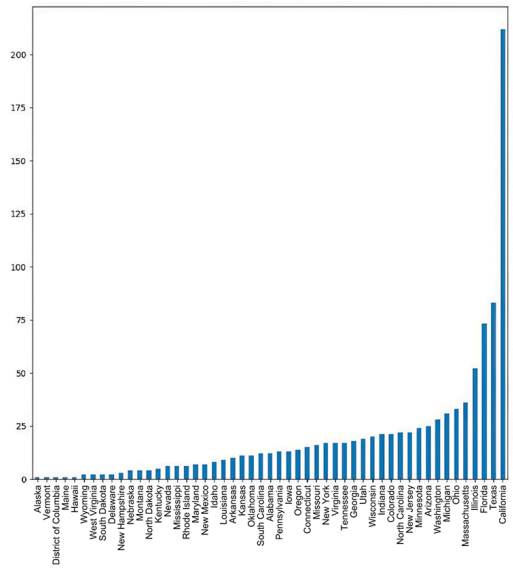


Figure 11.2 Bar plot showing how many large cities are in each state

Next, I asked you to create a bar plot showing growth in Pennsylvania cities, sorted from lowest to highest. For this task, we took data from the `growth_from_2000_ to_2013` column, along with the `city` column, all from rows in which `state` equals to `'Pennsylvania'`. I decided it would be easiest to turn these rows and columns into a separate, smaller data frame using `df.loc`:

```
  df.loc[df['state']=='Pennsylvania',           ①
         ['city','growth_from_2000_to_2013']    ②
      ]
```

① Row selector: only rows describing Pennsylvania cities

② Column selector: only two columns, the city name and its growth

As we've seen on many occasions, we select rows in which the state is equal to `'Pennsylvania'` and then the two columns that are of interest. Then, because we want to show the city names in our plot's x index, we make it the index of the data frame:

```
(
    df.loc[
        df['state']=='Pennsylvania',
        ['city','growth_from_2000_to_2013']]
    .set_index('city')                                  ①
)
```

① Sets the city name to be the index

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 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 so, 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_2013']]
    .set_index('city')
    ['growth_from_2000_to_2013']      ①
)
```

① Retrieves the growth column

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 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_2013']]
    .set_index('city')
    ['growth_from_2000_to_2013']
    .str.replace('%', '')          ①
)
```

① Removes the % sign from each growth string

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_2013']]
    .set_index('city')
    ['growth_from_2000_to_2013']
    .str.replace('%', '')
    .astype(np.float16)      ①
)
```

① Gets a float16 column from the growth string

We now have every city in Pennsylvania along with its growth percentage. We can 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_2013']]
    .set_index('city')
    ['growth_from_2000_to_2013']
    .str.replace('%', '')
    .astype(np.float16)
    .sort_values()
)
```

And with that in place, we create a bar plot, setting a size of `(10, 10)` to see it more easily in our notebook (figure 11.3):

```
(
    df.loc[
        df['state']=='Pennsylvania',
```

```
          ['city','growth_from_2000_to_2013']]
      .set_index('city')
      ['growth_from_2000_to_2013']
      .str.replace('%', '')
      .astype(np.float16)
      .sort_values()
      .plot.bar(figsize=(10,10))
  )
```

Next, I asked you to find all cities in Massachusetts and create a pie plot with all 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 the values, sums them, and produces a pie “slice” of that item’s proportion of the total.

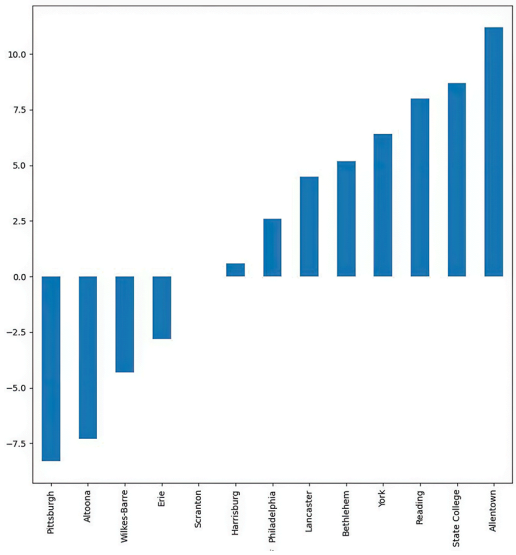


Figure 11.3 Bar plot showing growth in Pennsylvania cities

We first need to get names and populations of cities in Massachusetts. We can do that using the following query:

```
(
    df
    .loc[
        df['state'] == 'Massachusetts',     ①
        ['city','population']]              ②
    .set_index('city')                      ③
    ['population']                          ④
)
```

① Row selector: Only cities in Massachusetts

② Column selector: Two columns, city and population

③ Uses the city name as the index

④ Retrieves only the population column

This is 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 set the index of our data frame to be `city` and then retrieved the only remaining column, `population`, as a series.

Next, we draw 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_index('city')
      ['population']
      .plot.pie(figsize=(10,10))
  )
```

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 (figure 11.4).

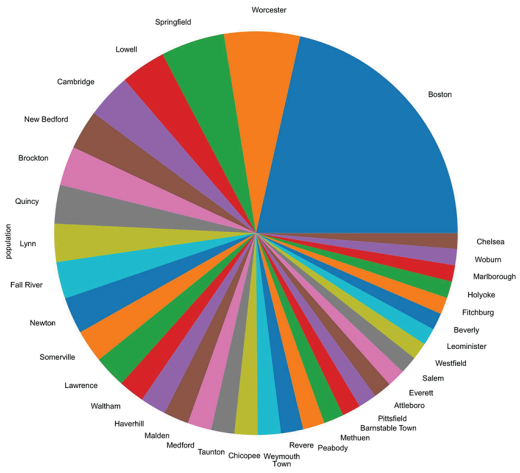


Figure 11.4 Pie chart showing the population of Massachusetts cities

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 the plot will look like . . . a map of the United States, at least the most densely populated areas (figure 11.5).

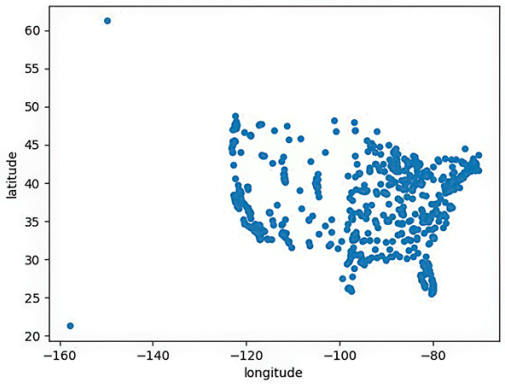


Figure 11.5 Scatter plot of cities' longitude and latitude

Solution

```
filename = '../data/cities.json'
df = pd.read_json(filename)                                    ①

(
    df
    .groupby('state')['city'].count()
    .sort_values()
    .plot.bar(figsize=(10,10))                                 ②
)

(
    df.loc[
        df['state']=='Pennsylvania',                           ③
        ['city','growth_from_2000_to_2013']]                   ④
    .set_index('city')
    ['growth_from_2000_to_2013']
    .str.replace('%', '')                                      ⑤
    .astype(np.float16)
    .sort_values()
    .plot.bar(figsize=(10,10))                                 ⑥
```

```
)

(
    df
    .loc[
        df['state'] == 'Massachusetts',
        ['city','population']]                    ⑦
    .set_index('city')                            ⑧
    ['population']
    .plot.pie(figsize=(10,10))
)

df.plot.scatter(x='longitude', y='latitude')      ⑨
```

① Reads the JSON into a data frame

② Gets the number of cities grouped by state, sorts the values, and creates a bar plot

③ Gets all cities in Pennsylvania

④ Only columns city and growth_from_2000_to_2013

⑤ Removes the % sign

⑥ Turns into a float, sorts values, and creates a bar plot

⑦ Gets all cities in Massachusetts, columns city and population

⑧ Creates a pie plot of the cities' populations

⑨ Creates a scatter plot from cities' longitude and latitude

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 boxplots, more formally known as *box-and-whisker plots*, and I was intrigued.
>
> We frequently use the `describe` method to describe 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 understand our data.
>
> The "Tukey" in this name refers to John Tukey, a famous

mathematician and statistician. Tukey developed not only 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, you probably know what software is.)

For example, let's create a simple series:

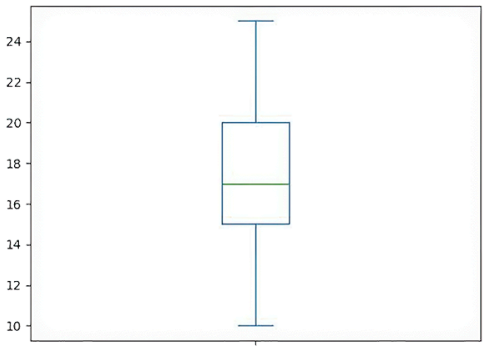
```
s = Series([10, 15, 17, 20, 25])
```

We can get all the descriptive statistics, including the five-number summary with `s.describe()`:

```
count     5.00000
mean     17.40000
std       5.59464
min      10.00000
25%      15.00000
50%      17.00000
75%      20.00000
max      25.00000
dtype: float64
```

A boxplot shows us this five-figure summary, but in graphical form. We can create boxplots in pandas using `plot.box` on a series (for a single plot) or a data frame (for one plot for each numeric column). We create a boxplot from `s` with

```
s.plot.box()
```



Boxplot from our series `s`

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.

A boxplot often has circles above and below the whiskers. These represent the outliers, defined in the case of our boxplots to be 1.5 *

IQR (interquartile range) below the first quartile (25% mark) or 1.5 * IQR above the third quartile (75% mark).

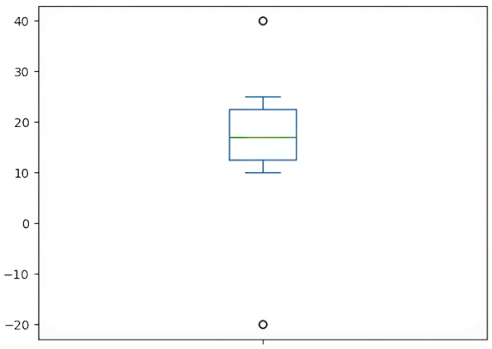
For example, here's another series:

```
s = Series([-20, 10, 15, 17, 20, 25, 40])
```

The descriptive statistics are as follows:

```
count     7.000000
mean     15.285714
std      18.273061
min     -20.000000
25%      12.500000
50%      17.000000
75%      22.500000
max      40.000000
dtype: float64
```

And the boxplot?



Boxplot from our series `s` with outliers

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. Plotting the different columns from a data frame can be particularly useful when creating machine-learning models, when having all the data in the same range increases the model's accuracy. Note that if you put all the columns in the same boxplot, they share a *y* axis, which is perfect for checking that they're in the same range. You can, however, have a separate *y* axis for each column if you pass `subplots=True` to your call:

```
df.plot.box(subplots=True)
```

Note that nowhere in the boxplot do we see the mean value. I personally think that's a shame, because the mean can also be a useful measure, imperfect though it may be.

Exercise 44 • Boxplotting weather

One of the phrases I often use when teaching data analytics is that you need to "know your data." And one of the best ways to know your data is with a boxplot. In this exercise, we use boxplots to understand the weather during the

winter of 2018–2019, using data in three different US cities. We 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).

Do the following:

1. 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`.
2. Create a boxplot of Chicago's minimum temperatures during this period.
3. Find the values that are represented as dots on that boxplot.
4. Create a boxplot of Chicago's minimum temperatures in February.
5. Create a side-by-side boxplot of Chicago's minimum and maximum temperatures in February and March.
6. Read in data from Los Angeles and Boston, 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.

7. Get descriptive statistics for `mintemp` and `maxtemp` grouped by city.
8. Create side-by-side boxplots showing minimum and maximum temperatures for each of the three cities.

Working it out

In this exercise, we combine techniques we've seen previously: specifically, using `read_csv` with a variety of parameters, combining several CSV files into a single data frame, and using a `datetime` column as an index. But the main point of this exercise is to create a number of different boxplots and, in so doing, 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 the index of type `datetime`. I also asked you to load the columns with the minimum and maximum temperatures found on each day. We do 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 we use them all at once. For starters, we indicate that we're interested in only the first three columns. In previous exercises, we often referred to these columns by name, using the names provided by the index. But here, we refer to the columns by number. That's because we want to give them names of our own, specified in the `names` parameter. We thus choose them by number and rename them in `names`. We also indicate that `date_time` should be parsed as a `datetime` column and used as the index of the data frame. Finally, just to be on the safe side, we pass `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, we end up with a data frame with 728 rows and 2 columns. The values start at midnight on December 12, 2018, and end at 9:00 p.m. on March 11, 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 (figure 11.6):

```
df['mintemp'].plot.box()
```

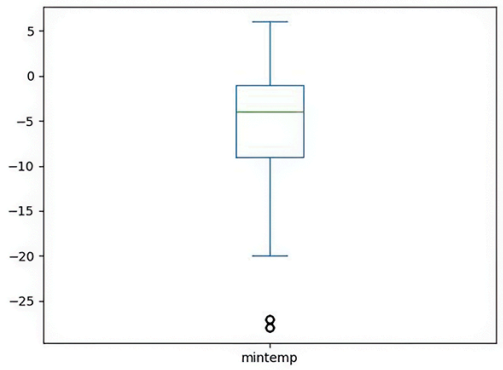


Figure 11.6 Boxplot of minimum temperatures in Chicago

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 at least 1.5 * IQR below the 0.25 mark or at least 1.5 * IQR above the 0.75 mark. Just to double check that the plot is showing them correctly, I asked you to find those values:

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

(
    df.loc[
        df['mintemp'] < df['mintemp'].mean() - (iqr_ 1.5),
        'mintemp'
        ]
)
```

Sure enough, we see a number of temperature readings (on January 30 and 31) when the temperature was –27 and –28 degrees Celsius—not only cold, but unusually cold, even for a Chicago winter. Our boxplot is thus right to show them as outliers.

Next, I asked you to create a boxplot for Chicago's minimum temperatures in February. We solve this as follows:

```
(
    df
    .loc[
         '01-Feb-2019':'28-Feb-2019',     ①
         'mintemp']                       ②
   .plot.box()
)
```

① Row selector in February 2019

② Column selector of mintemp

Our row selector is the slice from February 1 through February 28. Here we take 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. We choose the `mintemp` column and feed the resulting one-column data frame to `plot.box` (figure 11.17). 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 February and March. We again use a slice to select the appropriate rows, stretching from February 1 through March 30:

```
(
    df
    .loc['01-Feb-2019':'30-Mar-2019',    ①
        ['mintemp','maxtemp']]           ②
    .plot.box()
)
```

① Row selector, all of February and March

② Column selector, both mintemp and maxtemp

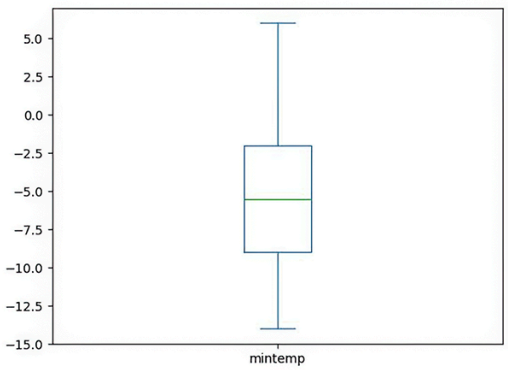


Figure 11.7 Boxplot for minimum Chicago temperatures in February

Once again, we select rows using a slice. But the column selector needs to be a list of strings: the names of the columns that we

want to plot. We then pass these to `plot.box` and get two boxplots displayed on the same scale, next to one another (figure 11.8).

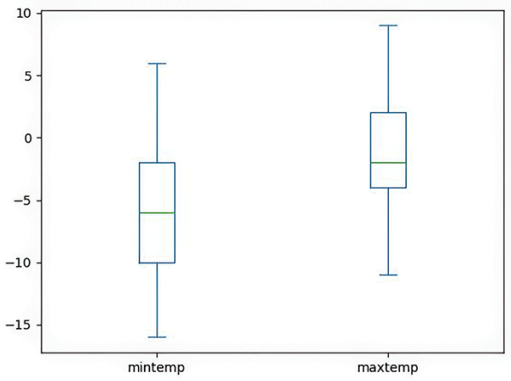


Figure 11.8 Boxplot for minimum and maximum Chicago temperatures in February and March

Having experienced, if only on paper, the cold Chicago winter, I thought 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. 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. Because `df` already contains information for Chicago, we set that value right away:

```
df['city'] = 'Chicago'
```

To load the other data, we use a `for` loop—typical in day-to-day Python programming, but unusual in pandas. Here, the loop runs not 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 we do here:

1. We set up a list with the filenames (minus the `'csv'` suffix) over which we want to run.
2. We use a `for` loop to iterate over those filenames.
3. We reuse the `read_csv` call that we used earlier, passing the complete filename.
4. As before, we select specific columns, indicating that `date_time` should be parsed as a `datetime` and set to the index.

5. We add a value to `city` for each of the loaded cities using a bunch of string methods to convert `city_stem` into a useful string:
    1. We use `str.split` on `city_stem`, getting a list—from which we take the initial part.
    2. We replace the character `'+'` with a space, `' '`.
    3. We invoke `str.title`, capitalizing each word.

Finally, we use `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 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 measurements for `mintemp` and then all measurements for `maxtemp`. But although these details may be interesting and useful, they're not as compelling as a boxplot. I thus asked you to

create a boxplot showing minimum and maximum temperatures for all three cities, grouped together. We solve it as follows:

```
(
    df
    .plot.box(column=['mintemp', 'maxtemp'],
              by='city')
)
```

This produces two side-by-side boxplots, one for `mintemp` and the second for `maxtemp`. In each plot, we see the five-number summary for each city, side by side (figure 11.9). It isn't a surprise to find that although Boston's winter months are warmer than Chicago's, Los Angeles is far warmer than either of them.

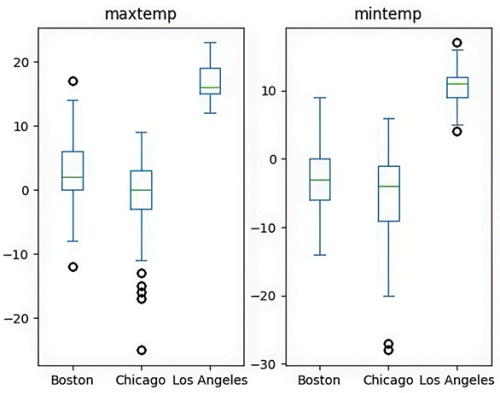


Figure 11.9 Boxplot for minimum and maximum temperatures in several cities

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

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

(
    df.loc[
        df['mintemp'] < df['mintemp'].mean() - (iqr * 1.5),
        'mintemp'
        ]
)                                                                           ⑦

(
    df
    .loc['01-Feb-2019':'28-Feb-2019',
    'mintemp']
    .plot.box()
)                                                                           ⑧

(
    df
    .loc['01-Feb-2019':'30-Mar-2019',
        ['mintemp','maxtemp']]
    .plot.box()
)                                                                           ⑨

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('+', ' ').title()      ⑬
    df = pd.concat([df, new_df])                                            ⑭

df.groupby('city')[['mintemp', 'maxtemp']].describe()                       ⑮
```

```
(
    df
    .plot.box(column=['mintemp', 'maxtemp'],
              by='city')
)
```
⑯

① Loads only three columns

② Explicitly tells pandas that the first line contains header info

③ Names the three columns we load

④ The date_time column should be parsed as a datetime.

⑤ Sets date_time to be an index

⑥ Creates a boxplot of min temp in Chicago

⑦ Finds the outliers, assuming 2.5*std above/below mean

⑧ Boxplot of February min temp in Chicago

⑨ Boxplot of Feb–March min+max temps in Chicago

⑩ New column, city, all with 'Chicago' values

⑪ Loads additional cities

⑫ Loads the CSV for each new city

⑬ Uses Python string methods to set the city name from the filename

⑭ Adds the new data frame to the existing ones

⑮ Gets descriptive statistics for temps by city

⑯ Boxplot of min and max temps by city

Beyond the exercise

- Rather than starting with data from Chicago, begin 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 are 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.

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 effect a good plot can have when presenting it to others, but how much better it can help you understand the data set yourself. You'll see new relationships in the data and know how to answer questions you already asked as well as what new questions you should be asking.

I'd like you to do the following:

1. Load data from all four NYC taxi files into a single data frame. We 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`.
2. 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.)
3. Create a bar plot showing the total amount paid in taxi rides for every year and month of the data set.

4. Create a bar plot showing `fare_amount`, `extra`, `mta_tax`, `tip_amount`, and `tolls_amount` paid in taxi rides per month and year, with the various components stacked in a single bar per year/month.
5. 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.
6. Create a histogram showing the frequency of each tipping percentage between (and including) 0% and 50%.

Working it out

This exercise is all about visualizing our taxi data. 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 July 2020. We load them, as we've done before, most recently in exercise 42, via a list comprehension:

```
filenames = ['../data/nyc_taxi_2019-01.csv',
             '../data/nyc_taxi_2019-07.csv',
             '../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',
                      'trip_distance',
                      'fare_amount',
                      'extra',
                      'mta_tax',
                      'tip_amount',
                      'tolls_amount',
                      'improvement_surcharge',
                      'total_amount',
                      'congestion_surcharge'],
             parse_dates=['tpep_pickup_datetime'])
             for one_filename in filenames]
```

In this case, we pass `usecols` the list of columns I asked for in the question. We also pass `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 create a list of data frames, which we can then 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. To do this, we 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. We run the query as follows:

```
(
    df
    .groupby([df['tpep_pickup_datetime'].dt.year,
              df['tpep_pickup_datetime'].dt.month])
    ['total_amount'].sum()
)
```

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 files, each supposedly containing one month of data, it may seem strange that we have data from other months and years. Some of that data may not have been stored in New York's databases when it was first created. Or the computer wasn't set to the right date. Or the data may be corrupt. Likely it's a combination of these and other 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'].sum()
    .plot.bar(figsize=(10,10))
)
```

The call to `plot.bar` creates the bar plot based on the data frame we get from the `groupby` (figure 11.10). That data frame's index serves as the plot's *x* axis and the values determine the y axis, which we allow to be generated automatically.

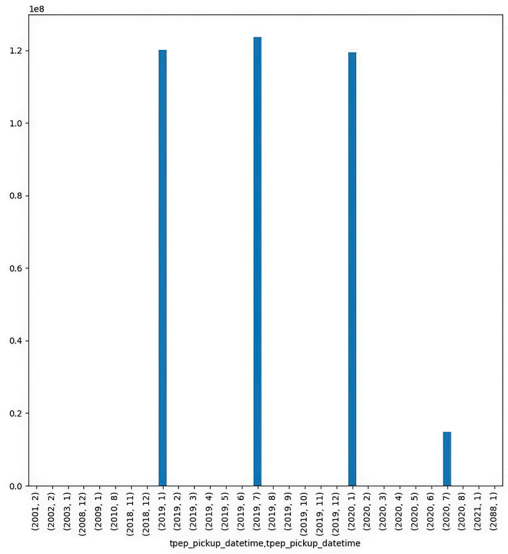


Figure 11.10 Bar plot showing how much money was paid to taxis in each month and year

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])
    ['passenger_count'].count()
    .plot.bar(figsize=(10,10))
)
```

Here, we're not interested in totaling the receipts but rather in counting the rows. Although we could run `count`, the aggregation method that counts rows, on the entire data frame, that would give us the count for every column. (So if a data frame has 10 columns, running `df.count()` will give you 10 results, one for each column.)

NOTE Because `count` only counts the number of non-`NaN` values, it can sometimes come in handy, allowing you to see which columns contain more (or fewer) `NaN` values.

We don't really need that. So we 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'].count()
)
```

Finally, we take this data frame and turned it into a bar plot. As before, we called `plot.bar` with a keyword argument of `figsize=(10, 10)`, ensuring that the image is a 10-inch square (figure 11.11):

```
(
    df
    .groupby([df['tpep_pickup_datetime'].dt.year,
              df['tpep_pickup_datetime'].dt.month])
    ['passenger_count'].count()
    .plot.bar(figsize=(10,10))
)
```

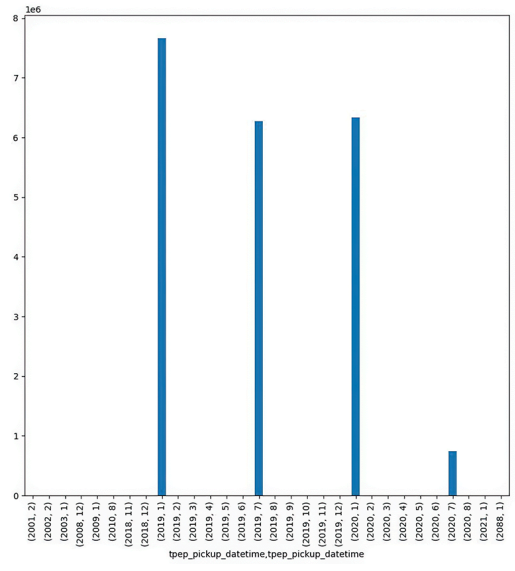


Figure 11.11 Bar plot showing how many taxi rides occurred per month and year

Although the *x* axis is the same in this plot and the previous one, and although we see bars in the same

places, the values are obviously different. Moreover, although July 2019 was the month with the greatest amount of revenue, it had the third-most 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.

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. Although 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 plot the amount of revenue each month in the data set and 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, we use `groupby` on the year and month columns:

```
(
    df.groupby([df['tpep_pickup_datetime'].dt.year,
                df['tpep_pickup_datetime'].dt.month])
)
```

Because we want to produce the plot with input from five columns — `fare_amount`, `extra`, `mta_tax`,

`tip_amount`, and `tolls_amount` — we name 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','extra','mta_tax',
      'tip_amount','tolls_amount']]
)
```

We then run the `sum` method, which gives us 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','extra','mta_tax',
      'tip_amount','tolls_amount']].sum()
    .plot.bar(stacked=True, figsize=(10,10))
)
```

Finally, we ask 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, figsize=(10,10))
)
```

However, there's a difference between our previous calls to `plot.bar` and this one: normally

we would get a separate plot for each column for each month. But because we specified `stacked=True`, we get all the bars for a given month stacked on top of one another. Moreover, each portion of the bar is in a different color, and pandas provides a legend, as well. In this way, we can 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 (figure 11.12). We can see that although the fare is by far the greatest proportion of the total taxi revenue, tips constitute a fairly large proportion, followed by `extra` charges, taxes, and tolls.

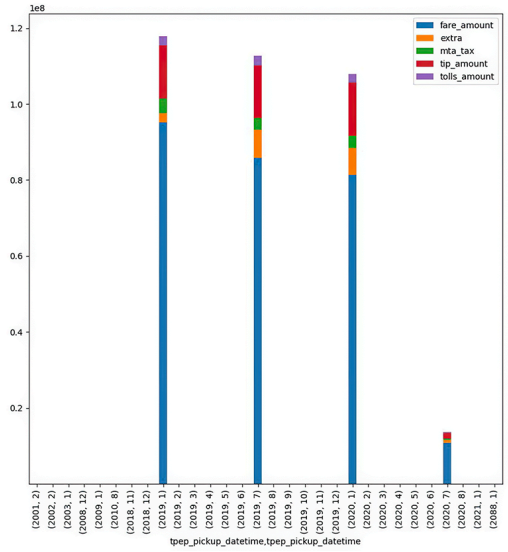


Figure 11.12 Stacked bar plot showing the relative components that go into the total fare per year and month

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 query similar to the previous one, grouping on `passenger_count` rather than by year-month combination (figure 11.13):

```
(
    df
    .groupby(df['passenger_count'])
    [['fare_amount','extra','mta_tax',
      'tip_amount','tolls_amount']].sum()
    .plot.bar(stacked=True, figsize=(10,10))
)
```

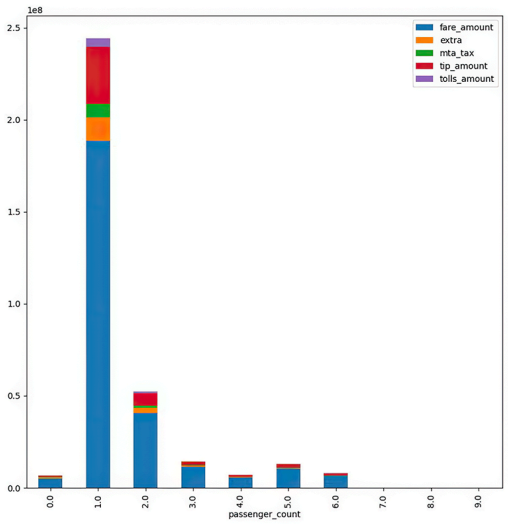


Figure 11.13 Stacked bar plot showing the relative components that go into the total fare per number of passengers

Finally, I asked you to create a histogram showing the frequency of each tipping percentage between (and including) 0 and 50. To do this, we need to find the tipping percentage for each ride and keep only those between 0 and

50. The easiest thing would be to create a new column, `tip_percentage`, by dividing `tip_amount` by `fare_amount`. But the real world includes all sorts of surprises, including `NaN` values and records in which the `fare_amount` is equal to zero—thus giving us an infinite (known as `np.inf`) value. To avoid this, we first get rid of any ride in which the fare was less than or equal to 0:

```
df = df[df['fare_amount'] > 0]
```

Then we create a new column, `tip_percentage`, knowing we won't get any `np.inf` values:

```
df['tip_percentage'] = df['tip_amount'] / df['fare_amount']
```

Finally, we plot all the values less than or equal to 50%:

```
(
    df
    .loc[
         df['tip_percentage'] <= .50,    ①
         'tip_percentage']               ②
    .plot.hist()
)
```

① Row selector

② Column selector

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% (figure 11.14).

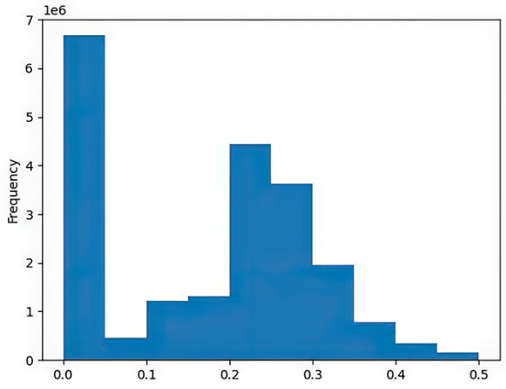


Figure 11.14 Histogram showing, across all rides, what nonzero percentage New York riders tip

We can create this histogram another way, using method chaining along with `.loc`, `assign`, and `lambda`:

```
(
    df
    .loc[lambda df_: df_['fare_amount'] > 0]
    .assign(tip_percentage =
            lambda df_: df_['tip_amount'] / df_['fare_amount'])
    .loc[lambda df_: df_['tip_percentage'] <= 0.5,
         'tip_percentage']
    .plot.hist()
)
```

It's easiest to understand this code if you read it one line at a time:

1. We use `.loc` to find all rows of `df` where `fare_amount` is more than 0. We use a `lambda` here, along with a temporary variable of `df_`, which is typical when chaining methods because it ensures that we're working with the data frame we get rather than the original `df`.
2. We use `assign` to create a new column, `tip_percentage`. Its value is the result of running a function, defined with `lambda`, that takes each row and divides `tip_amount` by `fare_amount`. The column created by `assign` isn't actually added to `df`; it's only on the data frame we're building via method chaining.
3. We again use `.loc` to keep only those rows where `tip_percentage` is less than 0.5. But here we use the two-argument value of `.loc`, filtering rows via the `lambda` and columns by explicitly naming the one we want: `tip_percentage`.
4. We call `plot.hist` and get a histogram.

Solution

```
filenames = ['../data/nyc_taxi_2019-01.csv',
             '../data/nyc_taxi_2019-07.csv',
             '../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',
                    'trip_distance',
                    'fare_amount',
                    'extra',
                    'mta_tax',
                    'tip_amount',
                    'tolls_amount',
                    'improvement_surcharge',
                    'total_amount',
                    'congestion_surcharge'],
           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])
    ['total_amount'].sum()
    .plot.bar(figsize=(10,10))
)

(
    df
    .groupby([df['tpep_pickup_datetime'].dt.year,
              df['tpep_pickup_datetime'].dt.month])
    ['passenger_count'].count()
    .plot.bar(figsize=(10,10))
)

(
    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, figsize=(10,10))
)

(
    df
    .groupby(df['passenger_count'])
    [['fare_amount','extra','mta_tax',
      'tip_amount','tolls_amount']].sum()
```

```
    .plot.bar(stacked=True, figsize=(10,10))
)

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()

(
    df
    .loc[lambda df_: df_['fare_amount'] > 0]
    .assign(tip_percentage = lambda df_: df_['tip_amount'] / df_['fare_amount'])
    .loc[lambda df_: df_['tip_percentage'] <= 0.5,
         'tip_percentage']
    .plot.hist()
)
```

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, they're considered positively correlated. If one goes up when the other goes down (and vice versa), they're said to be negatively correlated.

In addition to being 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 may be stronger than you'd expect.)

Let's take a simple example. The more electric power you use, the higher your electric bill. 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 multibillionaire, you probably have a jet or 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 thing causes another. And in some cases, that's certainly true: we can safely say that your higher electric bill is caused by greater consumption.

But just because two data points are correlated doesn't mean 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, 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 website 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 doesn't mean 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 12:00 noon and 1:00 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 numeric values to be correlated? Let's take two sets of numbers, the high and low temperatures for the 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],
         '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, days with the highest high temperatures will also have the highest low temperatures. And days with the lowest high temperatures will have the lowest low temperatures.
- If the columns are negatively correlated, days with the highest high temperatures will have the lowest low temperatures. And days with the lowest high temperatures will have the highest low temperatures.

If the two are strongly correlated, a large change in one is accompanied by a large change in the other. If they're weakly correlated, a large change in one will be accompanied by a small change in the other.

The most common measurement for correlation, and what we 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 is 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 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 we 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
```

This is a very strong positive correlation. It 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 we input was from a particularly turbulent time with a high degree of variability. But I think the real reason is that the sample from Modi'in is extremely small, with only 13 data points. It's hard to establish any correlations based on such a small sample.

Why are we interested in correlation? First and foremost, because it can inform our understanding and thus our behavior. If I know that my store gets a huge number of requests at lunchtime each day, perhaps I'll provision additional servers during that time. Or perhaps I'll offer discounts outside of that window to encourage sales during otherwise dead times.

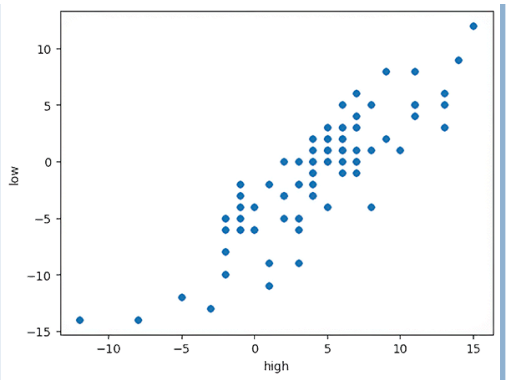
We can also use correlations to hint at underlying similarities and

relationships in our data. If two things are correlated, perhaps there's some behavior that explains the connection between the two. If that behavior or relationship isn't obvious, it can point to a topic worth investigating or understanding better.

Although 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 see a strong positive correlation matching the numeric calculations we performed earlier.



Scatter plot showing high vs. low Modi'in temperatures

Exercise 46 • Cars, oil, and ice cream

In this exercise, we'll try to answer a question that has probably occurred to you on many occasions: when the price of oil goes up, do people drive their cars more or less? And while we're at it, we'll also attempt to answer another question: whether the price of ice cream is correlated with the price of oil.

This exercise will not only try to identify these correlations but 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. Specifically, I want you to

1. Load the oil data (**wti-daily.csv**, as in exercise 41) into a data frame. Set the names of the columns to `date` and `oil`, with the `date` column parsed as a date and set to be the index.
2. 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.
3. Set the `icecream` column to be a floating-point value, removing any rows that stop you from accomplishing that.
4. Load historical US "miles traveled per month" data (from the file **miles-traveled.csv**) into a separate data frame. Name the columns `date` and `miles`, parsing `date` as a date and setting it to be the index.
5. 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.
6. Get the numeric correlations among the columns.

7. Create a scatter plot looking at the relationship between `oil` and `icecream`.
8. Create a scatter plot looking at the relationship between `oil` and `miles`.
9. Create a scatter matrix among all three columns.

Working it out

In this exercise, we take three distinct data sets, merge them to make a new data frame, and then find correlations among the various columns. And the results are . . . not what I was expecting, to say the least.

Before we can calculate the correlations, we have to load the data. I always like to create separate data frames and join them. This lets us do things step by step and ensures that we can debug, improve, and rerun our steps more easily.

The first data frame I asked you to create is similar to one we looked at in exercise 41. To make our `join` operation run more smoothly later, I asked you to standardize some parts of the naming. For example, we want to parse the `date` column as a datetime and also set it to be the index. We also rename 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 calls to `read_csv`. That makes the function call easier to read and debug. But when we want to rename the columns with the `names` parameter, we need to describe them numerically. Moreover, to avoid having the header row read as data, we need to indicate which row contains the header (0, in this case), effectively causing it to be ignored.

In the end, we load 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`) shows that we successfully created the data frame with the correct dtype. With the oil data in hand, it's time to create the next data frame based on the monthly ice cream price data from the US government.

This file is in a format very similar 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 CSV files. We can see where the problem is by trying to turn the column into an `np.float64` value:

```
ice_cream_df['icecream'].astype(np.float64)
```

Sure enough, it fails, telling us that it choked on a line containing nothing more than `.` instead of a price.

We decide to keep only those lines of `ice_cream_df` that contain at

least one digit, on the assumption that such values can be turned into a floating-point value. First, we create a boolean series based on `ice_cream_df['icecream']` with `True` wherever the value contains at least one digit. We do this using `str.contains` along with a regular expression, making sure to pass `regex=True`. We then convert the resulting value to `np.float64` using `astype`:

```
ice_cream_df = (
    ice_cream_df
    .loc[ice_cream_df['icecream'].str.contains(r'\d', regex=True)]
    .astype(np.float64)
)
```

Notice that we use 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 predigest our backslashes before they get to the regular expression engine.

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. We create the new data

frame using arguments similar to those 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's time to join them. We've already seen how we can join two data frames using the `join` method. But here I asked you to join three data frames. How can we do that?

The answer, once you see it, is straightforward: we join two data frames, getting a new one. We join this new data frame with the third to get a final new one. As long as all the data frames share an index, we should be fine:

```
df = oil_df.join(ice_cream_df).join(miles_df)
```

If we do things this way, we discover 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 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 we did previously and use `dropna` to remove all `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 our index will only contain values that existed in all three data frames. We 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='inner')
)
```

The result is a data frame whose index contains 275 distinct values, from April 1986 through December 2021. With all 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 gives 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 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 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 both 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. The first is between `oil` and `icecream` (figure 11.15):

```
df.plot.scatter(x='oil', y='icecream')
```

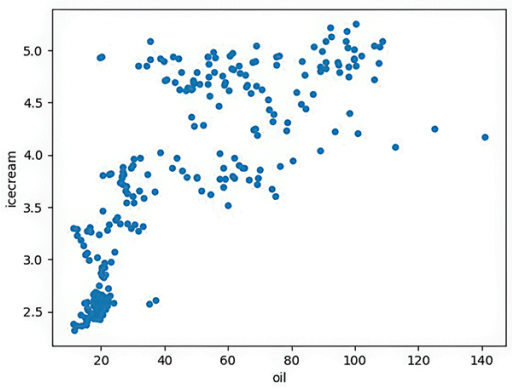


Figure 11.15 Scatter plot comparing oil prices and ice cream consumption

The second scatter plot I asked you to make is between `oil` and `miles` (figure 11.16):

```
df.plot.scFigure 11.16  atter(x='oil', y='miles')
```

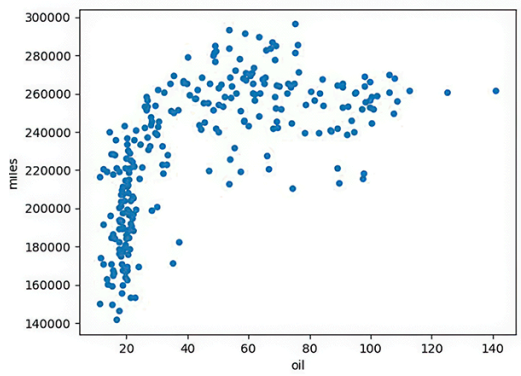


Figure 11.16 Scatter plot comparing oil prices and miles driven

Although you may be able to identify from these scatter plots whether there is a positive correlation here, I think the output from `corr()` gives a much clearer indication of the strength of that correlation.

Finally, I asked you to create a single scatter matrix plot, showing all numeric columns plotting against one another (figure 11.17):

```
from pandas.plotting import scatter_matrix
scatter_matrix(df)
```

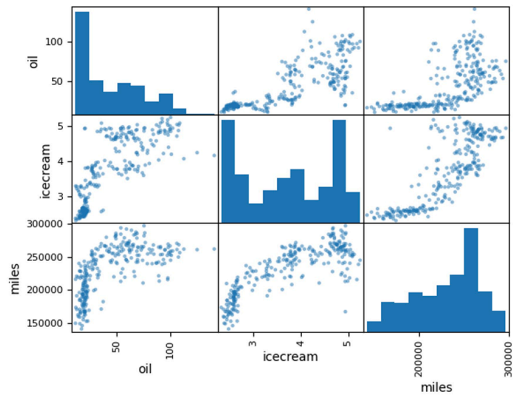


Figure 11.17 Scatter matrix

The scatter matrix is a great way to get a quick look at all the correlations in a 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.

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
    .loc[ice_cream_df['icecream']
         .str.contains(r'\d', regex=True)]        ④
    .astype(np.float64)                           ⑤
)

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='inner')                    ⑥
)

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
```

① Ignores the header row because we are naming the columns

② Sets the index based on column 0

③ Names the two columns in the file

④ Uses a regular expression to exclude rows lacking even one digit

⑤ Sets the dtype to be np.float64

⑥ Performs two inner joins, creating a single data frame with three columns

⑦ Gets a correlation matrix comparing all columns

⑧ Creates a scatter plot of oil vs. ice cream

⑨ Creates scatter plots of all possible combinations

Beyond the exercise

- Is the month correlated with any of these three values?
- Create a scatter plot of `icecream` versus `miles`.
- Instead of using an inner join, you could remove all rows from `oil_df` that weren't on the final day of the month. How could you do that?

Seaborn

Matplotlib is without a doubt the leading plotting system for Python. Many people find it hard to learn and use, however, which has led to the creation of several alternatives. One of the best-known, Seaborn (**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. The pandas API tries to simplify things, papering over much of the configuration that needs to happen 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 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 may be trying to draw from our plots. We can choose from four 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`.

To explore this more fully, let's load the temperature and precipitation data from our weather CSV files:

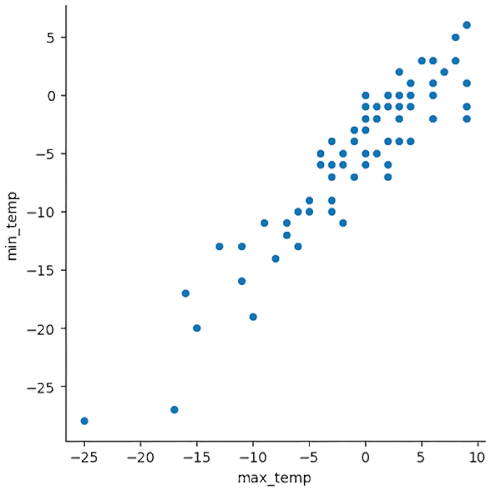
```
import glob

all_dfs = []

all_filenames = glob.glob('../data/*,*.csv')

for one_filename in all_filenames:
    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)
```

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 versus 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 is used for the *x* axis.
- `y` indicates which column from our data frame is used for the *y* axis.
- `data` is a data frame containing both of those columns.



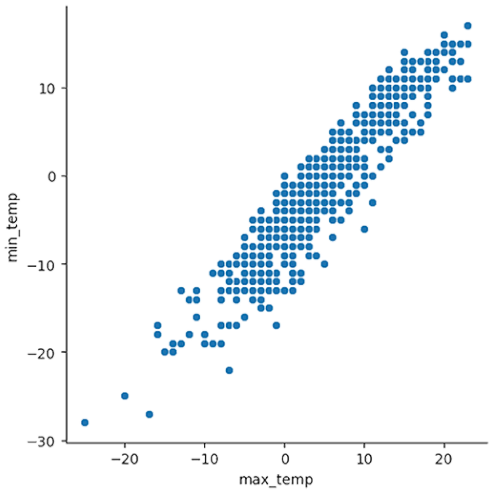
Scatter plot with Chicago weather

In this case, we provide only a subset of the data from `df` so we only see Chicago weather. But what if we want to see all the data from all 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. We've mixed all the weather reports from all the cities! Fortunately, Seaborn provides a

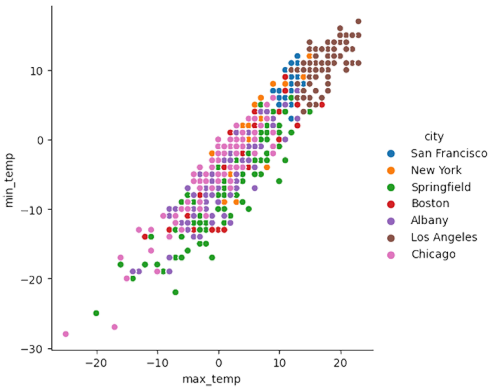
number of different ways to make the data more useful and interesting.



Scatter plot with all cities

For example, we can ask Seaborn to use a different color for each city by passing a column name to the `hue` keyword argument:

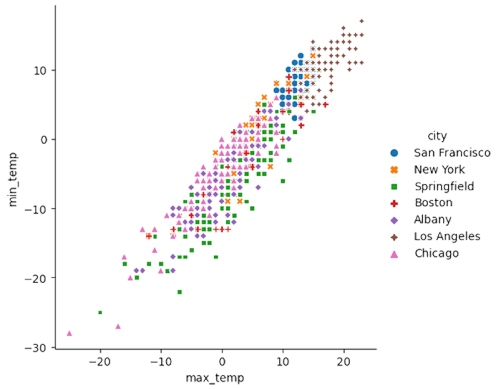
```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df,
            hue='city')
```



Scatter plot with all cities, with a different hue per city

We can have each city's dots use a different marker, as well, by giving the same `city` argument to the `style` parameter:

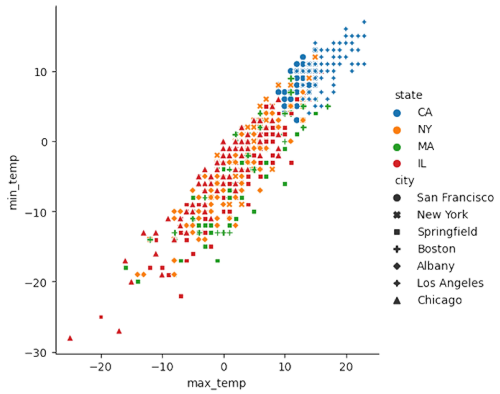
```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df,
            hue='city', style='city')
```



Scatter plot with all cities, with a different hue and marker per 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')
```

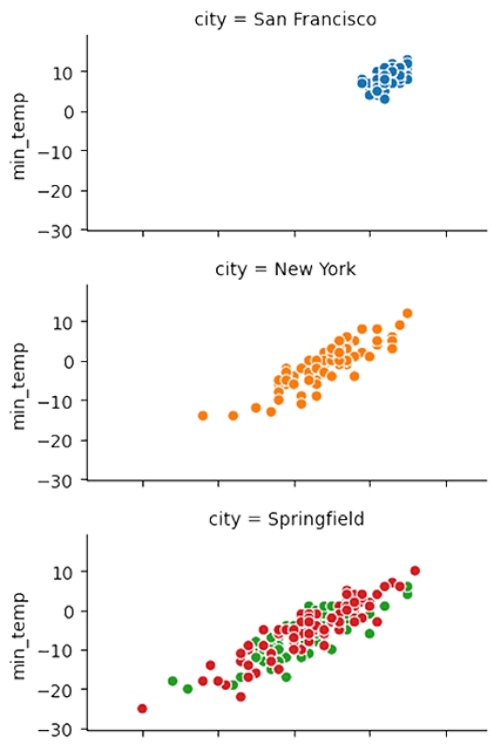


Scatter plot with all cities, with a different hue per state and marker per city

However, it's messy to see all 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')
```

Although 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:



Put each city on a different row.

```
sns.relplot(x='max_temp',
            y='min_temp',
            data=df,
            hue='state', kind='line')
```

This call is fine, *except* 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 they could not handle my data frame as it stood, because its index contained nonunique values.

We can fix this easily with `reset_index`:

```
df = df.reset_index(drop=True)
```

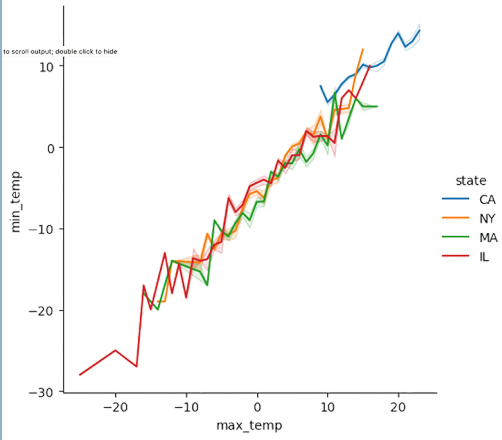
Note that we pass `drop=True` to avoid having the old index added

as a column to the data frame. We're happy to throw out the old index and replace it with a new one, so we 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')
```

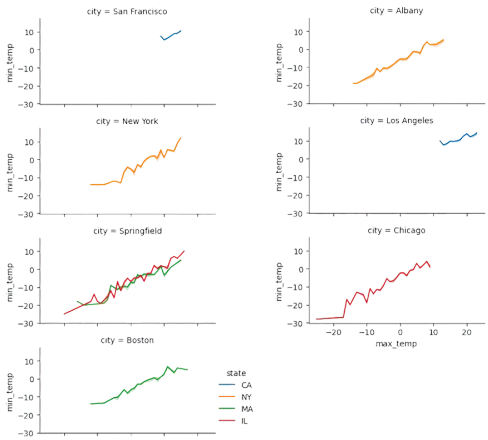
The good news is that we see all the values, and thanks to our value for `hue`, we have a different-colored line for each state. The bad news is that two of the cities in our data set are from the same state. And besides, it's hard to read this plot, with all the data squashed.



Line plot with temperatures per city

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')
```

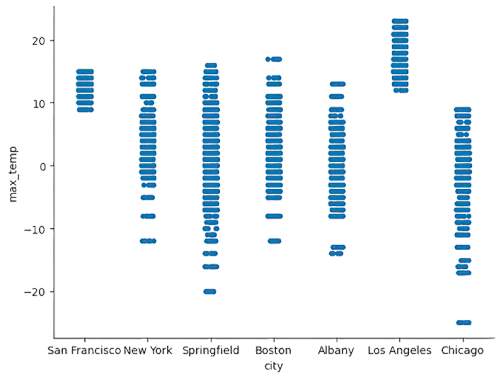


Line plot with temperatures per city, one city per row

Seaborn supports a wide variety of other plots, as well. For example, what if we want to see all the values of `max_temp` for a given city? You can think of this as a set of one-dimensional scatter plots:

```
sns.catplot(x='city', y='max_temp', data=df)
```

Notice how the *x* axis is for the categories, whereas the *y* axis describes which value we're seeing. This plots each of the values in the data set.

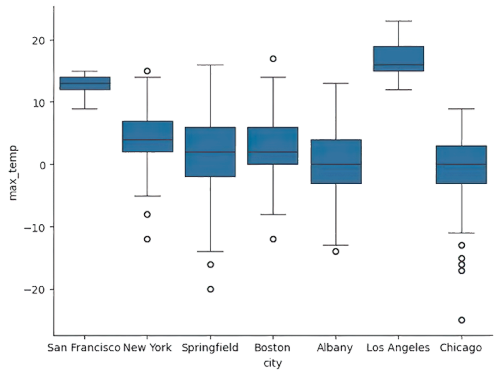


Plotting all temperatures for a city

If we instead want to summarize our data, we can ask for a boxplot, instead:

```
sns.catplot(x='city', y='max_temp', data=df, kind='box')
```

This shows a boxplot for each of the cities' values of `max_temp`, all side by side on the same *y* axis.

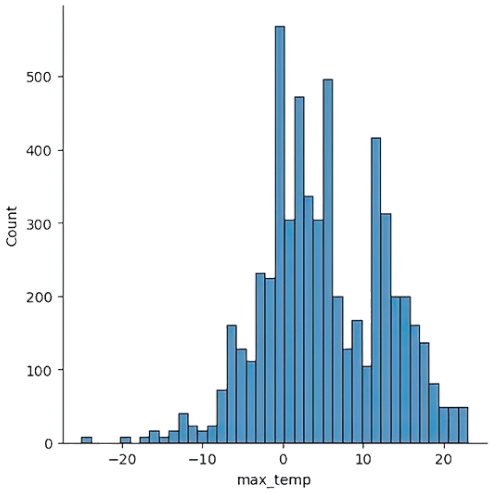


Boxplot for `max_temp` in each city

Finally, Seaborn offers 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)
```

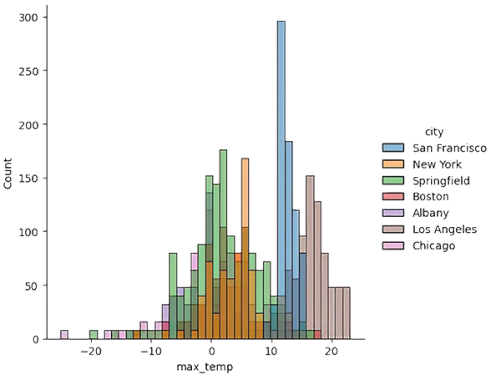
This, of course, shows the distribution of all values of `max_temp`.



Histogram of `max_temp` in all cities

We can also give each city its own colored bars by setting `hue`:

```
sns.displot(x='max_temp', data=df, hue='city')
```

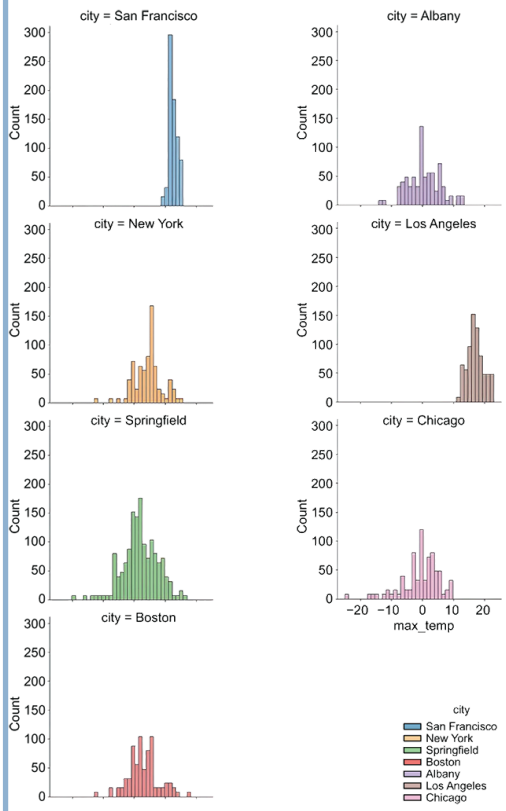


Histogram of `max_temp` in all cities, each city in a different hue

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')
```



Histogram of `max_temp` in all cities, each city in a different hue and in a separate plot

These are just some of Seaborn's many capabilities. If you're interested in seeing everything Seaborn can do, I strongly recommend checking out the documentation at **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.

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

1. Load data from NYC taxis in 2020 (i.e., both **nyc_taxi_2020-01.csv** and **nyc_taxi_2020-07.csv**), only loading the columns `tpep_pickup_datetime`, `passenger_ count`, `trip_distance`, and `total_amount`.
2. Add `month` and `year` columns from `tpep_pickup_datetime`. Keep only those data points in which the year is 2020 and the month is either January or July.
3. Set a new numeric range index numbered starting at 0.
4. Assign `df` to a random sample of 1% of the elements in the original `df`.
5. 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.

6. Determine why there are colors for `passenger_count` values of 1.5, 4.5, and 7.5.
7. 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 is the average trip distance. There should be two lines, one for each month.
8. 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.
9. Using Seaborn, create a boxplot of `total_amount` with one plot for each month.

Working it out

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 start 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_2020-07.csv']

all_dfs = [pd.read_csv(one_filename,
           usecols=['tpep_pickup_datetime',
                    'passenger_count',
                    'trip_distance',
                    'total_amount'],
```

```
            parse_dates=['tpep_pickup_datetime'])
            for one_filename in filenames]

  df = pd.concat(all_dfs)
```

Notice that we once again use `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 creates a list of data frames using a list comprehension. The list is passed to `pd.concat`, which returns a new data frame that combines all 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 asked you to ensure that all the data we look at is from January or July 2020. As we've seen, the taxi data is "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 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. We can check whether a data frame's index contains repeated values with the code:

```
df.index.is_unique
```

If this returns `True`, the values are already unique. If not, 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 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 begin to plot our data. But the data set is 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% of the original `df`'s values and 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% of our original data:

```
sns.relplot(x='trip_distance',
            y='total_amount',
            data=df,
            hue='passenger_count')
```

The `relplot` function shows relationships among numeric columns, and the default way to do that is with a scatter plot. Here, we tell `relplot` the following:

- The *x* axis should use values from the `trip_distance` column.
- The *y* axis should use values from the `total_amount` column.
- We use `df` as our data frame.

- We use `passenger_count` as the basis for coloring the lines and dots.

Sure enough, this works, giving us a nice scatter plot (figure 11.18).

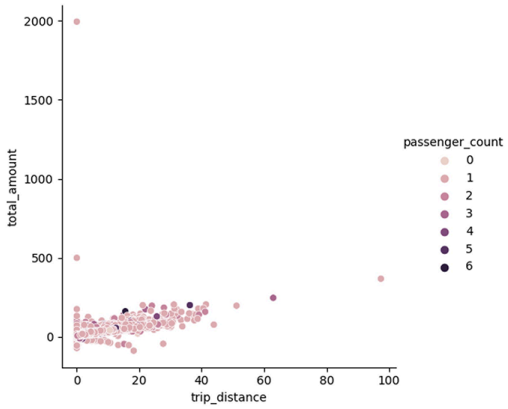


Figure 11.18 Scatter plot comparing `trip_distance` with `total_amount`

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. As before, we use `relplot` to get that plot:

- The *x* axis is from the `day` column.
- The *y* axis is 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 use separate colors for each value of `month`, we are able to plot two different lines on the same chart.

Notice, though, that there are gray lines around each plot. 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` (figure 11.19):

```
sns.relplot(x='day', y='trip_distance', kind='line',
            data=df, hue='month', ci=None)
```

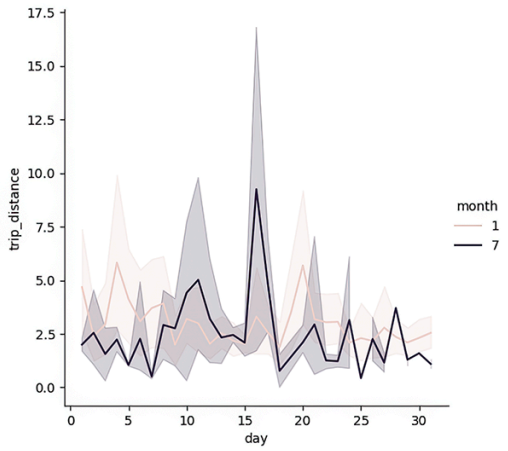


Figure 11.19 Line plot showing `trip_distance` per day, separated by month

Next, I asked you to show the number of trips taken on each day

of these months. This requires another line plot:

- The *x* axis is the day of each month.
- The *y* axis reflects 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? To do that, we need to use the `count` aggregation method. And indeed, here I suggest getting data back not 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 have access to `month` and `day` and also to the number of rides per day. (Using `year` is weird because we aren't counting the `year`—but we need to pick a column.) After performing this `groupby`, we 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()
            .reset_index(), ci=None)
```

This is a complex query, and it's used in a complex plot. So let's walk through it a step at a time:

1. We want to know how many rides there were on each day of each month. That requires `groupby(['month', 'day'])`.
2. We run the `count` aggregation method on the `groupby` object.
3. The result gives us a count for each remaining column in the data frame. We only need one, and we choose `year`.
4. We 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.
5. We pass the result from `reset_index` as the argument to `data` in our call to `relplot`.
6. We tell `relplot` that the *x* axis should be based on `day` and the *y* axis should be based on `year`, the count of rides.
7. We tell `relplot` to distinguish between months by color.
8. We ask for a line plot.
9. We ask for `ci='None'` to avoid showing any confidence intervals.

We then see, rather dramatically (figure 11.20), that there were fewer rides per day in July (in the middle of the pandemic) than there were in January (before it started).

Finally, we ask to see a boxplot of the `total_amount` column, separated by month. Boxplots 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 is the categories we're comparing: `month`.
- The *y* axis is the values we want to see graphically: `total_amount`.
- We're looking at data from `df`.
- We want to see a boxplot and thus specify `kind='box'`.

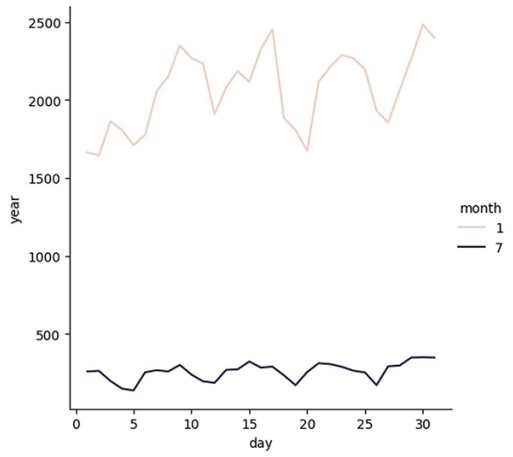


Figure 11.20 Line plot showing `trip_distance` per day, separated by month

The code is as follows (figure 11.21)

```
sns.catplot(x='month', y='total_amount', data=df, kind='box')
```

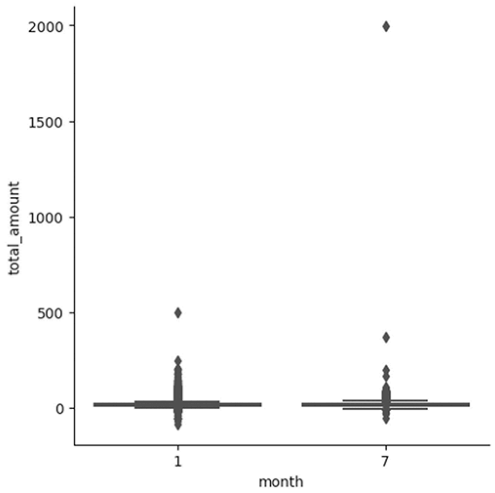


Figure 11.21 Boxplot for `total_amount` per month

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()
```

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',
               'trip_distance', '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()
            .reset_index(), ci=None)

sns.catplot(x='month', y='total_amount', data=df, kind='box')
```

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.
- Add a `trip_length` column for short, medium, and long trips, as we did in exercise 7. Show the trip distance per day of month in three plots alongside one another, with one for each category.

- Create a bar plot showing how many rides take 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.

ummary

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 saw a number of the ways pandas can perform visualizations using a simplified API to Matplotlib. We also saw, in the final exercise, how the Seaborn package can create attractive plots using data frames with its own separate API on top of Matplotlib.