5 Cleaning data

In the late 1980s, my employer wanted to know how much rain had fallen in various places. Their solution? They gave me a list of cities and phone numbers and asked me to call each in sequence, recording the previous day's rainfall in an Excel spreadsheet. Nowadays, getting that sort of information—and many other types—is pretty easy. Many governments provide data sets for free, and numerous companies make data available for a price. No matter what topic you're researching, data is almost certainly available. The only questions are where you can get it, how much it costs, and what format it comes in.

You should ask another question, too: how accurate is the data you're using? It's easy to assume that a CSV file from an official-looking website contains good data. But all too often, it will have problems. That shouldn't surprise us, given that the data comes from people (who can make mistakes) and machines (which make different types of mistakes). Maybe someone accidentally misnamed a file or entered data into the wrong field. Maybe the automatic sensors whose inputs

were used in collecting the data were broken or offline. Maybe the servers were down for a day, or someone misconfigured the XML feed-reading system, or the routers were being rebooted, or a backhoe cut the internet line.

All this assumes there was data to begin with. Often, we'll have missing data because there wasn't any data to record.

This is why I've heard data scientists say that 80% of their job involves cleaning data. What does it mean to "clean data"? Here is a partial list:

- Rename columns.
- Rename the index.
- Remove irrelevant columns.
- Split one column into two.
- Combine two or more columns into one.
- Remove nondata rows.
- Remove repeated rows.
- Remove rows with missing data (aka `NaN`).
- Replace `NaN` data with a single value.
- Replace `NaN` data via interpolation.
- Standardize strings.
- Fix typos in strings.
- Remove whitespace from strings.
- Correct the types used for columns.
- Identify and remove outliers.

We have discussed some of these techniques in previous chapters. But the importance of cleaning your data, and thus ensuring that your analysis is as accurate as possible, cannot be overstated.

In this chapter, we'll look at pandas techniques for cleaning data. We'll see a few ways to handle `NaN` values. We'll consider how to preserve as much data as possible, even when it's pretty dirty. We'll discuss how to better understand our data and its limitations. And we'll look at a some more advanced techniques for massaging data into a form that's more easily analyzed.

Table 5.1 What you need to know

Concept	What is it?	Example	To learn more
`df.shape`	A two-element tuple indicating the number of rows and columns in a data frame	`df.shape`	http://mng.bz/8rpg
`len(df)` or `len(df.index)`	Gets the number of rows in a data frame	`len(df)` or `len(df.index)`	http://mng.bz/EQdr
`s.isnull`	Returns a boolean series indicating where there are null (typically `NaN`) values in the series `s`	`s.isnull()`	http://mng.bz/N2KX
`s.notnull`	Returns a boolean series indicating where there are non-null	`s.notnull()`	http://mng.bz/D420

	values in the series `s`		
`df.isnull`	Returns a boolean data frame indicating where there are null (typically `NaN`) values in the data frame `df`	`df.isnull()`	http://mng.bz/lWGz
`df.replace`	Replaces values in one or more columns with other values	`df.replace('a':{'b':'c'}, 'd')`	http://mng.bz/Bm2q
`s.map`	Applies a function to each element of a series, returning the result of that application on each element	`s.map(lambda x: x**2)`	http://mng.bz/d1yz
`df.fillna`	Replaces `NaN` with	`df.fillna(10)`	http://mng.bz/rWrE

	other values		
`df.dropna`	Removes rows with `NaN` values	`df = df.dropna()`	http://mng.bz/V1gr
`s.str`	Works with textual data	`df['colname'].str`	http://mng.bz/x4Wq
`str.isdigit`	Returns a boolean series, indicating which strings contain only the digits 0–9	`df['colname'].str.isdigit()`	http://mng.bz/AoAE
`pd.to_numeric`	Returns a series of integers or floats based on a series of strings	`pd.to_numeric(df['colname'])`	http://mng.bz/Zq2j
`df.sort_index`	Reorders the rows of a data frame based on the values in its index in ascending order	`df = df.sort_index()`	http://mng.bz/RxAn

`pd.read_excel`	Creates a data frame based on an Excel spreadsheet	`df = pd.read_excel('myfile.xlsx')`	http://mng.bz/2DXN
`pd.read_csv`	Returns a new data frame based on CSV input	`df = pd.read_csv('myfile.csv')`	http://mng.bz/wvl7
`s.value_counts`	Returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	http://mng.bz/1qzZ
`s.unique`	Returns a series with the unique (i.e., distinct) values in `s`, including `NaN` (if it occurs in `s`)	`s.unique()`	http://mng.bz/PzA2
`s.mode`	Returns a series with the most	`s.mode()`	http://mng.bz/7vBm

commonly found values in `s`

How much is missing?

We've already seen, on several occasions, that data frames (and series) can contain `NaN` values. One question we often want to answer is, how many `NaN` values are in a given column? Or, for that matter, in a data frame?

One solution is to calculate things yourself. There is a `count` method you can run on a series, which returns the number of non-null values in the series. That, combined with the shape of the series, can tell you how many `NaN` values there are:

```
s.shape[0] - s.count()    ①
```

① Returns an integer: the number of null elements

This is tedious and annoying. And besides, shouldn't pandas provide a way to do this? Indeed it does, in the form of the `isnull` method. If you call `isnull` on a column, it returns a boolean series with `True` where there is a `NaN` value and `False` in other places. You can then apply the `sum` method to the series, which will return the number of `True` values, thanks to the fact that

Python's boolean values inherit from integers and can be in place of 1 (`True`) and 0 (`False`) if needed:

```
s.isnull().sum()     ①
```

① Calculates the numberof NaN values in s

If you run `isnull` on a data frame, you get a new data frame back, with `True` and `False` values indicating whether there is a null value in that particular row-column combination. And, of course, you can run `sum` on the resulting data frame to find out how many `NaN` values are in each column:

```
df.isnull().sum()     ①
```

① Calculates the number of NaN values in each column

Finally, the `df.info` method returns a wealth of information about the data frame on which it's run, including the name and type of each column, a summary of the number of columns of each type, and the estimated memory usage. (We'll talk more about this memory usage in Chapter 12.) If the data frame is small enough, it will also show you how many null values are in each column. However, this calculation can take some time. Thus, `df.info` will only count null values below a certain threshold. If

you're above that threshold (the `pd.options.display.max_info_columns` option), you need to tell pandas explicitly to count by passing `show_counts=True`:

```
df.info(show_counts=True)  ①
```

① Gets full information about the data frame df, including the number of null values in each column

NOTE Pandas defines both `isna` and `isnull` for both series and data frames. What's the difference between them? There is *no* Difference. If you look at the pandas documentation, you'll find that they're identical except for the name of the method being called. In this book, I use `isnull`, but if you prefer to go with `isna`, be my guest. Note that both are different from `np.isnan`, a method defined in NumPy, on top of which pandas is defined. I try to stick with the methods that pandas defines, which in my experience integrate better into the rest of the system. Rather than using `~`, which pandas uses to invert boolean series and data frames, you can often use the `notnull` methods for both series and data frame.

Exercise 25 • Parking cleanup

In chapter 4, we looked at parking tickets given in New York City in

2020. We were able to analyze that data and draw some interesting conclusions from it. But let's consider that this data was entered by a police officer, a parking inspector, or another person, which means there is a good chance it sometimes has missing or incorrect data. That may seem like a minor problem, but it can mean everything from cars being ticketed incorrectly to bad statistics in the system to people getting out of fines due to incorrect information. (A side note: when you're issued a parking ticket in Israel, it includes a photograph of your car and license plate, taken by the inspector when they issued the ticket. That makes it a bit harder to wriggle out of fines, but people manage to do it anyway.)

In this exercise, we will identify missing values, one of the most common problems you will encounter. We'll see how often values are missing and what effect they may have. Note that for this exercise, we're going to assume that a parking ticket that is missing data may be dismissed; don't blame me if this defense doesn't work when appealing any tickets you get in New York.

I want you to do the following:

1. Create a data frame from the file **nyc-parking-violations-2020.csv**. We are only interested in a handful of the columns:
    1. `Plate ID`
    2. `Registration State`
    3. `Vehicle Make`
    4. `Vehicle Color`
    5. `Violation Time`
    6. `Street Name`

    How many rows are in the data frame when it is read into memory?
2. Remove rows with any missing data (i.e., a `NaN` value). How many rows remain after doing this pruning? If each parking ticket brings $100 into the city, and missing data means the ticket can be successfully contested, how much money may New York City lose due to such missing data?
3. Let's instead assume that a ticket can only be dismissed if the license plate, state, car make, and/or street name are missing. Remove rows that are missing one or more of these. How many rows remain? Assuming $100/ticket, how much money would the city lose as a result of this missing data?

4. Now let's assume that tickets can be dismissed if the license plate, state, and/or street name are missing—that is, the same as the previous question, but without requiring the make of car. Remove rows that are missing one or more of these. How many rows remain? Assuming $100/ticket, how much money would the city lose as a result of this missing data?

Working it out

When you're first starting with data analytics, it's reasonable to think you can just toss out imperfect data. After all, if something is missing, you cannot use it, right? In this exercise, I hope you saw not only how to remove rows with missing data but also the potential problems associated with doing that.

For starters, let's load the CSV file into a data frame. We are only interested in a few columns, so loading looks like this:

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

df = pd.read_csv(filename,
                 usecols=['Plate ID',
                          'Registration State',
                          'Vehicle Make',
                          'Vehicle Color',
                          'Violation Time',
                          'Street Name'])
```

We can determine the number of rows in our data frame by getting the first element (i.e., index 0) from the `shape` attribute:

```
df.shape[0]
```

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

```
len(df)
```

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

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

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

Counting values

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

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

If you want to know the number of all values, including `NaN`, you can use the `size` attribute (not a method), which works on both series and data frames. Or invoke `np.size` on the series, as in `np.size(s)`.

However, as I said earlier, I prefer to call `len(df.index)`, which gives me the total length and seems to run fastest.

With that data frame in place, we can start to make a few queries, looking for tickets that could potentially be dismissed for lack of data. Our first query will apply the naive (but well-meaning) approach, in which we remove any rows that have missing data. We can do this with the `df.dropna` method. That method returns a new data frame, identical to our original `df`, but without any rows that have any `NaN` values (figure 5.1):

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

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

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

This means, by the way, that if every row in a data frame contains a single `NaN` value, the result of calling `df.dropna` will be an empty data frame. Its columns will be identical to your existing data frame but have zero rows (figure 5.2).

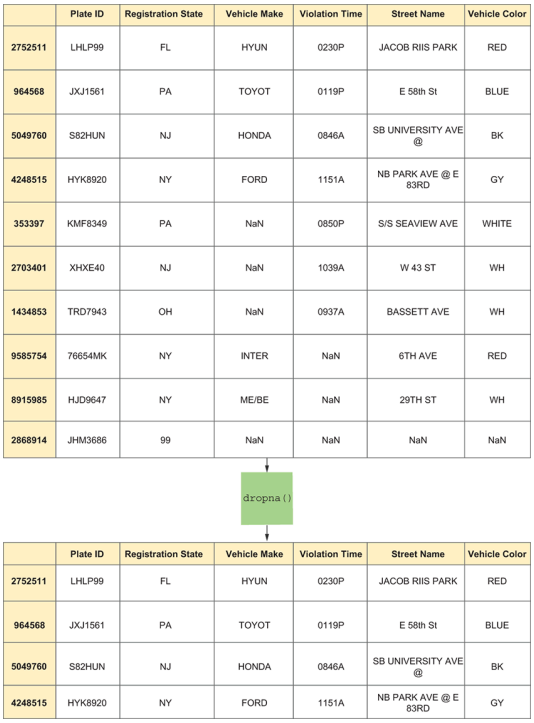


Figure 5.2 Running `dropna` on a data frame removes all `NaN` values and the rows containing them.

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

calculate that:

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

We get a large number as a result: 447,359. That represents about 3.5% of the data in the original data frame—which doesn't sound like much until we consider the next question: how much money New York City would lose if all those tickets were thrown out. Assuming that each parking ticket costs $100, we can calculate the total as follows:

```
(len(df.index) - len(all_good_df.index) ) * 100
```

That works out to a shockingly high number: $44.7 million. I decided to display this result as a string, taking advantage of the fact that Python's f-strings have a special `,` format code that, when put after `:` on an integer, puts commas before every three digits:

```
f'${(len(df.index) - len(all_good_df.index) ) * 100:,}'
```

As we can see in this (somewhat contrived) example, removing bad data can give us a better sense of confidence—but even when we remove a small amount (3.5%!), it can add up very quickly.

Next, I asked you to apply a slightly lighter standard, removing rows only if they have `NaN` in one of four

columns: `Plate ID`, `Registration State`, `Vehicle Make`, or `Street Name`. But this raises another question: how can we select only particular columns?

One approach is to remember that each column is a series, and we can apply `notnull` to that series, giving us a boolean series. We can combine those four series with `&`, giving us a boolean series in which `True` indicates that all values are non-null. Finally, we can apply that boolean series to our original `df`, giving us a data frame in which most (but not all) data is non-null:

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

This works. But there's a better way to do things, using `dropna`. Normally, as we just saw, `dropna` removes rows that contain any `NaN` values. But we can tell it to look in only a subset of the columns, ignoring `NaN` values in any other columns. The result is a much cleaner query (figure 5.3):

```
semi_good_df = df.dropna(subset=['Plate ID',
                                 'Registration State',
                                 'Vehicle Make',
                                 'Street Name'])
```

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

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

	Plate ID	Registration State	Vehicle Make	Violation Time	Street Name	Vehicle Color
2752511	LHLP99	FL	HYUN	0230P	JACOB RIIS PARK	RED
964568	JXJ1561	PA	TOYOT	0119P	E 58th St	BLUE
5049760	S82HUN	NJ	HONDA	0846A	SB UNIVERSITY AVE @	BK
4248515	HYK8920	NY	FORD	1151A	NB PARK AVE @ E 83RD	GY
353397	KMF8349	PA	NaN	0850P	S/S SEAVIEW AVE	WHITE
2703401	XHXE40	NJ	NaN	1039A	W 43 ST	WH

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

Using “thresh” with “dropna”

In this case, we want to ensure that all four columns have non-`NaN` values. However, passing an integer to the `thresh` keyword argument while we’re also passing a list of columns to `subset` allows us to indicate that only some of these columns must be non-`NaN`. For example, if we’re OK with any three of these four columns having non-`NaN` in them, we can say

```
semi_good_df = df.dropna(subset=['Plate ID',
                                 'Registration State',
                                 'Vehicle Make',
                                 'Street Name'],
                         thresh=3)
```

Of course, this means we’ll still have some `NaN` values in the resulting

data frame. But often that is a reasonable trade-off.

How many rows did we remove? And how much money may New York give up if we only remove these rows?

```
f'${(len(df.index) - len(semi_good_df.index) ) * 100:,}
```

According to this calculation, the result is $6,378,500. Still a fair amount of money, but a far cry from what we would have lost had we removed any and all problematic records.

But let's make the rules looser still, mandating only that three of the columns lack `NaN` values: `Plate ID`, `Registration State`, and `Street Name`. Once again, we can use `df.dropna` along with its `subset` parameter to remove only those rows that lack all three of these columns:

```
loosest_df = df.dropna(subset=['Plate ID',
                               'Registration State',
                               'Street Name'])
```

This removes only 1,618 rows from our original data frame. How much money would that translate into?

```
f'${(len(df.index) - len(loosest_df.index) ) * 100:,}
```

According to this calculation, it works out to $161,800, which seems like a far more reasonable amount of lost revenue.

Solution

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

df = pd.read_csv(filename,
                 usecols=['Plate ID',
                          'Registration State',
                          'Vehicle Make',
                          'Vehicle Color',
                          'Violation Time',
                          'Street Name'])                    ①

all_good_df = df.dropna()                                    ②
len(df.index) - len(all_good_df.index)                       ③
f'${(len(df.index) - len(all_good_df.index) ) * 100:,}'      ④

semi_good_df = df.dropna(subset=['Plate ID',
                                 'Registration State',
                                 'Vehicle Make',
                                 'Street Name'])             ⑤
len(df.index) - len(semi_good_df.index)                      ⑥
f'${(len(df.index) - len(semi_good_df.index) ) * 100:,}'     ⑦

loosest_df = df.dropna(subset=['Plate ID',
                               'Registration State',
                               'Street Name'])               ⑧
len(df.index) - len(loosest_df.index)                        ⑨
f'${(len(df.index) - len(loosest_df.index) ) * 100:,}'       ⑩
```

① Reads the CSV file using only a handful of columns

② Removes rows containing any NaN values

③ How many rows did we remove?

④ Uses an f-string to display potentially lost revenue with commas

⑤ Drops rows with NaN in any of four columns

⑥ How many rows did we remove now?

⑦ Uses an f-string to display potentially lost revenue with commas

⑧ Drops rows with NaN in any of three columns

⑨ How many rows did we remove this time?

⑩ Uses an f-string to display potentially lost revenue with commas

You can explore a version of this in the Pandas Tutor at **http://mng.bz/6nlo**.

Beyond the exercise

- So far, you have specified which columns must be all non-null. But sometimes it's OK for some columns to have null values, as long as it's not too many. How many rows would you eliminate if you required at least three non-null values from the four columns `Plate ID`, `Registration State`, `Vehicle Make`, and `Street Name`?

- Which of the columns you've imported has the greatest number of `NaN` values? Is this a problem?
- Null data is bad, but there is plenty of bad non-null data, too. For example, many cars with `BLANKPLATE` as a plate ID were ticketed. Turn these into `NaN` values, and rerun the previous query.

Combining and splitting columns

A common aspect of data cleaning involves creating one new column from several existing columns, as well as the reverse—creating multiple columns from a single existing column. For example, back in exercise 8, we saw how to create a new column, `current_net`, by calculating the net price of each product and then multiplying that by the quantity sold:

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

This may not seem like "cleaning" to you, but it's a common way to make data clearer and easier to understand. Plus, we can then identify holes and problems in our data and fix them accordingly.

I'll add something I often told my children when they were studying math in school: a large part of

mathematics involves finding ways to rewrite problems so they're easier to understand and then solve. The same is true about data structures in programming. And it's also true in data science, where having columns that are clearer and more easily understood can help clarify our analysis.

Perhaps even more frequently, though, cleaning data involves turning one complex column into one or more simpler columns. For example, you can imagine taking a column with a `float64` dtype and turning it into two `int64` columns, one with the integer portion and one with the floating-point portion.

This is especially true in the case of two complex data structures, which we'll have much more to say about in chapters 9 and 10. Let's look at a particularly common example: when we have string data and want to grab certain substrings from within that data. In a normal Python program, we would use a slice to retrieve a substring. For example:

```
s = '00:11:22'
print(s[3:5])   # prints '11'
```

Remember that Python slices are always of the form `[start:end+1]`. So if we want the characters at index 3 and index 4, we ask for

`3:5`, which means "starting at 3, up to and not including 5."

Let's assume that `s` isn't a single string but a series containing strings. To retrieve the slice `3:5` from each of those strings, we can use the `str` accessor on the series followed by the `slice` method. The syntax is a bit different than with Python strings, but it should still feel familiar:

```
s.str.slice(3,5)
```

The result of this code is a new series of string objects, of same length as `s`, containing two-element strings taken from indexes 3 and 4 of each row in `s`.

It's common to slice and dice the columns of a data frame this way, retrieving only those parts that are of interest to us. This makes the problem easier to see, understand, and solve and allows us to remove the original (larger) column, saving memory and improving computation speed.

Exercise 26 • Celebrity deaths

Sometimes, as in the previous exercise, only a small fraction of the data is unreadable, missing, or corrupt. In other cases, a much larger proportion is problematic—and if you want to use the data set,

you'll need to not only remove bad data but also massage and salvage the good data.

For this exercise, we'll look at a (slightly morbid) data set: a list of celebrities who died in 2016 and whose passing was recorded in Wikipedia, including the date of death, a short biography, and the cause of death. The problem is that this data set is messy, with some missing data and some erroneous data that will prevent us from working with it easily.

The goal of this exercise is to find the average age of celebrities who died February–July 2016. Getting there will take several steps:

1. Create a data frame from the file **celebrity_deaths_2016.csv**. For this exercise, we'll use only two columns:
    1. `dateofdeath`
    2. `age`
2. Create a new `month` column containing the month from the `dateofdeath` column.
3. Make the `month` column the index of the data frame.
4. Sort the data frame by the index.
5. Clean all nonintegers from the `age` column.
6. Turn the `age` column into an integer value.
7. Find the average age of celebrities who died during that period.

Finding numeric strings

Normally, we can turn a string column into an integer column with

```
df['colname'] = df['colname'].astype(np.int64)
```

However, this will fail if any of the rows in `df['colname']` cannot be turned into integers. That's because the strings either are empty or contain nondigit characters.

We can determine which rows in a column can be successfully turned into integers by applying the `isdigit` method via the `str` accessor:

```
df['colname'].str.isdigit()
```

This returns a boolean series in which `True` values correspond with `NaN` in `df['colname']` and `False` values correspond to non-`NaN` values in `df['colname']`. This boolean series can then be applied as a mask index to the original column. This technique comes in handy when working with dirty data—as we are doing here.

Working it out

In this exercise, we create and clean up a two-column data frame. Each column needs to be cleaned differently for us to answer the question I asked: what was the

average age of celebrities who died in February through July?

We start by loading the CSV file into a data frame. We are only interested in two of the columns, so we load the file as follows:

```
filename = '../data/celebrity_deaths_2016.csv'

df = pd.read_csv(filename,
                 usecols=['dateofdeath', 'age'])
```

With that in place, we must tackle our two cleaning tasks.

Because we're only interested in celebrity deaths during particular months, we need to grab the month value from the `dateofdeath` columns. (There are other ways to attack this problem; in chapter 9, we'll discuss a few.) Because `dateofdeath` is a string column, we can use the `slice` method of the `str` accessor to get the months—which happen to be in indexes 5 and 6 of the date string. This means we can retrieve the two-digit month as

```
df['dateofdeath'].str.slice(5,7)
```

and we can assign that value to a new column, `month`, as follows (and figure 5.4):

```
df['month'] = df['dateofdeath'].str.slice(5,7)
```

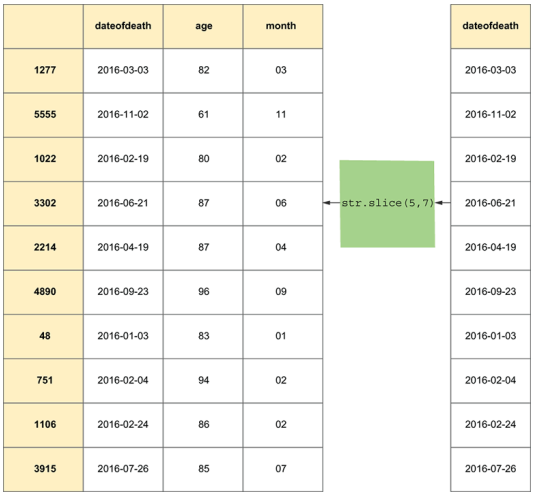


Figure 5.4 Adding a new `month` column to our data frame based on the month in `dateofdeath`

Notice that we aren't turning the column into an integer. We could, but the leading `0` on the two-digit months makes it trickier. Besides, we don't need to do that, and the data set is relatively small, so we don't have to worry about the memory implications.

Now that we have created the `month` column, we want to turn it into the index:

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

I next asked you to sort the data frame by the index, meaning we should sort the rows so the index is in ascending order. We do this because we want to retrieve several rows via a slice, and when an index contains repeated values, it needs to be sorted before we can retrieve slices from it. So let's sort by the index:

```
df = df.sort_index()
```

We are now set to retrieve rows from a single month or a range of months. But we're not done yet, because we want to find the average age at which celebrities died in 2016. To do that, we need to turn the `age` column into a numeric value, most likely an integer. We can try like this:

```
df['age'] = df['age'].astype(np.int64)
```

However, this will fail for two reasons: first, some values contain characters other than digits. Second, some values are `NaN`, which, as floating-point values, cannot be coerced into integers. Before willy-nilly removing the `NaN` values, though, we should probably check to see how many there are. We can do that with the `isnull().sum()` trick we've already seen and combine that with the `shape` method to find the percentage of null values:

```
df['age'].isnull().sum() / len(df['age'])
```

We get the answer 0.004, meaning 0.4% of the values are `NaN`. We can sacrifice that many rows and not worry about how much data we're losing. As a result, we can remove the `NaN` values:

```
df = df.dropna(subset=['age'])
```

Notice that we're again using the `subset` parameter. Not that there are any rows in the index with `NaN` values, but it's always a good idea to be specific, just in case.

How can we remove the rest of the troublesome data, though? That is, how can we remove rows that contain nondigit characters? One way would be to rely on the `str.isdigit` method, which returns `True` if a string contains only digits (and isn't empty). (It returns `False` if there is a `-` sign or decimal point, so it's not a failsafe for finding numbers, but it will work with ages.) We can apply that to `df['age']` as follows:

```
df['age'].str.isdigit()
```

We can then use this boolean series as a mask index to remove rows in `df` whose ages cannot be turned into integers:

```
df = df[df['age'].str.isdigit()]
```

But, as is often the case, pandas has a more elegant solution: the `pd.to_numeric` function. This function—which is defined at the top `pd` level rather than on a series or data frame—tries to create a new series with numeric values. The function attempts to turn the values

into integers, but if it cannot, it returns floats instead:

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

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

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

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

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

Here's the result:

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

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

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

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

Our data frame is now ready for our final calculation, which we've been working up to this entire time:

```
df.loc['02':'07', 'age'].mean()
```

Notice that because our index uses strings, we need to specify the slice with strings from `'02'` to `'07'`. The answer we get is 77.1788.

Solution

```
filename = '../data/celebrity_deaths_2016.csv'

df = pd.read_csv(filename,
                 usecols=['dateofdeath', 'age'])          ①

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

df = df.sort_index()                                      ④

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

① Loads the CSV into a two-column data frame

② Turns month data from dateofdeath into a new column

③ Turns the month column into an index

④ Sorts the data frame by the index

⑤ Removes NaN values in age

⑥ Gets a numeric column from the strings in age

⑦ Gets the mean age from February through July

You can explore a version of this in the Pandas Tutor at **http://mng.bz/or7d**.

Beyond the exercise

- Add a new column, `day`, from the day of the month in which the celebrity died. Then create a multi-index (from `month` and `day`). What was the average age of death from Feb. 15 through July 15?
- The CSV file contains another column, `causeofdeath`. Load it into a data frame, and find the five most common causes of death. Now replace any `NaN` values in that column with the string `'unknown'`, and again find the five most common causes of death.

- If someone asked whether cancer is in the top 10 causes, what would you say? Can you be more specific than that?

Exercise 27 • Titanic interpolation

When our data contains `NaN` values, we have a few options:

- Remove them
- Leave them
- Replace them with something else

What is the right choice? The answer, of course, is “it depends.” If you’re getting data ready to feed into a machine-learning model, you’ll likely need to get rid of the `NaN` values, either by removing those rows or by replacing them with something else. If you’re calculating basic sales information, you may be okay with null values because they aren’t going to affect your numbers too much. And of course, there are many variations on these approaches.

If you choose option 3, “replace them with something else,” that raises another question: what do you want to replace the `NaN` values with? A value you have chosen? Something calculated from the data frame itself? Something calculated on a per-column basis? Any and all of these are appropriate under different circumstances.

In this exercise, we will fill in missing data from the famous Titanic data set: a table of all passengers on that famous, doomed ship. Many of the columns in this file are complete, but some are missing data. It will be up to you to decide whether and how to fill in that missing data. We saw in exercise 13 how to use the `interpolate` method on a data frame to perform this task automatically.

For this exercise, I would like you to do the following:

1. Load the **titanic3.xls** data into a data frame. Note that this file is an Excel spreadsheet, so you won't be able to use `read_csv`. Rather, you'll have to use `read_excel`.
2. Determine which columns contain null values.
3. For each column containing null values, decide whether you will fill it with a value—and if so, with what value, calculated or otherwise.

Unlike many of the exercises in this book, this one has no obvious right or wrong answer. There are, of course, techniques for calculating values—such as the mean and mode for a column—but I hope you'll consider not just how to make such calculations but also why you would do so and when it's most appropriate.

Working it out

This exercise is practical, but it's also philosophical. That's because there often is no "right" answer to the question of what you should do with missing data. As I often tell my corporate training clients, you have to know your data, which means being familiar with it and how it will be analyzed and used. You also may choose incorrectly or discover that a decision you made was appropriate for one type of analysis but not for another type.

That's one reason it's useful to have your work in a Jupyter notebook or a similar, reproducible format. When you need to, you can modify part of the code, keeping the rest intact.

Let's go through each of the steps in this exercise and see what decisions we could make, as well as the actual decision I made. First, I asked you to create a data frame based on the Excel file titanic3.xls. We do this with the `read_excel` method:

```
filename = '../data/titanic3.xls'
df = pd.read_excel(filename)
```

NOTE Like `read_csv`, `read_excel` is a method we run on `pd` rather than on an individual data frame object. That's because we're not trying to modify an existing data frame but rather to create a new one. Also like `read_csv`, the `read_excel` method

has `index_col`, `usecols`, and `names` parameters, allowing us to specify which columns should be used for the data frame, what they should be called, and whether one or more should be used as the data frame's index.

Now that we have created our data frame, we should check for null values. We do that two different ways. First, we use `isnull.sum()` to find out how many `NaN` values are in each column of the data frame. We can then check to see which columns have a nonzero number of `NaN` values. This returns a boolean series, which we can then apply as a mask index to `df.columns`:

```
df.columns[df.isnull().sum() > 0 ]
```

We get the following result:

```
Index(['age', 'fare', 'cabin', 'embarked',
       'boat', 'body', 'home.dest'],
    dtype='object')
```

Notice that the column names are stored in an `Index` object, which works similarly to a series object.

We can also run `df.isnull().sum()` by itself to see how many `NaN` values are in each column:

```
df.isnull().sum()
```

We get the following result (figure 5.5):

```
pclass          0
survived        0
name            0
sex             0
age           263
sibsp           0
parch           0
ticket          0
fare            1
cabin        1014
embarked        2
boat          823
body         1188
home.dest     564
dtype: int64
```

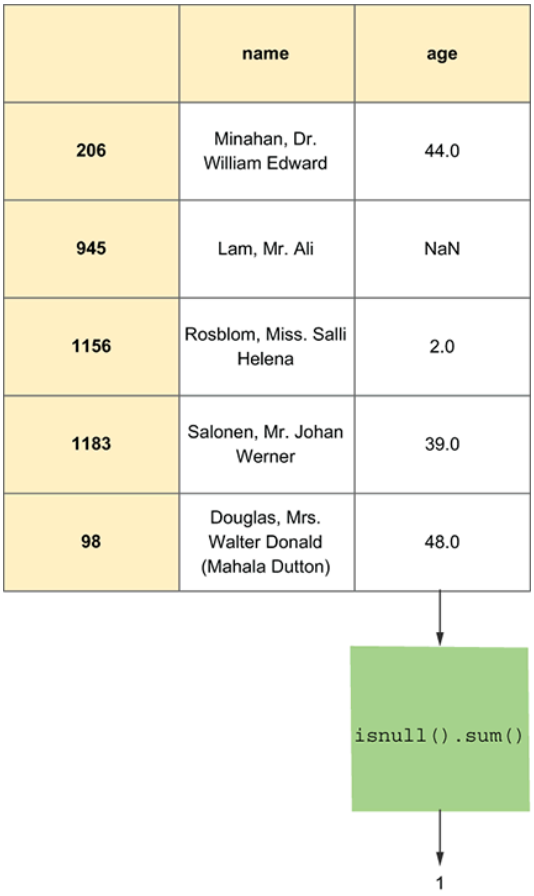
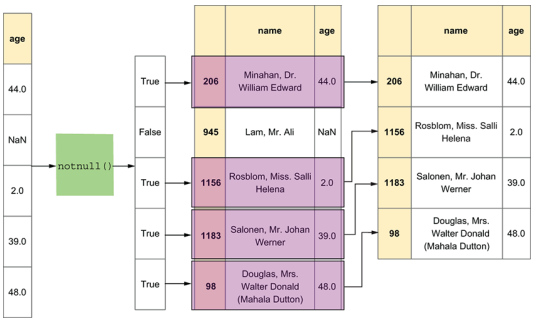


Figure 5.5 Finding the number of `NaN` values in a column by summing the result of `isnull()`

Deciding what to do with each `NaN`-containing column depends on various factors, including the type of data the column contains. Another factor is how many rows have null values. Two cases, `fare` and `embarked`, have one and two null rows, respectively. Given that our data frame has more than 1,300 rows, missing 1 or 2 of them won't make a significant difference. So, I suggest that we remove those rows from the data frame (figure 5.6):

```
df = df.dropna(subset=['fare', 'embarked'])
```

Figure 5.6 Removing rows in which a column contains `NaN`



When it comes to the `age` column, though, we want to consider our steps carefully. I'm inclined to use the mean here, but we could use the mode. We could also use a more sophisticated technique, using the mean from within a particular cabin. We could even try to get the complete set of ages on the Titanic and choose from a random distribution built from it.

Using the mean age has some advantages: it won't affect the mean age, although it will reduce the standard deviation. It's not necessarily wrong, even though we know it's not totally right. In another context, such as sales of a particular product in an online store, replacing missing values with the mean can sometimes work, especially if we have similar products with a similar sales history.

In any event, we can replace `NaN` in the `age` column as follows (figure 5.7):

```
df['age'] = df['age'].fillna(df['age'].mean())
```

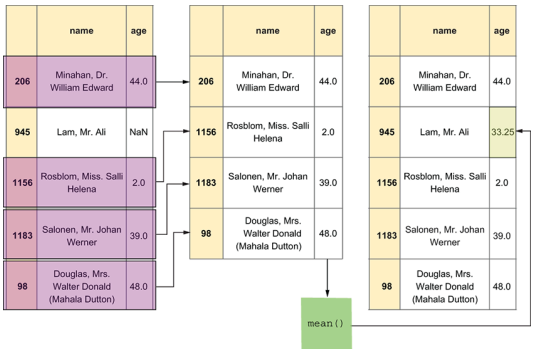


Figure 5.7 Replacing `NaN` in the `age` column with the mean of `age`

Let's break this into several parts, starting with the expression on the right side:

1. Calculate `df['age'].mean()`. Pandas ignores `NaN` values by default, which means this calculation is based on the non-null numeric values in that column. We get a single float value back from this calculation: 29.8811345124283.
2. Run `fillna` on `df['age']`. And what value should we put instead of `NaN`? What we just calculated as the mean of `df['age']`. And yes, it looks confusing to use `df['age']` twice. The result of invoking `fillna` is a new series identical to `df['age']`, except the `NaN` values are replaced with 29.8811345124283, the float we got back in the previous step.
3. The result of `df['age'].fillna` is a new series, which we assign back to `df['age']`, replacing the original values.

In the end, we've replaced any `NaN` values in `df['age']` with the mean of the existing values.

Finally, we want to set the `home.dest` column similarly to what we did with the `age` column—but instead of using the mean, we'll use the mode (i.e., the most common value). We'll do this for two reasons: first, we can only calculate the mean from a numeric value, and the destination is a categorical/textual value. Second, given no other information, we may be able to assume that a passenger is going

where most others are going. We may be wrong, but this is the least wrong choice we can make. We could, of course, be more sophisticated, choosing the mode of `home.dest` for all passengers who embarked at the same place, but we'll ignore that for now.

Our code looks very similar to what we did for the `age` column, but using `mode` instead of `mean`. And because `mode` always returns a series, we need to grab its first value with `[0]` rather than just pass it to `fillna`:

```
df['home.dest'] = df['home.dest'].fillna(df['home.dest'].mode()[0])
```

Let's break this apart:

1. Calculate `df['home.dest'].mode()`, which returns the most common value from this column. Another way to get the same value would be to invoke `df['home.dest'].value_counts().index[0]`, which counts how often each value appears in `home.dest` and returns a series with this information. We get the index from that series (the different data points from `df['home.dest']` and then get the first (i.e., most common) item from the index.

2. After grabbing the most common destination, we pass it as an argument to `fillna`, which we invoke on `df['home.dest']`. In other words, we replace all null values in `home.dest` with the non-null mode from `home.dest`.
3. Because `fillna` returns a series, we assign the result back to `df['home.dest']`, replacing the original column with the new, null-free, column.

Solution

```
filename = '../data/titanic3.xls'

df = pd.read_excel(filename)                                                    ①

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

df['age'] = df['age'].fillna(df['age'].mean())                                  ④
df = df.dropna(subset=['fare', 'embarked'])                                     ⑤
df['home.dest'] = df['home.dest'].fillna(df['home.dest'].mode())[0]             ⑥
```

① Loads all columns from Excel

② Which columns contain NaN values?

③ Shows how many NaN values each column contains

④ Replaces NaN values in the age column with the mean age

⑤ Removes null values in fare and embarked

⑥ Replaces NaN values in home.dest with the mode

You can explore a version of this in the Pandas Tutor at **http://mng.bz/n17a**.

Beyond the exercise

In these tasks, we will do something I mentioned earlier: replace `NaN` values in the `home.dest` column with the most common value from that person's `embarked` column. This will take several steps:

1. Create a series (`most_common_destinations`) in which the index contains the unique values from the `embarked` column and the values are the most common destination for each value of `embarked`.
2. Replace `NaN` values in the `home.dest` column with values from `embarked`. (Because values in `embarked` and `home.dest` are distinct, this is an OK middle step.)
3. Use the `most_common_destinations` series to replace values in `home.dest` with the most common values for each embarkation point.

Exercise 28 • Inconsistent data

Missing data is a common problem you must deal with when importing

data sets. But equally common is inconsistent data, when the same value is represented by several different values.

I once encountered this while doing a project for a university's fundraising department. Their database had been written years before and was a mess. In particular, I remember that the database column for "country" contained all of the following values:

- `United States of America`
- `USA`
- `U.S.A.`
- `U.S.A`
- `United States`
- `US`
- `U.S.`

Although people understand that these refer to the same country, a computer doesn't. If your data is inconsistent, it will be hard for you to analyze it in any sort of serious way. Thus, a big part of cleaning real-world data involves making it more consistent—or, to use a term from the world of databases, *normalizing* it.

In this exercise, we return to the parking tickets database, trying to make it more consistent and thus easier to analyze. (I am sure that even after this exercise, a data set this large will still have some

inconsistencies.) Here is what I want you to do:

1. Create a data frame from the file **nyc-parking-violations-2020.csv**. We are only interested in a handful of the columns:
    1. `Plate ID`
    2. `Registration State`
    3. `Vehicle Make`
    4. `Vehicle Color`
    5. `Street Name`
2. Determine how many different vehicle colors (the `Vehicle Color` column) there are.
3. Look at the 30 most common colors, and identify colors that appear multiple times but are written differently. For example, the color `WHITE` is also written `WT`, `WT.`, and `WHT`.
4. Prepare a Python dict in which the keys represent the various color-name inputs and the values represent the values you want them to have in the end. I suggest using longer names, such as `WHITE`, rather than shorter ones.
5. Replace the existing (old) colors with your translations. How many colors are there now?
6. Look through the top 50 colors now that you have removed a bunch of them. Are there any you could still clean up? Are there any you cannot figure out? Can you identify some consistent typos and errors in the colors?

Working it out

We're all guilty of typos—but if you make a mistake writing an e-mail, your friend or colleague will (hopefully) forgive you. In the case of data science, typos and other errors are often more insidious, because they take place one at a time as a small and unnoticed drip. When you start to analyze the data, you discover how many mistakes occurred and how many repeated themselves. This is especially true when we're getting data from people rather than from sensors and other automated equipment, although those can cause all sorts of interesting and weird problems, too.

In this exercise, I asked you to look at the colors of the vehicles that received parking tickets in New York City in 2020. As it turns out, there are many opportunities for the people issuing tickets to make mistakes, potentially affecting our analysis (although it's unlikely we would do any serious analysis of the vehicle colors).

Before we can fix the color names, we need to understand what we're dealing with. After all, maybe it isn't even a problem. After reading the data into a data frame, we can quickly check to see how many distinct vehicle colors are listed in the parking-ticket database:

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

`value_counts` is a fantastic method for getting the unique values from a series, determining how often each value appears, and sorting them from most to least common. Because `value_counts` returns a series, we can ask for its index and call `len` on it.

We find 1,896 different colors recorded for parking tickets. Color experts may argue that this is a small number of colors compared to what the human eye can distinguish, but it seems a bit high for the purposes of distinguishing cars that have been ticketed.

What were the 30 most common colors in 2020 parking tickets? Let's take a look:

```
df['Vehicle Color'].value_counts().head(30)
```

We can already see that there is little or no standardization and the people giving tickets are wildly inconsistent in how they describe colors. And that's just from looking at the first 30 colors—they've described colors almost 1,900 other ways we haven't even looked at.

To clean this up, we'll create a regular Python dictionary. We could also use a series, but a dict seems like

the easiest and most straightforward solution:

```
colormap = {'WH': 'WHITE', 'GY':'GRAY', 'BK':'BLACK',
            'BL':'BLUE', 'RD':'RED', 'SILVE':'SILVER',
            'GR':'GRAY', 'TN':'TAN', 'BR':'BROWN',
            'YW':'YELLO', 'BLK':'BLACK', 'GRY':'GRAY',
            'WHT':'WHITE', 'WHI':'WHITE', 'OR':'ORANGE',
            'BK.':'BLACK', 'WT':'WHITE', 'WT.':'WHITE'}
```

This dict has 18 key-value pairs to standardize 18 color names.

In this dict, the keys are the strings we found describing the colors and the values are the strings that we *want* to see. This sort of translation table is pretty common in data-cleaning pipelines, and over time, you'll likely find yourself adding new key-value pairs as you discover new (and surprisingly creative) ways for people to misspell color names.

By applying the `replace` method to our series (i.e., the `Vehicle Color` column), we can get back a new series. That new series can then be assigned back to `df['Vehicle Color']`, replacing our existing one:

```
df['Vehicle Color'] = df['Vehicle Color'].replace(colormap)
```

NOTE Any values not in `colormap` remain unchanged. The match in `colormap` must be precise, including whitespace, punctuation, and case.

If we check the number of distinct colors again

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

we get 1,880, which is 16 less than before. That means at two of the colors didn't change anything. How can that be? Well, it turns out *we* made two mistakes.

First, we said to look for the shortened color name `SILVE` and turn it into `SILVER`. The problem is, the backend system into which parking tickets are entered limits the `Vehicle Color` field to five characters. So changing `SILVE` to `SILVER` didn't combine two color designations into one, because there were no cars with `SILVER` in the original data set. We can thus remove `SILVER` from the `colormap` dictionary because it isn't shortened.

What about `OR`? When we mapped `OR` to `ORANGE`, we accidentally used a six-letter color name. So `OR` was a duplicate, but of `ORANG` rather than `ORANGE`. By changing `colormap` to switch from `OR` to `ORANG`, we reduced the number of different colors by one, uniting all the orange cars under one (very bright and tacky) roof.

Our final working replacement dictionary is as follows:

```
colormap = {'WH': 'WHITE', 'GY':'GRAY',
            'BK':'BLACK', 'BL':'BLUE',
            'RD':'RED', 'GR':'GRAY',
            'TN':'TAN', 'BR':'BROWN',
            'YW':'YELLO', 'BLK':'BLACK',
            'GRY':'GRAY', 'WHT':'WHITE',
            'WHI':'WHITE', 'OR':'ORANG',
            'BK.':'BLACK', 'WT':'WHITE',
            'WT.':'WHITE'}
```

We can then apply `colormap` to the colors using `replace`:

```
df['Vehicle Color'] = df['Vehicle Color'].replace(colormap)
```

The call to `replace` returns a new series in which any value in `df['Vehicle Color']` that matches a key in `colormap` is changed to be the corresponding value in `colormap`. After doing this, we can check to see how many different colors we're now tracking:

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

The result is 1,879. If we take the problem of color standardization seriously, we still have a lot of work cut out for us. And this is just for one column in one data set—you can see why data cleaning is both important and time-consuming.

Solution

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

df = pd.read_csv(filename,
```

```
                  usecols=['Plate ID',
                           'Registration State',
                           'Vehicle Make',
                           'Vehicle Color',
                           'Street Name'])

len(df['Vehicle Color'].value_counts().index)   ①
df['Vehicle Color'].value_counts().head(30)     ②

colormap = {'WH': 'WHITE', 'GY':'GRAY',
             'BK':'BLACK', 'BL':'BLUE',
             'RD':'RED', 'GR':'GRAY',
             'TN':'TAN', 'BR':'BROWN',
             'YW':'YELLO', 'BLK':'BLACK',
             'GRY':'GRAY', 'WHT':'WHITE',
             'WHI':'WHITE', 'OR':'ORANG',
             'BK.':'BLACK', 'WT':'WHITE',
             'WT.':'WHITE'}                     ③

df['Vehicle Color'] = df[
    'Vehicle Color'].replace(colormap)          ④
len(df['Vehicle Color'].value_counts().index)   ⑤
df['Vehicle Color'].value_counts().head(50)     ⑥
```

① How many different colors are listed?

② What are the 30 most commonly listed colors on parking tickets?

③ Creates a dict for translating bad color names to good ones

④ Uses replace to apply our colormap dict, assigning it back to the column

⑤ Sees that the number of colors has indeed declined

⑥ Looks at the top 50 colors and finds other potential cleanup targets

You can explore a version of this in the Pandas Tutor at **http://mng.bz/M9EW**.

Beyond the exercise

- Run `value_counts` on the `Vehicle Make` column, and look at some vehicle names. (There are more than 5,200 distinct makes, which almost certainly indicates a lot of inconsistency in this data.) What problems do you see? Write a function that, given a value, cleans up the data: putting the name in all caps, removing punctuation, and standardizing whatever names you can. Then use the `apply` method to fix the column. How many distinct vehicle makes are there when you're done?
- How standardized are the street names in the data set? What changes could you apply to improve things?
- Would you need to clean up the `Registration State` column? Why or why not?

Summary

Cleaning data is one of the most important parts of data analysis, although it's not glamorous. In this chapter, we saw that effective data cleaning requires not just knowing the techniques, but also applying judgment—knowing when you can allow null or duplicate values and

what you should do with them. pandas comes with a wide variety of tools we can use to clean our data, from removing `NaN` values to replacing them, replacing existing values, and running custom functions on each row in a series or data frame. The techniques we explored in this chapter, along with the `interpolate` method we saw in exercise 13, are important tools in your data-cleaning toolbox and will likely come up in many of the projects you work on.