12 Performance

It's hard to fathom just how powerful modern computers are. They perform billions of calculations per second, allowing us to have video chats with people around the world, predict the weather with incredible accuracy, and search through entire libraries of documents in the blink of an eye. An office worker from 100 years ago would be awestruck by how much data we can process and how little time it takes us to do so.

But let's be honest: when were you last satisfied by your computer's speed? If you're like me, you spend very little time amazed by the speed with which our computers operate and a lot of time frustrated by how long they take to do things.

I often say that Python is the perfect language for an age in which computers are cheap and people are expensive. By that, I mean Python optimizes for programmer productivity, often at the expense of efficient execution. Things aren't all bad; the fact that pandas uses NumPy under the hood makes it far faster and slimmer than would be the case with standard Python objects. The more we stay in the high-powered world of NumPy and away from built-in Python objects, the better it will be.

Beyond that general rule of thumb, though, there are numerous techniques for keeping your pandas data frames slim and your queries fast. Many of these come down to the simple rule of only using the data you need for a data frame. Because pandas keeps all data in memory, less data means faster processing and results.

In this chapter, we'll explore a number of topics having to do with pandas performance. We'll talk about how to measure the memory usage of a data frame and why pandas sometimes lies to us about it. We'll see how we can measure how much time it takes to perform tasks using `timeit`. We'll look at saving memory by using categories. And we'll explore how to speed up performance using PyArrow, the Python implementation of the Arrow library, both for loading CSV files and as a backend replacement for NumPy.

By the end of this chapter, you'll understand many of the problems surrounding pandas performance as well as how you can (and should) address them.

Table 12.1 What you need to know

Concept	What is it?	Example	To learn more
`df.info`	Gets information about a data frame, including its memory usage	`df.info()`	http://mng.bz/D4eE
`df.memory_usage`	Gets information about a data frame's memory usage	`df.memory_usage(deep=True)`	http://mng.bz/lWjy
Categorical data	Pandas documentation for categorical data	`df['a'].astype('categorical')`	http://mng.bz/BmaJ
`df.to_feather`	Writes a data frame to feather format	`df.to_feather('mydata.feather')`	http://mng.bz/d1pQ
`pd.read_feather`	Creates a data frame based on a feather-formatted file on disk	`df = pd.read_feather('mydata.feather')`	http://mng.bz/rWlX
`pd.read_csv`	Returns a new data frame based on CSV input	`df = df.read_csv('myfile.csv')`	http://mng.bz/V1r5

`pd.read_json`	Returns a new data frame based on JSON input	`df = df.read_json('myfile.json')`	http://mng.bz/x4qB
`time.perf_counter`	Gets the number of seconds (useful for timing programs)	`time.perf_counter()`	http://mng.bz/AonW
`df.query`	Writes an SQL-like query	`df.query('v > 300')`	http://mng.bz/ZqBZ
`df.eval`	Performs actions and queries on a data frame	`df.eval('v + 300')`	http://mng.bz/Rx9P
`pd.eval`	Performs a variety of pandas actions in an evaluated string	`pd.eval('df.v > 300')`	http://mng.bz/2D9X
`timeit`	Python module for benchmarking code speed, and a Jupyter “magic command” for invoking it	`%timeit 3+2`	http://mng.bz/1qKg
`isin`	Checks whether a value is in a	`df['a'].isin([10, 20, 30])`	http://mng.bz/Pz0P

	Python sequence		
`pd.CategoricalDtype`	Returns a new categorical dtype	`pd.CategoricalDtype(['a', 'b', 'c', 'd'])`	http://mng.bz/Jgev

Saving memory with categories

Let’s say we want to work with data from our Olympics CSV file:

```
filename = '../data/olympic_athlete_events.csv'
df = pd.read_csv(filename)
```

How much memory does this data set consume? That’s an important question when working with pandas, because all our data needs to fit into memory. We can find out by running the `memory_usage` method on our data frame:

```
df.memory_usage()
```

This returns a series telling us how many bytes are consumed by each column. (The column names from `df` constitute the index of the returned series.) We can get the total memory usage by summing the values:

```
df.memory_usage().sum()
```

On my computer, this comes up as 32,534,048 bytes or just over 31 MB of RAM.

But you know what? This number is completely wrong. That’s because pandas, by default, ignores the size of any Python objects contained in a data frame. Given that these objects are generally strings and can be any length, the difference between the actual memory usage and what is reported here can be big.

We can tell pandas to include all the objects in its size calculation by passing the `deep=True` keyword argument:

```
df.memory_usage(deep=True).sum()
```

On my computer, the same data frame gives me a result of 186,408,012 bytes, or about 182 MB of RAM—five times the originally calculated amount.

But wait: this is a lot of memory, and the data set is relatively small. A much larger data set will obviously consume much more memory, potentially more than I can fit into my computer. How can I cut down the size of the data set, thus allowing me to potentially work with more data? We’ve already talked about several of them in past chapters:

- Limit which columns are imported, by passing a value to `usecols`
- Explicitly specify the `dtype` for each column, allowing us to choose types with fewer bits while simultaneously speeding up the loading of data

However, the majority of the memory is being used by strings. We can see this by running `df.memory_usage(deep=True).sort_values()`. The columns using the most memory contain strings, not numbers. This means we need to somehow reduce the size or number of the text strings in our data frame.

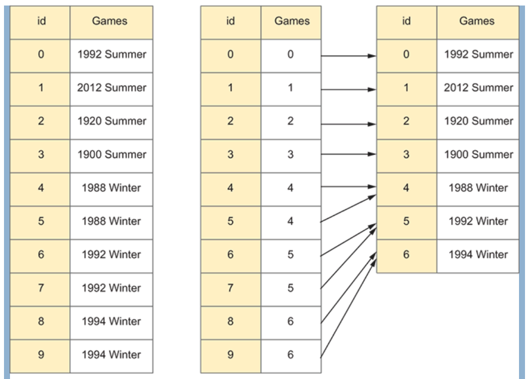
One way to do this is with a special pandas data type known as a *category*. In the case of a category, each distinct string value is stored a single time and then referred to multiple times. This replacement is completely transparent to us, as users of the data frame: we can continue to pretend that the column contains strings, including use of the `str` accessor to apply string methods to every element of the column.

We've often used `astype` to create a new series based on an existing one. We can do the same thing to create a new categorical column based on one containing text strings.

We demonstrate this using the `Games` column, which contains a different string for each time the Olympics were held. Running `df['Games'].value_counts().head(10)` gives the following output:

```
Games
2000 Summer    13821
1996 Summer    13780
2016 Summer    13688
2008 Summer    13602
2004 Summer    13443
1992 Summer    12977
2012 Summer    12920
1988 Summer    12037
1972 Summer    10304
1984 Summer     9454
Name: count, dtype: int64
```

In other words, the string `2000 Summer` appears in the `Games` column 13,821 times. By creating a category, we can create a single string with that value, assign an integer to represent that string, and then store the integer in the column rather than the string (or, more accurately, a reference to the Python string object containing that value). Assuming that the integer is smaller than the reference, this can save a lot of memory.



id	Games
0	1992 Summer
1	2012 Summer
2	1920 Summer
3	1900 Summer
4	1988 Winter
5	1988 Winter
6	1992 Winter
7	1992 Winter
8	1994 Winter
9	1994 Winter

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

id	Games
0	1992 Summer
1	2012 Summer
2	1920 Summer
3	1900 Summer
4	1988 Winter
5	1992 Winter
6	1994 Winter

Ten rows from `Games`, before and after being turned into a category

We can create our category this way:

```
df['Games'].astype('category')
```

However, this doesn't do anything useful, because it doesn't store our new series anywhere. It's often easiest to just assign the newly created categorical series back to the original column, replacing it with an equivalent-but-slimmer version:

```
df['Games'] = df['Games'].astype('category')
```

How much memory does that action save us? We can find out by running `memory_ usage` again:

```
df.memory_usage(deep=True).sum()
```

Sure enough, memory usage has gone down to 168,248,812 bytes, or more than 160 MB. In other words, we've trimmed 15 MB of storage from our data frame simply by turning the `Games` column from a string into a category.

Which columns should we attack first? Well, we want those in which the same strings are often repeated. Consider this code:

```
(df.count() / df.nunique()).sort_values(ascending=False)
```

Here, we divide the number of non-null rows in each column by the number of distinct values in that column. The higher the number, the more times the same string is repeated, and thus the greater the memory savings we can achieve by switching the column to a category. We then use `sort_values(ascending=false)` to sort the rows in order of priority.

I decided to choose all categories with a `dtype` of `object` in which a value is repeated at least 100 times. This leads to the following code:

```
for column_name in ['Sex', 'Season', 'Medal', 'City', 'Games',
                    'Sport', 'NOC', 'Event', 'Team']:
    print(column_name)
    df[column_name] = df[column_name].astype('category')
```

The result? A data frame that’s just over 33 MB in size. After only a handful of lines of code that took several seconds to execute, we’ve cut the memory requirement to about 20% of its original value. That seems like an extremely worthwhile use of our time.

But wait a second: this method creates the category based on the data that's already in the series. What if we know the series may include other values in the future, even if they're not in the original data set? Here's a simple example:

```
s = Series(['a', 'b', 'c', 'a', 'b', 'c', 'c', 'c']).astype('category')
```

We now try to set one of the values to `'d'`:

```
s.loc[7] = 'd'
```

This fails with a `TypeError` exception, telling us that we cannot set a value that wasn't included in the category.

We can solve this problem by creating the category before creating the series (or column of the data frame), including all possible values it may contain. Then we can ask pandas not to create the category with `astype` but rather to assign the specific category type that we've defined, with all its values. Let's first see how this may work with the earlier series:

```
abcd_category = pd.CategoricalDtype(['a', 'b', 'c', 'd'])
s = Series(['a', 'b', 'c', 'a', 'b',
    'c', 'c', 'c']).astype(abcd_category)
s.loc[7] = 'd'  # Success!
```

In this code, we create a new category with all its values by calling `pd.CategoricalDtype`. Then, when we call `astype`, we pass the category we created rather than asking pandas to create a new, anonymous category. We can do the same in our Olympics data frame:

```
medals_category = pd.CategoricalDtype(['Gold', 'Bronze', 'Silver'])
df['Medal'] = df['Medal'].astype(medals_category)
```

xercise 48 • Categories

We've explored New York City's parking tickets on several previous occasions in this book, but we were always concerned by how much memory the full data set would require. Indeed, if I load the entire data set onto my computer, it uses a *lot* of memory—about 18 GB. We'd like to crunch that down to a much smaller number by turning many of the columns into categories.

NOTE Because I realize that not everyone reading this book has many gigabytes of RAM to spare, you'll limit the number of columns you load for this exercise. If you are fortunate enough to have such a computer, though, I encourage you to load the entire data set into memory and pare the columns down using the same techniques. If you're like me, you'll be amazed by how much memory categories can save you. If your computer cannot load even the subset of columns I specify for this exercise, feel free to cut them down even further.

1. Read the NYC parking violations data into a data frame. Only load the following columns: `Plate ID`, `Registration State`, `Vehicle Make`, `Vehicle Color`, `Vehicle Body Type`, `Violation Time`, `Street Name`, and `Violation Legal Code`.
2. Determine how much memory is being used by the data frame you've created.
3. Turn each column into a category.
4. Answer these questions:
    1. What types are your columns now?
    2. How much memory does your data consume now?
    3. How much memory have you saved thanks to using categories?

Working it out

This exercise has fewer steps than many of the recent ones we've done, for two reasons. First, I want to show you how easily we can create and work with categories. Second, when we're dealing with large amounts of memory, even the fastest and most tricked-out computers can take a while to calculate things.

With that in mind, let's go through the code and see what we can do. First, we load the data set, limiting ourselves to the eight columns I asked for:

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

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

You'll likely get a `DtypeWarning` from pandas because one or more columns have mixed types. We've seen this warning before, and we'll soon be turning this column into a category, so we can ignore this warning.

Next, I asked you to calculate how much total memory the data frame is using. There are actually two ways to do this. The first is to run the `memory_usage` method, passing the keyword argument `deep=True`. This returns a series in which the index contains the data frame's column names and the values show how much memory is being used by each column:

```
df.memory_usage(deep=True)
```

On my computer, I get the following result:

```
Index                          128
Plate ID                 798282162
Registration State       737248306
Vehicle Body Type        758166224
Vehicle Make             768611575
Violation Time           774726961
Street Name              879156216
Violation Legal Code     515644296
Vehicle Color            735089399
dtype: int64
```

According to this report, each column requires more than half a gigabyte of RAM. Even in our modern era of cheap, plentiful RAM, this is still a large data set—and given the alternative, there's no reason for us to use all this memory.

I want to remind you that it's important to always use `deep=True` if you truly want to know the size of your data frame. If we hadn't passed `deep=True`, we would have gotten something like this:

```
Index                         128
Plate ID                 99965872
Registration State       99965872
Vehicle Body Type        99965872
Vehicle Make             99965872
```

```
Violation Time           99965872
Street Name              99965872
Violation Legal Code     99965872
Vehicle Color            99965872
dtype: int64
```

Notice how all the columns, aside from the index, have the same size: 99,965,872 bytes—basically 100 MB. Not a small amount of memory, but far less than the actual size of our data, whose size we calculated using `deep=True`.

Why does pandas not run `deep=True` all the time? Because instead of just checking the size of the memory allocated by the NumPy backend, pandas has to go through each object in Python and ask for its size. That can take substantially longer, so we have to ask for it explicitly.

Calling `memory_usage` returns a series: the size of each column. We can add the values using `sum`:

```
df.memory_usage(deep=True).sum()
```

On my computer, the result is 5,966,925,267, or about 6 GB.

Next, I asked you to turn each of these columns into a category. Remember that given a column named `colname`, we can turn it into a category with

```
df['colname'] = df['colname'].astype('category')
```

When we do this, pandas removes `NaN` values in the column, looks at the remaining unique values, builds a new category object from it, and then uses that category to assign values. Although the values still appear to be there, as before, pandas has replaced them with much-smaller integers, storing each string a single time.

Before transforming the columns into categories, we keep track of how much memory our original version of the data frame is using:

```
orig_mem = df.memory_usage(deep=True).sum()
```

Next, we do the transformation itself, using a `for` loop. You may be surprised to see this suggestion, given that I often point out that if you're using a `for` loop in pandas, you're almost certainly doing something wrong. But that's if you're trying to perform a calculation on each row; for such purposes, pandas has a lot of functionality that is generally faster than any loop. Because so much of the backend data uses NumPy, pulling the data into Python data structures uses significantly more memory than taking advantage of its vectorized, compiled, and optimized systems.

But this case is different: we're interested in performing one vectorized operation per column. There isn't any vectorizing to be done across the columns. For this reason, a `for` loop is perfectly reasonable. The index object we get back from `df.columns` is iterable, allowing us to get each column name, one at a time. We thus write

```
for one_colname in df.columns:
    print(f'Categorizing {one_colname}...')
    df[one_colname] = df[one_colname].astype('category')
    print('\tDone.')
```

Notice that we put two calls to `print` inside the `for` loop: once before starting the transformation and once after. This is because the creation of a category can take some time, and it would be useful to know when pandas is starting to work on a column and when it has finished. In addition, if something goes wrong while creating the columns, we know exactly where we were when the problem took place.

After performing this transformation, we want to get confirmation that things changed. By retrieving `dtypes` on our data frame, we can see precisely what type each column has:

```
df.dtypes
```

Sure enough, pandas shows that all the columns have been changed to have `category` types. But what effect does that have on the memory usage? As before, we can ask for a deep memory check:

```
new_mem = df.memory_usage(deep=True).sum()
```

This time, on my computer, I get the value 574,455,678—still half a gigabyte of RAM, but a far cry from the original value of 6 GB. In other words, we have cut down our memory usage by about 90%! And indeed, if we perform a quick calculation

```
new_mem / orig_mem
```

we get a result of 0.096, meaning we are indeed using approximately 10% of the original data frame's memory while still using the same data and enjoying the same benefits from it.

How much memory?

The `df.info` method returns a summary of information about the data frame, including the total memory usage. By default, it doesn't do a "deep" memory check; in such cases, and if there are object columns, the memory is returned with a `+` sign following the number. You can avoid the `+`

and get a precise calculation by passing `memory_ usage='deep'` as a keyword argument to `info`:

```
df.info(memory_usage='deep')
```

This gives you a summary of the total memory used.

Solution

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

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

orig_mem = df.memory_usage(deep=True).sum()                  ①

for one_colname in df.columns:                               ②
    print(f'Categorizing {one_colname}...')
    df[one_colname] = df[one_colname].astype('category')     ③
    print('\tDone.')

df.dtypes()                                                  ④

new_mem = df.memory_usage(deep=True).sum()                   ⑤

print(new_mem /  orig_mem)                                   ⑥
```

① Gets the memory usage of each column, sums them, and stores the result in orig_mem

② Goes through each column in df

③ Turns each column into a category column

④ Gets the dtype of each column

⑤ Gets the memory usage of each column after categorization and stores it in new_mem

⑥ Shows how much memory we are using now

Beyond the exercise

- Without calculating: Of the columns you loaded, which would make less sense to turn into categories? Once you've thought about it, calculate how many repeated values are in each column and determine (more formally) which would give the biggest improvement when using categories.
- In exercise 25, we saw that the vehicle makes and colors were far from standardized, with numerous misspellings and variations. If you were to standardize the spellings before creating categories, would that have any effect on the memory savings you gain from categorization? Why or why not?
- Read only the first 10,000 lines from the CSV file, but all columns. Show the 10 columns that will most likely benefit greatest from using categories.

Apache Arrow

For nearly all of this book, I've assumed that pandas acts as something of a wrapper around NumPy. Sure, it provides a great deal of functionality, but the actual storage is handled by NumPy, which means the `dtype` of a column is nearly always defined by NumPy. The major exceptions are strings (stored as an `object` dtype) and categories (discussed elsewhere in this chapter).

An open source project known as Apache Arrow may change all that. Arrow is designed to be a highly efficient in-memory storage mechanism for data. Some basic facts about Arrow:

- Arrow works not only with pandas but also with other languages and systems such as R and Apache Spark.
- The Python bindings for Arrow are known as PyArrow.
- Arrow has its own data types, similar to but distinct from those defined by NumPy.
- Arrow's types are nullable, meaning if we have an integer column with a single `NaN` value, the column's `dtype` doesn't need to change to a floating-point type.

- Arrow supports two file formats, feather and parquet, which are binary, thus consuming less disk space and taking less time to read and write.
- Arrow can also read and write CSV files.

Let's start with this final point: by default, pandas reads CSV files using its own internal engine. We can speed up the loading of CSV files by asking pandas to instead use PyArrow:

```
df = pd.read_csv(filename, engine='pyarrow')
```

I've found that using the PyArrow engine for loading CSV is 20 times faster than the built-in engine. Unless you're working with very small data sets, it's probably worth always using this option with `read_csv`.

What about Arrow's binary formats? The feather format, as I mentioned, combines compression and binary storage to give us smaller files that are faster to read and write than either CSV or JSON.

To write a pandas data frame to a feather-formatted file, we can use the `to_feather` method, which works similarly to `to_csv` and `to_json`:

```
df.to_feather('mydata.feather')
```

We can similarly read from a feather-formatted file into a data frame using the `pd.from_feather` method, which works similarly to `from_csv` and `from_json`:

```
df = pd.from_feather('mydata.feather')
```

As of pandas 2.0, there is also experimental support for using PyArrow for backend storage, rather than NumPy. When we read a CSV file into pandas, we can specify that it should use PyArrow by passing the `dtype_backend='pyarrow'` keyword argument:

```
df = pd.read_csv(filename, dtype_backend='pyarrow')
```

If you check the `dtypes` attribute on your data frame, you'll find that the types are all from PyArrow and not from NumPy.

Note that the use of `dtype_backend` is separate from the engine used to read the CSV file. You can use one or both of these keyword arguments.

The use of PyArrow as a pandas backend is still experimental as of this writing, and it's also not guaranteed to be faster. In some experiments that I ran just after pandas 2.0 was released, simple comparisons were faster with PyArrow, but more sophisticated queries, such as grouping and joins, took longer than the NumPy backend. I have no doubt that this will improve over time, but for now you shouldn't assume that PyArrow will always be faster. That said, it's worth doing some experiments to see how it works on your data and your queries.

xercise 49 • Faster reading and writing

Each file format has its own advantages and disadvantages, among them being the speed with which you can read and write data. Given a data frame, is it faster to write it as a CSV, JSON, or feather file? (If you read the side bar on Apache Arrow and feather, you may have a good sense of the answer.) How much of a difference is there? And is there a significant difference in speed when reading CSV, JSON, and feather files into a data frame?

To understand that, I'm asking you to do the following:

1. Load the New York parking data CSV file into a data frame.

2. Write the data frame out to the filesystem in each of three different formats: CSV, JSON, and feather. Time the writing of each format, and print the format along with the number of seconds it took to write to it.
3. Check the size of the files you've created.
4. Read each file you created into a data frame. Once again, time how long the loading takes and print that timing alongside the format name.

NOTE The New York parking data set is large and may overwhelm computers with less than 32 GB of free memory. If you're working on such a computer, I encourage you to use the `usecols` keyword parameter to reduce the number of columns read into the data frame at the start of the exercise. You may see less of a difference between writing to and then reading from the various formats, but at least you'll be able to finish the exercise.

Working it out

The first thing I asked you to do was load the New York parking violations data set for 2020. We'll assume that your computer has enough memory to load the entire thing, which we do as follows:

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

```
    df = pd.read_csv(filename, low_memory=False)
```

Notice that we pass the `low_memory=False` keyword argument. This tells pandas that we have enough RAM that it can look through all the rows in the data set when trying to determine what `dtype` to assign to each column.

With the data frame in place, we can begin writing to different formats, timing how long each takes. But, of course, that means we need some way to keep track of time. Python's `time` module, part of the standard library, provides a number of different methods that could theoretically be used, but it's generally considered best to use `time.perf_counter()`. This function uses the highest-resolution clock available and returns a float indicating a number of seconds. The number returned by `perf_counter` should not be relied on for calculating the current date and time; but within the same program, it can be used to measure the passage of time, which is precisely what we want to do.

NOTE Python's standard library also includes the `timeit` module (http://mng.bz/67OA), which includes a number of utilities for benchmarking. I'm generally a big fan of `timeit`, and we'll use it in exercise 50. But `timeit` runs code several times and reports the mean time after those runs. In this case, where the code will take a long time to run, we run it a single time—and thus opt to use `perf_counter`.

We want to try writing to CSV, JSON, and feather formats. In theory, we could write code that looks like this:

```
df.write_json('parking-violations.json')
df.write_csv('parking-violations.csv')
df.write_feather('parking-violations.feather')
```

But, of course, we want to determine how long each one takes. So we add some benchmarking code above and below each format:

```
start_time = time.perf_counter()                        ①
df.write_json('parking-violations.json')
end_time = time.perf_counter()                          ②
total_time = end_time - start_time                      ③
print(f'\tWriting JSON: {total_time=}')

start_time = time.perf_counter()                        ④
df.write_csv('parking-violations.csv')
end_time = time.perf_counter()
total_time = end_time - start_time
print(f'\tWriting CSV: {total_time=}')

start_time = time.perf_counter()                        ⑤
df.write_feather('parking-violations.feather')
end_time = time.perf_counter()
total_time = end_time - start_time
print(f'\tWriting feather: {total_time=}')
```

① Gets the time at the start of the JSON run

② Gets the time at the end of the run

③ Calculates the run time

④ Repeats for CSV

⑤ Repeats for JSON

This code works and does the job. But it also violates an important rule of programming: “Don’t repeat yourself,” often abbreviated DRY. We’re basically doing the same thing three times. If we can consolidate that code into a loop, our code will be cleaner, easier to read, easier to debug, and easier to extend. But how can we do that? After all, we’re calling three different methods.

This is where some knowledge and understanding of Python, not just pandas, comes in handy: we can create a dictionary in which the keys are the file formats and the values are the methods we want to use to write the data frame. That’s right—we can store any Python object, including a function or method—as the value in a dict. We can thus say

```
root = 'parking-violations'
write_methods = {'JSON': df.to_json,
             'CSV': df.to_csv,
             'feather': df.to_feather
            }

for one_format, method in write_methods.items():
      print(f'Saving in {one_format}')
      start_time = time.perf_counter()
      method(f'parking-violations.{one_format.lower()}')
      end_time = time.perf_counter()

      total_time = end_time - start_time
      print(f'\tWriting {one_format}: {total_time=}')
```

The `for` loop here iterates over the dict, getting each key (a string, stored in `one_ format`) and value (`method`, containing the method to be run) from the `write_methods` dict. We print the current format, just for debugging purposes, and then run `time .perf_counter()`, getting back the current time (more or less) in seconds. We then invoke `method` on the filename via an f-string. After writing the file to disk, we call `time .perf_counter()` again, storing the difference in `total_time`, which we then print.

On my computer, I get the following results from running this code:

```
Saving in JSON
        Writing JSON: total_time=46.29149689315818
Saving in CSV
        Writing CSV: total_time=114.35314526595175
Saving in feather
        Writing feather: total_time=7.929971480043605
```

In other words, it took about 114 seconds (nearly two minutes) to write our data frame to a CSV file. It took 46 seconds to write the same data to JSON. But it took just under 8 seconds—about 14 times faster!—to write the same data to feather. If that doesn't convince you to consider using feather, I'm not sure what will.

Notice that for this code to work, we have to define `df`, the data frame, *before* the `write_methods` dictionary is defined. We also use `one_format.lower()` to take the format name and ensure that it is only in lowercase letters.

How big are the files we've created? We again rely on Python's standard library. We've seen the `glob.glob` function in previous exercises; here we use it to retrieve all filenames that start with the value of our `root` variable. But we then want to get the size of each file, something we can do easily with `os.stat`. This function returns a special data structure that's modeled on Unix's `stat` functionality. In Python, we can get the size of the file in bytes by retrieving the `st_size` attribute from the value we get back from `os.stat`:

```
for one_filename in glob.glob(f'{root}*'):
    print(f'{one_filename:27}: {os.stat(one_filename).st_size:,}')
```

Inside the f-string, we use two tricks to adjust the way the values are formatted:

- We tell the f-string to pad `one_filename` with spaces so each filename uses 27 characters. This help ensure that the results line up.
- We tell the f-string to add commas before every three digits in the integer it displays, making them more readable.

The result, on my computer, is

```
parking-violations.json    : 8,820,247,015
parking-violations.csv     : 2,440,860,181
parking-violations.feather : 1,466,535,674
```

We can see here that the CSV file is about 2 GB in size, the JSON file is about 8 GB (!), and Apache Arrow's feather format is just over 1 GB. This isn't the only reason writing feather files is faster, but it's certainly one of them; at the end of the day, pandas has to write one-eighth as much data to disk.

However, I also wanted you to benchmark reading these files back from the filesystem. We use the same technique as before: creating a dictionary (this time called `read_methods`) containing the file extensions and the methods we want to run. The code is as follows:

```
read_methods = {'JSON': pd.read_json,
           'CSV': pd.read_csv,
           'feather': pd.read_feather
          }
```

```
for one_format, method in read_methods.items():
    print(f'Reading from {one_format}')
    start_time = time.perf_counter()
    df = read_methods[one_format](
        f'parking-violations.{one_format.lower()}')
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tReading {one_format}: {total_time=}')
```

As before, we iterate over a dictionary, getting each key (a string, stored in `one_ format`) and value (`method`) from the `read_methods` dict. We print the current format and then run `time.perf_counter()`. We retrieve the appropriate read method with `read_methods[one_format]` and invoke the method we get on the appropriate filename. After reading the file into a data frame, we call `time.perf_counter()` again, storing the difference in `total_time`, which we then print.

If you're like me, you'll likely get the `DtypeWarning` we've previously discussed. (As a reminder, this warning crops up when pandas is trying to figure out the type of data contained in a column, doesn't read all the rows to avoid using too much memory, and then worries that it may guess the `dtype` incorrectly. This doesn't happen in other formats because the type of data we're reading into each column is more explicit.) For our purposes, we can ignore it, in no small part to avoid having to worry about which method and which format are being read. But the benchmarking results are as follows:

```
Reading from JSON
        Reading JSON: total_time=469.92014819500037
Reading from CSV
        Reading CSV: total_time=35.20077076088637
Reading from feather
        Reading feather: total_time=9.132312984904274
```

This time the JSON file takes the longest to read into memory, at a hefty 469 seconds, or nearly 8 minutes. In second place, and taking less than 10% of the time, is CSV, at 35 seconds. But the speed champion remains feather, taking just over 9 seconds.

From this simple demonstration, it seems pretty clear that Apache Arrow and its feather format are significantly faster for reading and writing than both CSV and JSON. This doesn’t mean you can or should move everything to feather—but it has a number of clear advantages, both in terms of speed and in its footprint on the filesystem.

Solution

```
import glob
import os

filename = '../data/nyc-parking-violations-2020.csv'
df = pd.read_csv(filename, low_memory=False)
```

```
root = 'parking-violations'
write_methods = {'JSON': 'to_json',                                    ①
           'CSV': 'to_csv',
           'feather': 'to_feather' }

for one_format, method in write_methods.items():                      ②
    print(f'Saving in {one_format}')
    start_time = time.perf_counter()
    method(f'parking-violations.{one_format.lower()}')                 ③
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tWriting {one_format}: {total_time=}')

for one_filename in glob.glob(f'{root}*'):                              ④
    print(f'{one_filename:27}: {os.stat(one_filename).st_size:,}')     ⑤

read_methods = {'JSON': 'read_json',
           'CSV': 'read_csv',
           'feather': 'read_feather' }

for one_format, method in read_methods.items():                        ⑥
    print(f'Reading from {one_format}')
    start_time = time.perf_counter()
    df = read_methods[one_format](
        f'parking-violations.{one_format.lower()}')                    ⑦
    end_time = time.perf_counter()

    total_time = end_time - start_time
    print(f'\tReading {one_format}: {total_time=}')
```

① Dict of formats and methods

② Iterates over formats and write methods

③ Invokes the method on the filename

④ Goes through each filename we created

⑤ Uses os.stat to display the size of each file

⑥ Iterates over formats and read methods

⑦ Invokes the appropriate read method for the format

Beyond the exercise

- If you read the CSV file using the `pyarrow` engine, do you see any speedup? That is, can you read CSV files into memory any faster if you use a different engine?
- If you specify the dtypes to `read_csv`, does it take more time or less than without doing so?
- How much memory does the data frame use with a NumPy backend versus a PyArrow backend?

Speeding things up with eval and query

Over the course of this book, I've emphasized a number of techniques that you should use to speed up your pandas performance:

- Never use standard Python iterations (`for` loops and comprehensions) on a series or data frame.
- Take advantage of broadcasting.

- Use the `str` accessor for anything string related.
- Use the smallest `dtype` you can without sacrificing accuracy.
- Avoid double square brackets when setting and retrieving values.
- Load only those columns that you really need for your analysis.
- Columns with repeated values should be turned into categories.
- Use a binary format, such as feather, for data you'll repeatedly save or load.

Even after using all these techniques, we may find that our queries are still running slowly or using lots of memory. This often occurs when performing an arithmetic operation on two columns, each of which contains many rows. A related problem is when broadcasting an operation on a scalar and a series. Although pandas takes advantage of the high-speed calculations in NumPy, much of the work is still done within the Python language, which is slower to execute than C.

Another problem occurs when creating a boolean series for use as a mask index based on several conditions. It's certainly convenient to use `&` and `|` to combine conditions with logical "and" and "or," but behind the scenes, pandas has to create multiple boolean series, which are then combined. If we have 1 million rows in your original column, combining three conditions creates at least 3 million rows in temporary series before combining and applying them together.

We can avoid these problems, as well as make our queries more readable, using the `query` method that I introduced back in chapter 2, as well as two versions of the more general `eval` method. These reduce the memory required in queries using `|` and `&` and can often execute expressions in a library known as `numexpr`. The combination of reduced memory and increased speed can sometimes give dramatically faster results while also using fewer resources.

However, it’s important to understand that these methods are not cure-alls for performance problems:

- Using them on small data frames with fewer than 10,000 rows will often result in slower performance, not faster performance.

- Often, the bottleneck in performance is in the assignment or retrieval of elements, not in the calculation. There won’t be a speed boost in such cases.
- You’ll need to install the `numexpr` package from PyPI and then explicitly tell pandas to use it. If you don’t make this explicit, pandas will use its default Python-based engine for parsing the query string, resulting in no speedup.
- Anything that doesn’t involve calculations, comparisons, and boolean operators will either raise an exception or run at the standard (non-enhanced) speed.

Let’s start by looking at the `query` for data frames. We’ll then talk about two versions of `eval` that are part of the same family.

Given a data frame `df`, the method `df.query` allows us to describe which rows you we want to get back from `df`. The description is passed as an SQL-like string in which

the columns can be named as if they were variables. The result of the query is a data frame, a subset of `df`, with all the columns from `df` and those rows for which the comparison returned a `True` value. For example, given a data frame `df` with numeric columns `a` and `b` in which we want all rows where `a` is greater than 100 and `b` is less than 700, we would normally say

```
np.random.seed(0)
df = DataFrame(np.random.randint(0, 1000, [5,5]),
               index=list('vwxyz'),
               columns=list('abcde'))

df.loc[((df['a'] > 100) &
        (df['b'] < 700))]
```

But using `df.query`, we can instead write

```
df.query('a > 100 & b < 700')
```

The version using `query` will sometimes run faster, but it will almost always use less memory. That's because it doesn't need to create two separate, temporary boolean series, one for `a > 100` and another for `b < 700`. We may not see these boolean series when running a traditional query, but they're there and can use a great deal of memory without us realizing it. I should add that some people prefer to use `df.query` for all their pandas work because of its readability and reduced memory use.

A related data frame method is `df.eval`, which allows us to retrieve from a data frame (as in `df.query`) as well as perform other actions, including broadcasting and assigning. For example:

```
df.eval('(a + b)* 3')
```

This code adds columns `a` and `b` and multiplies the new series by 3 via broadcasting. The returned value is a series. What if we were to pass the same code we used before, with `df.query`?

```
df.eval('a > 100 & b < 700')
```

This returns a boolean series. Whereas `df.query` applies that boolean series to `df`, `df.eval` returns the boolean series itself and allows us to apply it if and when we want to do so. We can even add a new column (or update an existing one) by assigning to a column name:

```
df.eval('f = d + e - c')
```

Using a triple-quoted string, we can perform multiple assignments with `df.eval`:

```
df.eval('''
f = d + e - c
g = a * 2
h = a * b
''')
```

In general, `df.eval` can be used for either conditions or assignments. However, when we pass a triple-quoted string to `df.eval`, it is only for assignments; conditions aren't allowed.

The third and final method that allows us to use less memory, speed up computation, and write more readable queries is `pd.eval`. Notice that this is a top-level function in the `pd` namespace rather than a method we run on a specific data frame. We can use `pd.eval` instead of `df.eval`, although we need to explicitly state the name of the data frame we're working on. For example, we can say

```
pd.eval('df[df.a > 100 & df.b < 700]')
```

When using `pd.eval`, you'll probably want to use the dot syntax to reference columns, rather than the square-bracket syntax that I have generally used in this book, to avoid too much syntactic messiness. To retrieve column `a` from data frame `df`, we say `df.a` rather than `df['a']`. This also means column names cannot contain spaces.

This code returns all rows of `df` in which `a` is greater than 100 and `b` is less than 700, as before.

However, we have written the query as a string, which is passed to `numexpr`. That package will, as we've seen, use less memory and (usually) result in better performance. Note that a call to `df.eval` is translated into a call to `pd.eval`, which means you can probably get better performance if you just call `pd.eval`. That said, the convenience of the syntax in `df.eval` is hard to beat.

As with `df.eval`, we can assign to one or more columns in the string we pass to `pd.eval`. But because we're invoking `pd.eval`, the data frame on which the assignment should take place isn't known to the system. We must set it by passing the `target` keyword argument. The assignment is reflected in the data frame that is returned:

```
pd.eval('f = df.d + df.e - df.c', target=df)
```

So, when should you use each of these? Again, the biggest wins are likely to be with compound queries (using `&` and `|`) on large data frames. The larger the data frame and the more complex the query, the bigger the speed boost you may see—but even if you don't, you'll almost certainly be using less memory.

Meanwhile, here's a quick recap on each of these three functions:

- To retrieve selected rows from a data frame, use `df.query`.
- To assign multiple columns or to perform either queries or assignments on a data frame, use `df.eval`.
- To work on multiple data frames, use `pd.eval`. But it doesn't handle multiline assignments, and the syntax makes it uglier.

xercise 50 • "query" and "eval"

In this exercise, we'll look through New York parking tickets one final time, running queries using the traditional `df.loc` accessor and also using `df.query` and `df.eval`. For each of these questions, I'd like you to run the query via `timeit`, allowing us to compare the executing time needed for the various types of queries. Specifically, I'd like you to

1. Load the New York parking data CSV file into a data frame. You'll only need the following columns: `Plate ID`, `Registration State`, `Plate Type`, `Feet From Curb`, `Vehicle Make`, and `Vehicle Color`.

2. Rename the columns to `pid`, `state`, `ptype`, `make`, `color`, and `feet`. (This will make it easier to use `df.eval`.)
3. Find all cars whose registration state is from New York (`NY`), New Jersey (`NJ`), or Connecticut (`CT`) using `.loc`.
4. Find all cars whose registration state is New York, New Jersey, or Connecticut using `df.query`.
5. How much faster is it to use `query`?
6. Use `isin` to search for the states. How does this technique compare?
7. Perform each of the following queries using `df.loc`, `df.query`, and `df.eval`, all within `timeit`. In each case, which type of query runs the fastest?
    1. Find cars from New York.
    2. Find cars from New York with passenger (`PAS`) plates.
    3. Find white cars from New York with passenger (`PAS`) plates.
    4. Find white cars from New York with passenger (`PAS`) plates that were parked > 1 foot from the curb.
    5. Find white Toyota-brand cars from New York with passenger (`PAS`) plates that were parked > 1 foot from the curb.
8. Which type(s) of query appears to run the fastest?

Working it out

In this exercise, I want you to learn several things:

1. How to formulate the same query using `.loc`, `df.query`, and `df.eval`
2. How to use `timeit` to time your queries and thus compare their relative speeds
3. What may lead a query to be slower
4. Some of the syntactic problems associated with alternative query mechanisms

The first thing I asked you to do was load a number of columns from the New York parking-ticket dataset, much as we've often done in this book:

```
df = pd.read_csv(filename,
     usecols=['Plate ID', 'Registration State',
         'Plate Type',
         'Vehicle Make', 'Vehicle Color', 'Feet From Curb'])
```

There is nothing inherently wrong with loading the data this way. However, when we use `pd.query` and `pd.eval`, it's often annoying to have column names that include spaces. Yes, we can use backticks, but it's more convenient to give them names that allow us to treat them as variables inside the query string. So although there's nothing technically wrong with loading the data as we do here, we then want to set the headers to be single-word names. We can do that by assigning a list of strings to `df.columns`:

```
df.columns = ['pid', 'state', 'ptype', 'make', 'color', 'feet']
```

You may be thinking that it would be more effective to set these names as part of the call to `read_csv`. After all, `read_csv` has a `names` parameter, which takes a list of strings that are assigned to the newly created data frame. However, things get tricky if we want to rename the columns (with `names`) and also load a subset of the columns (with `usecols`). That's because passing a value to `names` means we need to use those names rather than the original ones from the file when choosing columns in `usecols`. And we can only do that if we name all the columns, which is annoying.

Actually, there is another way to do it: we can specify which columns we want by passing a list of integers to `usecols`. Pandas selects the columns at those indexes. We can then assign them names by passing a value to the `names` parameter. Here's how to do that:

```
df = pd.read_csv(filename,
                usecols=[1, 2, 3, 7, 33, 37],
                names=['pid', 'state', 'ptype',
                       'make', 'color', 'feet'])
```

Will this work? Yes, it will, and in many cases it may be the preferred way to go. However, I have two problems with it. First, I find it somewhat annoying to find the integer positions for the columns we want to load. And second, when I ran this code on my computer, I got the “low memory” warning that we’ve sometimes seen in previous examples. I thus decided to avoid the annoyance of finding the desired columns’ numeric locations and the low-memory warning and to use the two-step column renaming that appears in the solution.

With our data frame in place, we can start to perform some queries. One of the main points of this exercise is to get comfortable timing queries, to find out how quickly they run. Python provides the `timeit` module, which we can use in standard programs, but Jupyter provides a special `%timeit` magic method that can be used inside Jupyter cells. We can say

```
%timeit myfunc(2, 3, 4)
```

In this example, `timeit` runs `myfunc(2,3,4)` a number of times, reporting the mean execution time along with the variation it detects. Just how many loops `timeit` runs is determined by the code speed; something that takes a fraction of a second may run hundreds or even thousands of times, whereas something that takes more than a few seconds may be run only a handful of times.

NOTE When using the `%timeit` magic command in Jupyter, don't forget: your code must be written on a single line, just after the `%timeit` magic command. If you have more than one line, wrap it into a function and invoke that function. Or use the related `%%timeit` command, which works on an entire cell rather than a single line. Also, if you're timing a function, don't forget to put `()` after the function's name.

For the first task, I asked you to find all rows in `df` that were for parking tickets issued in New York (`'NY'`), New Jersey (`'NJ'`), or Connecticut (`'CT'`), using both the traditional `loc` accessor and the `query` method. I also asked you to time each of these for comparison.

We start with the traditional `.loc` accessor, combining three separate queries:

```
%%timeit
df.loc[(df['state'] == 'NY') |
       (df['state'] == 'NJ') |
       (df['state'] == 'CT')]
```

On my computer, this query took 1.84 seconds. (As usual, the timing will vary slightly with each run.)

Consider everything that pandas has to do for this query:

- Compare each element in `df['state']` with `'NY'`
- Compare each element in `df['state']` with `'NJ'`
- Compare each element in `df['state']` with `'CT'`
- Perform an “or” operation on the first two (New York and New Jersey) boolean series
- Perform an “or” operation on the result of this “or” and the Connecticut series
- Apply that final boolean series to `df.loc` as a mask index

There’s no doubt that with so many rows, each comparison will take some time. Moreover, the “or” operations, resulting in a single boolean series, will also take a while. Using the `query` method won’t help with the first part; we still need to perform the comparisons. However, by using `query`, we can dramatically reduce the number of “or” operations involved. That’s because `query` uses the `numexpr` backend to perform such operations, which does them far more efficiently. How much more? Here’s how we rewrite things to use `query`:

```
%timeit df.query("state == 'NY' or state == 'NJ' or state == 'CT'")
```

On my computer, using `query` took only 1.03 seconds, about 0.8 seconds (or 45%) less than the original query. That’s a pretty dramatic speed improvement and points to how much `query` can improve performance for certain queries.

However, the comparisons with each of the three state abbreviations also takes some time. Can we cut down on the number of comparisons? Yes, if we use the `isin` method on our column to search for a match within a Python list:

```
%timeit df.loc[df['state'].isin(['NY', 'NJ', 'CT'])]
```

This query took even less time than the previous one, clocking in at 0.77 seconds on my computer. That represents a 58% speedup from the original query.

But wait: maybe we can enjoy an even greater speedup if we use `query` and `isin` together. Let’s give it a try:

```
%timeit df.query('state.isin(["NY", "NJ", "CT"])')
```

Unfortunately, this didn’t seem to improve things; it took 0.80 ms on my computer—still better than the original queries, but not as good as simply using `isin`.

From this small comparison, we see that optimization of queries is rarely a matter of always using one particular technique. It requires thinking about what we're doing, considering what pandas is doing behind the scenes, and then performing some tests to check our assumptions. That said, we can conclude at least two things from these queries. First, if you're combining queries with `|` or `&`, you'll likely get a decent improvement by using `query` rather than `loc`, thanks to the speedups provided by `numexpr`. Second, using `isin` will almost always be faster than combining multiple queries because you're making a single comparison per row, rather than three.

Following this first set of queries, I asked you to perform a number of increasingly complex queries, each in three different ways: using the traditional `loc` accessor, then using `df.query`, and finally using `df.eval`. I did this not only to give you some practice building queries in different ways and comparing the time each takes but also to see that the improvements using `query` and `eval` become more pronounced as the query becomes more complex.

For starters, I asked you to find all parking tickets given to cars with New York license plates. Here are the three queries together:

```
%timeit df.loc[(df['state'] == 'NY')]
%timeit df.query('state == "NY"')
%timeit df[df.eval('state == "NY"')]
```

On my computer, these gave me timings of 903 ms, 733 ms (19% faster), and 758 ms (17% faster), respectively. We thus already see that `loc` is the slowest of the three, with the use of `df.query` and `df.eval` coming in almost the same.

But wait—the result of `df.eval` is a boolean series, which we then apply to `df`. Perhaps, instead of using a mask index on `df`, we should do so on `df.loc`. Using `%timeit`, we can find out pretty quickly:

```
%timeit df.loc[df.eval('state == "NY"')]
```

Sure enough, we get the fastest result, albeit by just a hair, when using `df.loc` here: 729 ms. In other words, it would seem that selecting via `df.loc` and a mask index gives better performance than just `df` and a mask index—something I've seen elsewhere, too. The rest of my solutions in this exercise all use `df.loc` for a fairer comparison.

Next, I asked you to find passenger cars (i.e., with `ptype` equal to `'PAS'`) from New York. Here are the three solutions:

```
%timeit df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS'))]
%timeit df.query('state == "NY" & ptype == "PAS"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS"')]
```

I got timings of 1.27 seconds for the traditional use of `df.loc` versus 965 ms for `df.query` (24% faster) and 924 ms for `df.eval` (27% faster). Here we use `&` to combine the two boolean series we get back from each comparison. Although we were able to speed up our "or" query using `isin`, there isn't an exact equivalent for "and" queries.

Next, I asked you to expand the query further, thus narrowing the potential results, looking for white passenger cars from New York that had been ticketed. Again, we can compare the queries:

```
%%timeit
df.loc[((df['state'] == 'NY') &
                (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE'))]

%%timeit
df.query(
    'state == "NY" & ptype == "PAS" & color == "WHITE"')

%%timeit
df.loc[df.eval(
    'state == "NY" & ptype == "PAS" & color == "WHITE"')]
```

This time, I got timings of 1.34 seconds, 728 ms for `df.query` (45% faster), and 727 ms for `df.eval` (also 45% faster). We can see that adding another condition slows the traditional query a bit but actually results in faster queries when using the `numexpr` backend. I'm not sure why this is the case, except that perhaps `numexpr` is only activated once the query reaches a certain size threshold.

Next, I asked you to find tickets for white passenger cars from New York that were parked more than 1 foot from the curb. Here are the queries:

```
%%timeit
df.loc[((df['state'] == 'NY') & (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') & (df['feet'] > 1))]

%%timeit
df.query('state == "NY" & ptype == "PAS" &
                  color == "WHITE" & feet > 1')

%%timeit
df.loc[df.eval('state == "NY" & ptype == "PAS" &
                    color == "WHITE" & feet > 1')]
```

In this case, I got timings of 1.31 seconds for the traditional query, 712 ms for `df.query` (45% faster), and 706 ms for `df.eval` (46% faster). Again, we can see that when the queries are complex, using the `numexpr` backend gives us a big speed advantage.

Finally, I asked you to find tickets given to white Toyota passenger cars with license plates from New York state that were parked more than 1 foot from the curb. Here is how we write those queries:

```
%%timeit
df.loc[((df['state'] == 'NY') &
                (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') &
                (df['feet'] > 1) &
                (df['make'] == 'TOYOT'))]

%%timeit
df.query('state == "NY" & ptype == "PAS" &
                  color == "WHITE" & feet > 1 &
                  make == "TOYOT"')

%%timeit
df.loc[df.eval('state == "NY" & ptype == "PAS" &
                    color == "WHITE" & feet > 1 &
                    make == "TOYOT"')]
```

I got timings of 1.75 seconds for the traditional query, 896 ms for `df.query` (49% faster), and 899 ms for `df.eval` (48% faster). The added condition slowed all the queries, but the `numexpr` backend continued to prove its worth, giving us the same answer at nearly twice the speed.

Does this mean it's always worth using `df.query` or `df.eval`? I know there are pandas users who would say "yes," given that even in the simplest of cases, we see a speedup. And in the most complex cases, the speedup is dramatic. So you could argue that because it doesn't matter much for simple queries on a short data set but it matters a lot for complex queries on large ones, you should always use these techniques.

However, focusing on speed before you've thought hard about the problem and potential bottlenecks can be misleading. Remember that `df.query` returns all the columns from a data frame—so if a data frame contains more columns than we want to get back, it may end up using lots of memory unnecessarily. By contrast, `df.loc` provides not only a row selector but also a column selector for more flexibility. I thus tend to use `df.loc` for my queries while I'm still putting them together. When I'm done, I can then experiment with these techniques to see how to reduce memory and speed things up.

Solution

```
filename = '../data/nyc-parking-violations-2020.csv'
df = pd.read_csv(filename,
                usecols=['Plate ID', 'Registration State',
                         'Plate Type', 'Feet From Curb',
                        'Vehicle Make', 'Vehicle Color'])
df.columns = ['pid', 'state', 'ptype',
              'make', 'color', 'feet']

%timeit df.loc[(df['state'] == 'NY') |
               (df['state'] == 'NJ') |
               (df['state'] == 'CT')]
%timeit df.query("state == 'NY' or
                  state == 'NJ' or
                  state == 'CT'")
%timeit df.loc[df['state'].isin(['NY', 'NJ', 'CT'])]

%timeit df.loc[(df['state'] == 'NY')]
%timeit df.query('state == "NY"')
%timeit df.loc[df.eval('state == "NY"')]

%timeit df.loc[((df['state'] == 'NY') &
                (df['ptype'] == 'PAS'))]
%timeit df.query('state == "NY" & ptype == "PAS"')
%timeit df.loc[df.eval('state == "NY" & ptype == "PAS"')]

%timeit df.loc[((df['state'] == 'NY') &
                (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE'))]

%timeit df.query(
    'state == "NY" & ptype == "PAS" & color == "WHITE"
    ')
%timeit df.loc[df.eval(
    'state == "NY" & ptype == "PAS" & color == "WHITE"')
    ]

%timeit df.loc[((df['state'] == 'NY') &
                (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') &
                (df['feet'] > 1))]

%timeit df.query(
    'state == "NY" & ptype == "PAS" & color == "WHITE" & feet > 1'
    )

%timeit df.loc[df.eval('state == "NY" & ptype == "PAS" &
                    color == "WHITE" & feet > 1')]

%timeit df.loc[((df['state'] == 'NY') &
                (df['ptype'] == 'PAS') &
                (df['color'] == 'WHITE') &
                (df['feet'] > 1) &
                (df['make'] == 'TOYOT'))]
%timeit df.query(
    ('state == "NY" & ptype == "PAS" & color == "WHITE"' +
     '& feet > 1 & make == "TOYOT"')
    )
%timeit df.loc[df.eval(
    ('state == "NY" & ptype == "PAS" & color == "WHITE"' +
        '& feet > 1 & make == "TOYOT"')
       )]
```

Beyond the exercise

In this exercise, we ran a number of queries using plain ol' `.loc` as well as `df.query` and `df.eval`, comparing their performance times. Here are some additional challenges for you to try along these lines:

- In `df.query`, you can use the words `and` and `or`, rather than the symbols `&` and `|`, thanks to the `numexpr` library. Rewrite the final query using the words. Does this change the speed at all?
- I prefer measuring distance in meters rather than feet. I thus want to find all cars that were ticketed when they were more than 1 meter from the curb. (Every 1 meter is 3.28 feet.) Perform this query using the traditional `df.loc` and also using `df.query`. Which one runs faster?
- What if you modify the query to look for cars that are more than 1 meter from the curb and for which the state is New York? Which query runs faster and by how much?

Summary

Calculations and analysis with pandas are much faster than they would be in pure Python. Even so, when you're working with a large data set, you'll often want or need to reduce the memory footprint of your data frame and use techniques that can improve performance. In this chapter, we reviewed a number of techniques you can use to speed up your queries and also use fewer resources:

- Choosing columns carefully
- Reducing memory usage with categories
- Reading data from feather format rather than CSV
- Speeding up complex queries with `df.query` and `df.eval`

If you've reached this part of the book, congratulations! You've successfully gone through all 50 exercises! I have no doubt that if you've made it this far, you have a much better, deeper understanding of pandas, what it can do, and how you can use it in a variety of situations. You should feel good about yourself and confident about your ability to use pandas at work.

But before you stop reading: the next chapter contains a large project in which I'll ask you to use all the techniques from this book to analyze a large real-world data set. I hope you'll take the time to do the project, which will help cement the lessons you've learned and help you use pandas even more effectively in the future.