2 Extracting maximum performance from built-in features

This chapter covers

- Profiling code to find speed and memory bottlenecks
- Making more efficient use of existing Python data structures
- Understanding Python's memory cost of allocating typical data structures
- Using lazy programming techniques to process large amounts of data

There are many tools and libraries to help us write more efficient Python. But before we dive into all the external options to improve performance, let's first take a closer look at how we can write pure Python code that is more efficient, in both computing and IO performance. Indeed many, although certainly not all, Python

performance problems can be solved by being more mindful of Python's limits and capabilities.

To demonstrate Python's own tools for improving performance, let's use them on a hypothetical, although realistic problem. Let's say you are a data engineer tasked with preparing the analysis of climate data around the world. The data will be based on the Integrated Surface Database from the US National Oceanic and Atmospheric Administration (NOAA; http://mng.bz/ydge). You are on a tight deadline, and you will only be able to use mostly standard Python. Furthermore, buying more processing power is out of the question due to budgetary constraints. The data will start to arrive in one month, and you plan on using the time before it arrives to increase code performance. Your task, then, is to find the places in need of optimization and to increase their performance.

The first thing that you want to do is to profile the existing code that will ingest the data. You know that the code that you already have is slow, but before you try to optimize it, you need to find empirical evidence of the bottlenecks. Profiling is important because it allows you to search, in a rigorous and systematic way, for bottlenecks in your code. The most common alternative, guesstimating, is particularly ineffective here because many slowdown points can be quite unintuitive.

Optimizing pure Python code is the low-hanging fruit, but it is also where most problems tend to reside, so optimizing can often be quite advantageous. In this chapter, we will see what pure Python offers out of the box to help us develop more performant code. We will start by profiling the code, using several profiling tools, to detect problem areas. Then we will focus on Python's basic data structures: lists, sets, and dictionaries. Our goal here will be to improve the efficiency of these data structures and to allocate memory to them in the best way for optimal performance. Finally, we will see how modern Python lazy programming techniques can help us to improve the performance of our data pipeline.

This chapter will only discuss optimizing Python without external libraries, but we will still use some external tools to help us optimize performance and access data. We will be using Snakeviz to visualize the output of Python profiling, as well as line_profiler to profile code line by line. Finally, we will use the requests library to download data from the internet.

If you use Docker, the default image has all you need. If you follow the instructions for Anaconda Python from appendix A, you are all set. Let's now start our profiling process by downloading data from weather stations and studying the temperature at each station.

2.1 Profiling applications with both IO and computing workloads

Our first objective will be to download data from a weather station and get the minimum temperature for a certain year on that station. Data on NOAA's site has CSV files, one per year and then per station. For example, the file https://www.ncei.noaa.gov/data/global-hourly/access/2021/01494099999.csv has all the entries for station

01494099999 for the year 2021. This includes, among other entries, temperature and pressure, recorded potentially several times a day.

Let's develop a script to download the data for a set of stations on an interval of years. After downloading the data of interest, we will get the minimum temperature for each station.

2.1.1 Downloading data and computing minimum temperatures

Our script will have a simple command-line interface, where we pass a list of stations and an interval of years of interest. Here is the code to parse the input (the code can be found in `02-python/sec1-io-cpu/load.py`):

```
import collections
import csv
import datetime
import sys

import requests

stations = sys.argv[1].split(",")
years = [int(year) for year in sys.argv[2].split("-")]
start_year = years[0]
end_year = years[1]
```

To ease the coding part, we will be using the requests library to get the file. Here is the code to download the data from the server:

```
TEMPLATE_URL = "https://www.ncei.noaa.gov/data/global-hourly/access/{year}/
➥ {station}.csv"
TEMPLATE_FILE = "station_{station}_{year}.csv"

def download_data(station, year):
    my_url = TEMPLATE_URL.format(station=station, year=year)
    req = requests.get(my_url)
    if req.status_code != 200:
        return  # not found
    w = open(TEMPLATE_FILE.format(station=station, year=year), "wt")
    w.write(req.text)
    w.close()


def download_all_data(stations, start_year, end_year):
    for station in stations:
        for year in range(start_year, end_year + 1):
            download_data(station, year)
```

Requests makes it easy to access web content.

This code will write each downloaded file to disk for all the requested stations across all years. Now, let's get all the temperatures in a single file:

```
def get_file_temperatures(file_name):
  with open(file_name, "rt") as f:
    reader = csv.reader(f)
```

```
        header = next(reader)
        for row in reader:
            station = row[header.index("STATION")]
            # date = datetime.datetime.fromisoformat(row[header.index('DATE')])
            tmp = row[header.index("TMP")]
            temperature, status = tmp.split(",")
            if status != "1":
                continue
            temperature = int(temperature) / 10
            yield temperature
```

We ignore entries for which the data is not available.

The format for the temperature field includes a subfield with the status quality of the data.

Let's now get all temperatures and the minimum temperature per station:

```
def get_all_temperatures(stations, start_year, end_year):
    temperatures = collections.defaultdict(list)
    for station in stations:
        for year in range(start_year, end_year + 1):
            for temperature in get_file_temperatures(
➥ TEMPLATE_FILE.format(station=station, year=year)):
                temperatures[station].append(temperature)
    return temperatures


def get_min_temperatures(all_temperatures):
    return {station: min(temperatures) for station, temperatures in
➥ all_temperatures.items()}
```

Now we can tie everything together: download the data, get all temperatures, compute the minimum per station, and print the results:

```
download_all_data(stations, start_year, end_year)
all_temperatures = get_all_temperatures(stations, start_year, end_year)
min_temperatures = get_min_temperatures(all_temperatures)
print(min_temperatures)
```

For example, to load the data for stations 01044099999 and 02293099999 for the year 2021, we do

```
python load.py 01044099999,02293099999 2021-2021
```

with the output being

```
{'01044099999': -10.0, '02293099999': -27.6}
```

Now, the real fun starts. Our goal is to continue to download lots of data from lots of stations over many years. To handle this quantity of data, we want to make the code as efficient as possible. The first step in making the code more efficient is to profile it in an organized and thorough way to find the bottlenecks slowing it down. For this, we will use Python's built-in profiling machinery.

2.1.2 Python's built-in profiling module

As we want to make sure our code is as efficient as possible, the first thing we need to do is to find existing bottlenecks in that code. Our first port of call will be profiling the code to check each function's time consumption. For this, we run the code via Python's `cProfile` module. This module is built into Python and allows us to obtain profiling information from our code. Make sure you do not use the `profile` module, as it is orders of magnitude slower; it's only useful if you are developing profiling tools yourself.

We can run the profiler with:

```
python -m cProfile -s cumulative load.py 01044099999,02293099999 2021-
     2021 > profile.txt
```

Remember that running Python with the `-m` flag will execute the module, so we are running the `cProfile` module. This is Python's recommended module to gather profiling information. We are asking for profile statistics ordered by cumulative time. The easiest way to use the module is by passing our script to the profiler in a module call like this:

```
375402 function calls (370670 primitive calls) in 3.061 seconds

   Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
158/1    0.000    0.000    3.061   3.061 {built-in method builtins.exec}
    1    0.000    0.000    3.061   3.061 load.py:1(<module>)
    1    0.001    0.001    2.768   2.768 load.py:27(download_all_data)
    2    0.001    0.000    2.766   1.383 load.py:17(download_data)
    2    0.000    0.000    2.714   1.357 api.py:64(get)
    2    0.000    0.000    2.714   1.357 api.py:16(request)
    2    0.000    0.000    2.710   1.355 sessions.py:470(request)
    2    0.000    0.000    2.704   1.352 sessions.py:626(send)
 3015    0.017    0.000    1.857   0.001 socket.py:690(readinto)
 3015    0.017    0.000    1.829   0.001 ssl.py:1230(recv_into)
[...]
    1    0.000    0.000    0.000   0.000 load.py:58(get_min_temperatures)
```

Basic summary information can be found on the first line: the number of function calls and total run time.

The computing costs of our code (computing is done in get_min_temperatures) are neglegible.

The output is ordered by cumulative time, which is all the time spent inside a certain function. Another output is the number of calls per function. For example, there is only a single call to `download_all_data` (which takes care of downloading all data), but its cumulative time is almost equal to the total time of the script. You will notice two columns called `percall`. The first one states the time spent on the function *excluding* the time spent on all the subcalls. The second one includes the time spent on subcalls. In the case of `download_all_data`, it is clear that most time is consumed by some of the subfunctions.

In many cases, when you have some intensive form of I/O like here, there is a strong possibility that I/O dominates in terms of time needed. In our case, we have

both network I/O (getting the data from NOAA) and disk I/O (writing it to disk). Network costs can vary widely, even between runs, as they are dependent on many connection points along the way. As network costs are normally the biggest time sink, let's try to mitigate those.

2.1.3 Using local caches to reduce network usage

To reduce network communication, let's save a copy for future use when we download a file for the first time. We will build a local cache of data. We will use the same code as the previous, save for the function `download_all_data` (the code can be found in `02-python/sec1-io-cpu/load_cache.py`):

```
import os
def download_all_data(stations, start_year, end_year):
  for station in stations:
    for year in range(start_year, end_year + 1):
      if not os.path.exists(TEMPLATE_FILE.format(
      ➥ station=station, year=year)):
        download_data(station, year)
```

We check whether the file already exists and only download it if not.

The first run of the code will take the same time as the previous solution, but a second run will not require any network access. For example, given the same run as the previous, it goes from 2.8 s to 0.26 s—more than an order of magnitude increase. Remember that due to high variability in network access, the time to download files can vary substantially in your case. This is yet another reason to consider caching network data: having a more predictable execution time:

```
python -m cProfile -s cumulative load_cache.py 01044099999,02293099999
➥ 2021-2021 > profile_cache.txt
```

Now, the result is different in where time is consumed:

```
299938 function calls (295246 primitive calls) in 0.260 seconds

   Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
156/1    0.000    0.000    0.260    0.260 {built-in method builtins.exec}
    1    0.000    0.000    0.260    0.260 load_cache.py:1(<module>)
    1    0.008    0.008    0.166    0.166 load_cache.py:51(
➥ get_all_temperatures)
33650    0.137    0.000    0.156    0.000 load_cache.py:36(
➥ get_file_temperatures)
[...]
    1    0.000    0.000    0.001    0.001 load_cache.py:60(
    ➥ get_min_temperatures)
```

While the time to run decreased one order of magnitude, IO is still top. Now, it's not the network but disk access. This is mostly caused by the computation being acually low.

WARNING Caches, as this example shows, can speed up code by orders of magnitude. However, cache management can be problematic and is a common source of bugs. In our example, the files never change over time, but there are many use cases for caches where the source might be changing. In that case, the cache management code needs to recognize that problem. We will revisit caches in other parts of the book.

We are now going to consider a case where CPU is the limiting factor.

2.2 Profiling code to detect performance bottlenecks

Here we look at code where CPU is the resource costing the most time in a process. We'll take all stations in the NOAA database and compute the distance between them, a problem of complexity `n2`.

In the repository, you will find a file (`02-python/sec2-cpu/locations.csv`) with all the geographical coordinates of the stations (the code can be found in `02-python/sec2-cpu/distance_cache.py`):

```
import csv
import math


def get_locations():
    with open("locations.csv", "rt") as f:
        reader = csv.reader(f)
        header = next(reader)
        for row in reader:
            station = row[header.index("STATION")]
            lat = float(row[header.index("LATITUDE")])
            lon = float(row[header.index("LONGITUDE")])
            yield station, (lat, lon)


def get_distance(p1, p2):
    lat1, lon1 = p1
    lat2, lon2 = p2

    lat_dist = math.radians(lat2 - lat1)
    lon_dist = math.radians(lon2 - lon1)
    a = (
        math.sin(lat_dist / 2) * math.sin(lat_dist / 2) +
        math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) *
        math.sin(lon_dist / 2) * math.sin(lon_dist / 2)
    )
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    earth_radius = 6371
    dist = earth_radius * c

    return dist


def get_distances(stations, locations):
```

This is the code to compute the distance between two stations.

```
    distances = {}
    for first_i in range(len(stations) - 1):
        first_station = stations[first_i]
        first_location = locations[first_station]
        for second_i in range(first_i, len(stations)):
            second_station = stations[second_i]
            second_location = locations[second_station]
            distances[(first_station, second_station)] = get_distance(
                first_location, second_location)
    return distances


locations = {station: (lat, lon) for station, (lat, lon) in get_locations()}
stations = sorted(locations.keys())
distances = get_distances(stations, locations)
```

As we are comparing all the stations between each other, the complexity is of the order n^2.

The previous code will take a long time to run. It also takes a lot of memory. If you have memory problems, limit the number of stations that you are processing. Let's now use Python's profiling infrastructure to see where most time is spent.

2.2.1 Visualizing profiling information

Again, we use Python's profiling infrastructure to find pieces of code that are delaying execution. But to better inspect the trace, we'll use an external visualization tool, SnakeViz (https://jiffyclub.github.io/snakeviz/).

We start by saving a profile trace:

```
python -m cProfile -o distance_cache.prof distance_cache.py
```

The `-o` parameter specifies the file where the profiling information will be stored. After that, we have the call to our code as usual.

> **NOTE** Python provides the `pstats` module to analyze traces written to disk. You can do `python -m pstats distance_cache.prof`, which will start a command-line interface to analyze the cost of our script. You can find more information about this module in the Python documentation or in the profiling section of chapter 5.

To analyze this information, we will use the web-based visualization tool, SnakeViz. You just need to do `snakeviz distance_cache.prof`. This will start an interactive browser window (figure 2.1 shows a screenshot).

Familiarizing yourself with the SnakeViz interface

This would be a good time to play with the SnakeViz interface a bit. For example, you can change the style from Icicle to Sunburst (arguably cuter but with less information as the file name disappears). Reorder the table at the bottom. Check the Depth and Cutoff entries. Do not forget to click some of the colored blocks, and, finally, return to the main view by clicking Call Stack and choosing the 0 entry.

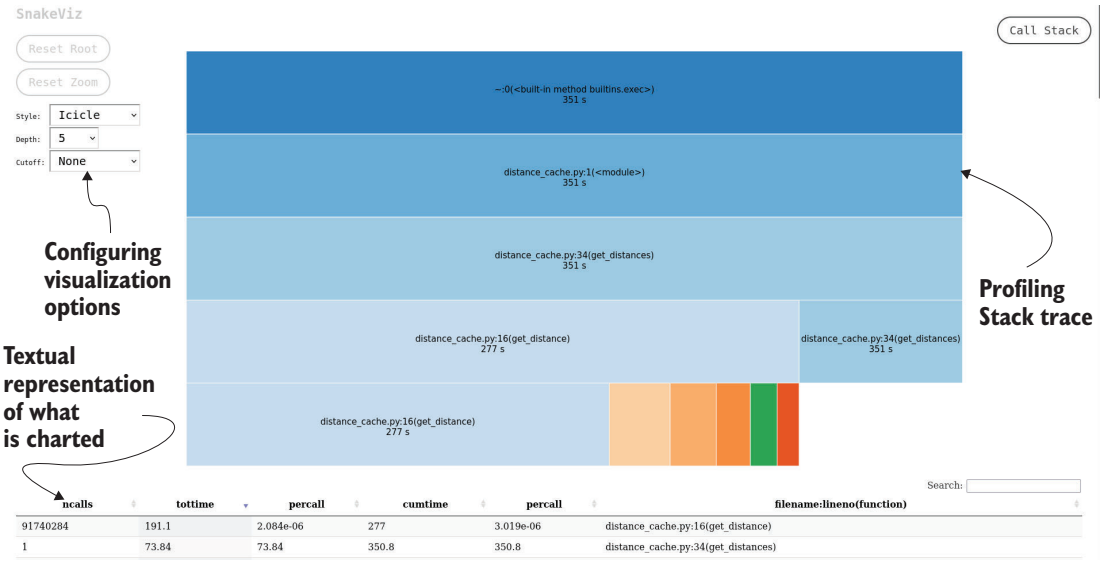

Figure 2.1 Using SnakeViz to inspect profiling information of our script

Most of the time is spent inside the function `get_distance`, but exactly where? We can see the cost of some of the math functions, but Python's profiling doesn't allow us to have a fine-grained view of what happens inside each function. We only get aggregate views for each trigonometric function. Yes, there is some time spent in `math.sin`, but given that we use it in several lines, where exactly are we paying a steep price? For that, we need to recruit the help of the line profiling module.

2.2.2 Line profiling

Built-in profiling, like we used previously, allowed us to find the piece of code that was causing a massive delay. But there are limits to what we can do with it. We'll discuss those limits here and introduce line profiling as a way to find further performance bottlenecks in our code.

To understand the cost of each line of `get_distance`, we will use the `line_profiler` package, which is available at https://github.com/pyutils/line_profiler. Using the line profiler is quite easy: you just need to add an annotation to `get_distance`:

```
@profile
def get_distance(p1, p2):
```

You might have noticed that we have not imported the `profile` annotation from anywhere. This is because we will be using the convenience script `kernprof` from the `line_profiler` package that will take care of this. Let's then run the line profiler in our code:

```
kernprof -l lprofile_distance_cache.py
```

Be prepared for the instrumentation required by the line profiler to slow the code substantially, by several orders of magnitude. Let it run for a minute or so and, after that, interrupt it (`kernprof` would probably run for many hours if you let it complete). If you interrupt it, you will still have a trace. After the profiler ends, you can have a look at the results with the command:

```
python -m line_profiler lprofile_distance_cache.py.lprof
```

If you look at the output shown in listing 2.1, you can see that it has many calls that take a long time. So we will probably want to optimize that code. At this stage, as we are discussing only profiling, we will stop here, but afterward, we would need to optimize those lines (we will do so later in this chapter). If you are interested in optimizing this piece of code, have a look at chapter 6 about Cython or appendix B on Numba as they provide the most straightforward avenues to increase the speed.

Listing 2.1 The output of the `line_profiler` package for our code

The total running time for our code

The information that we are getting for each line that is being profiled. For each line, we get the number of times the line is called, the sum of the time spent on the line, the time per call, and the percentage of time on the line.

```
Timer unit: 1e-06 s

Total time: 619.401 s
File: lprofile_distance_cache.py
Function: get_distance at line 16

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           @profile
    17                                           def get_distance(p1, p2):
    18  84753141   36675975.0      0.4      5.9      lat1, lon1 = p1
    19  84753141   35140326.0      0.4      5.7      lat2, lon2 = p2
    20
    21  84753141   39451843.0      0.5      6.4      lat_dist = math.
    ➥ radians(lat2 -lat1)
    22  84753141   38480853.0      0.5      6.2      lon_dist = math.
    ➥ adians(lon2 - lon1)
    23  84753141   28281163.0      0.3      4.6      a = (
    24 169506282   84658529.0      0.5     13.7       math.sin(lat_dist / 2)
    ➥ * math.sin(
    ➥ lat_dist / 2) +
    25 254259423  118542280.0      0.5     19.1       math.cos(math.radians(
    ➥ lat1)) * math.cos(
    ➥ math.radians(
    ➥ lat2)) *
    26 169506282   81240276.0      0.5     13.1       math.sin(lon_dist / 2)
    ➥ * math.sin(
    ➥ lon_dist / 2)
    27                                               )
    28  84753141   65457056.0      0.8     10.6      c = 2 * math.atan2(
    ➥ math.sqrt(a),
    ➥ math.sqrt(1 - a))
    29  84753141   29816074.0      0.4      4.8      earth_radius = 6371
```

```
30  84753141   33769542.0      0.4      5.5      dist = earth_radius * c
31
32  84753141   27886650.0      0.3      4.5      return dist
```

Hopefully, you will find line_profiler's output substantially more intuitive than the output from the built-in profiler.

2.2.3 The takeaway: Profiling code

As we've seen, overall built-in profiling is a big help as a first approach; it is also substantially faster than line profiling. But line profiling is significantly more informative, mostly because built-in Python profiling doesn't provide a breakdown inside the function. Instead, Python's profiling only provides cumulative values per function, as well as showing how much time is spent on subcalls. In specific cases, it is possible to know if a subcall belongs to another function, but, in general, that is not possible. An overall strategy for profiling needs to take all this into account.

The strategy we used here is a generally sensible approach: first, try the built-in Python profiling module `cProfile` because it is fast and does provide some high-level information. If that is not enough, use line profiling, which is more informative but also slower. Remember, here we are mostly concerned with locating bottlenecks; later chapters will provide ways to optimize the code. Sometimes just changing parts of an existing solution is not enough and a general re-architecturing will be necessary; we will also discuss that in due time.

Other profiling tools

Many other utilities can be useful if you are profiling code, but a profiling section would not be complete without a reference to one of these, the `timeit` module. This is probably the most common approach that newcomers take to profile code and you can find endless examples using the `timeit` module on the Internet. The easiest way to use the `timeit` module is by using IPython or Jupyter Notebook, as these systems make `timeit` very streamlined. Just add the `%timeit` magic to what you want to profile, for example, inside iPython:

```
In [1]: %timeit list(range(1000000))
27.4 ms ± 72.5 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [2]: %timeit range(1000000)
189 ns ± 22.6 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops
➥ each)
```

This gives you the run time of several runs of the function that you are profiling. The magic will decide how many times to run and report basic statistical information. In the previous snippet, you have the difference between a `range(1000000)` and a `list(range(1000000))`. In this specific case, `timeit` shows that the lazy version of `range` is two orders of magnitude faster than the eager one.

(continued)
You will be able to find more details in the documentation of the `timeit` module, but for most use cases, the `%timeit` magic of IPython will be enough to access its functionality. You are encouraged to use IPython and its magic, but in most of the rest of the book, we will use the standard interpreter. You can read more about the `%timeit` magic here: https://ipython.readthedocs.io/en/stable/interactive/magics.html.

Now that you are familiar with both a toolset and an approach to profiling, let's direct our attention to a different subject: optimizing the usage of Python data structures.

2.3 Optimizing basic data structures for speed: Lists, sets, and dictionaries

Next, we will try to find inefficient uses of Python basic data structures and rewrite pieces of code more efficiently. To demonstrate this process, we will continue to use the temperature data from NOAA. But here our challenge is to determine whether certain temperatures occurred in a station during a specified time interval.

We will reuse the code from the first section of the chapter to read the data (the code can be found in `02-python/sec3-basic-ds/exists_temperature.py`). What we are interested in, for the sake of this example, is the data from station 01044099999 for the years 2005 to 2021:

```
stations = ['01044099999']
start_year = 2005
end_year = 2021
download_all_data(stations, start_year, end_year)
all_temperatures = get_all_temperatures(stations, start_year, end_year)
first_all_temperatures = all_temperatures[stations[0]]
```

`first_all_temperatures` has a list of temperatures for the station. We can get some basic stats with `print(len(first_all_temperatures), max(first_all_temperatures), min(first_all_temperatures))`. We have 141,082 entries with a maximum of 27.0 C and a minimum of -16.0 C.

2.3.1 Performance of list searches

Checking whether a temperature is in the list is a matter of `temperature in first_all_temperatures`. Let's get a rough estimate of how much time it takes to check whether -10.7 is in the list:

```
%timeit (-10.7 in first_all_temperatures)
```

The output on my computer is:

```
313 µs ± 6.39 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Let's now try this query with a value that we know is not on the list:

```
%timeit (-100 in first_all_temperatures))
```

The result is:

```
2.87 ms ± 20.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This is roughly one order of magnitude slower than our search for -10.7.

Why such low performance in the second search? Because to complete this search, the `in` operator does a sequential scan starting from the beginning of the list. This approach means, in a worst-case scenario, that the entire list will be searched, which is exactly the case when the element that we are looking for (-100) is *not* on the list. For small lists, it adds a trivial amount of time to start the search at the top and go straight through. But as the list grows, as well as the number of searches that you might have to do on those ever-growing lists, the time adds up significantly.

At this stage, we have no numbers to compare against, but it's safe to assume that ranges between a millisecond and even a microsecond are not very encouraging. This should be doable in orders-of-magnitude less time.

2.3.2 Searching using sets

Let's see whether we can do better by switching our data structure from lists to sets. Let's convert our ordered list into a set and try to do a search

```
set_first_all_temperatures = set(first_all_temperatures)

%timeit (-10.7 in set_first_all_temperatures)
%timeit (-100 in set_first_all_temperatures)
```

with the time costs being

```
62.1 ns ± 3.27 ns per loop (mean ± std. dev. of 7 runs,
➥ 10,000,000 loops each)
26.6 ns ± 0.115 ns per loop (mean ± std. dev. of 7 runs,
➥ 10,000,000 loops each)
```

This is several orders of magnitude faster than the solutions in the previous section! But why such an improvement? There are two main reasons: one is related to set size and another is related to complexity. The complexity part will be discussed in the next subsection. Here we'll look at the role of set size.

With regards to the size, remember that the original list had 141,082 elements. But with a set, all repeated values are collapsed into a single value—and there are plenty of repeated elements on the original list. The set size is reduced to `print(len(set_first_all_temperatures))`, which is 400 elements (350 times fewer). No wonder searching is so much faster as the size of the structure is much smaller.

The takeaway is that we should be aware of possible repeated elements in a list and know that there are potential advantages of using sets so the search can happen on much smaller data structures. But there is also a more profound difference between the implementation of lists and sets in Python.

2.3.3 List, set, and dictionary complexity in Python

The improved performance from the previous example was mostly due to the de facto reduction in the size of the data structure when we switched from a list to a set. This begs the question: what would happen if there was no repetition, so both the list and the set were the same size? Let's find out. We can simulate this with a range, which will specify that all elements will be different:

```
a_list_range = list(range(100000))
a_set_range = set(a_list_range)

%timeit 50000 in a_list_range
%timeit 50000 in a_set_range
%timeit 500000 in a_list_range
%timeit 500000 in a_set_range
```

So now we have a range of 0 to 99,999 that is implemented as both a list and a set. We search both data structures for 50,000 and 500,000. Here are the timings:

```
455 µs ± 2.68 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
40.1 ns ± 0.115 ns per loop (mean ± std. dev. of 7 runs,
➥ 10,000,000 loops each)
936 µs ± 9.37 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
28.1 ns ± 0.107 ns per loop (mean ± std. dev. of 7 runs,
➥ 10,000,000 loops each)
```

The set implementation still has much better performance. That is because in Python (to be more precise, CPython) a set is implemented with a hash. Finding an element thus has the cost of searching a hash. Hash functions come in many flavors and have to deal with many design problems. But when comparing lists and sets, we can generally assume that set lookup is mostly constant and will perform as well with a collection of size 10 or 10 million. This is not actually correct, but it is reasonable for understanding, in an intuitive way, why set lookups compare favorably against list lookups.

Remember also that a set is usually implemented like a dictionary, without values, which means that when you search on a dictionary key, you get the same performance as searching on a set. However, sets and dictionaries are not the silver bullets that they might seem here. For example, if you want to search an interval, an ordered list is substantially more efficient. In an ordered list, you can find the lowest element and then traverse from that point up until you find the first element above the interval and then stop. In a set or dictionary, you would have to do a lookup for each element in the interval. So if you know the value you are searching for, then a dictionary can be extremely fast. But if you are looking in an interval, then it suddenly stops being a reasonable option; an ordered list with a bisection algorithm would perform much better.

Given that lists are so pervasive and easy to use in Python, there are many cases where more appropriate data structures exist. But it is worth stressing that lists are a fundamental data structure that has many good use cases. The point is to be mindful of your options, not to banish lists.

TIP Be careful when using `in` to search inside large lists. If you browse through Python code, the pattern of using `in` to find elements in a list (the method `index` of list objects is, in practice, the same thing) is quite common. This is not a problem for small lists as the time penalty is quite small and perfectly reasonable, but it can be serious with large lists.

From a very down-to-earth software engineering perspective, the use of `in` with lists can go from an unnoticed problem in development to a massive problem in production. The common pattern is a developer testing with small data examples, because feeding big data is normally not practical with most unit testing. The real data might be very large, however, and once it's introduced, it could bring a production system to a halt.

A more systematic solution would be to test the code—maybe not always but at least from time to time—with very large data sets. This can occur in different stages of testing, from unit to end-to-end testing. This should not be construed as an argument against using `in` with lists. Just be mindful of the discrepancies between performance during development and production due to data size.

By the way, for most searching operations, there is a substantially better family of data structures than lists, sets, or dictionaries: trees. But in this chapter, we are evaluating Python's built-in data structures, which do not include trees.

The whole topic of choosing appropriate algorithms and data structures is the subject of many books and often makes up some of the most difficult courses for a computer science degree. The point is not to have an exhaustive discussion of the topic but to make you aware of the most common alternatives in Python. If you believe existing Python data structures are not enough for your needs, you may want to consider other types of data structures. This book's focus is on Python, but other resources will cover data structures outside of Python; for example, *Data Structures and Algorithms in Python*, by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser (Wiley 2013), provides a good introduction.

Another helpful resource is Python's own data on TimeComplexity (https://wiki.python.org/moin/TimeComplexity). Here you can look up the time complexity of a wide range of operations over many Python data structures.

So far in this chapter we have focused on time performance. But that is not the only factor when dealing with performance problems with large data sets. Let's turn to another important factor: conserving memory.

2.4 Finding excessive memory allocation

Memory consumption can be crucial for performance, and it's not just that you might run out of memory. Effective memory allocation can allow for more processes to be run in parallel on the same machine. Even more significantly, judicious memory use might allow for in-memory algorithms.

Let's return to our familiar scenario, the NOAA database, to see how we can reduce the disk consumption of our data. To do this, we will start with a study of the content of the data files. Our objective here is to load a few of those files and do some statistics on character distributions.

```
def download_all_data(stations, start_year, end_year):
    for station in stations:
        for year in range(start_year, end_year + 1):
            if not os.path.exists(TEMPLATE_FILE.format(
            ➥ station=station, year=year)):
                download_data(station, year)


def get_all_files(stations, start_year, end_year):
    all_files = collections.defaultdict(list)
    for station in stations:
        for year in range(start_year, end_year + 1):
            f = open(TEMPLATE_FILE.format(station=station, year=year), 'rb')
            content = list(f.read())
            all_files[station].append(content)
            f.close()
    return all_files


stations = ['01044099999']
start_year = 2005
end_year = 2021
download_all_data(stations, start_year, end_year)
all_files = get_all_files(stations, start_year, end_year)
```

`all_files` now has a dictionary where each item contains the contents for all the files related to a station. Let's study the memory usage of this.

2.4.1 Navigating the minefield of Python memory estimation

Python provides a function in the `sys` module, `getsizeof`, that supposedly returns the memory occupied by an object. We can get an understanding of the memory occupied by our dictionary using the following code:

```
print(sys.getsizeof(all_files))
print(sys.getsizeof(all_files.values()))
print(sys.getsizeof(list(all_files.values())))
```

The result is:

```
240
40
64
```

`getsizeof` might not return what you expect. The files on the disk are in the megabyte range, so estimates below 1 KB sound quite suspicious. `getsizeof` is actually returning the size of the containers (the first is a dictionary, the second is an iterator, and the third is a list) *without* accounting for the content. So, we have to account for two things occupying memory: the content of the container and the container itself.

> **NOTE** Note that there is no problem with the `getsizeof` implementation in the language; it is just that the expectation of an unsuspecting user is typically of something different—namely, that it would return the memory footprint of everything referred in the object. If you read the official documentation, you will even find instructions for a recursive implementation that solves most problems. For us, the intricacies of `getsizeof` are mostly a starting point to discuss CPython memory allocation in depth.

Let's get some basic information about our station data:

```
station_content = all_files[stations[0]]
print(len(station_content))
print(sys.getsizeof(station_content))
```

The output is:

```
17
248
```

Our dictionary has only one entry, corresponding to a single station. It contains a list with 17 entries. The list itself takes 248 bytes, but remember, that doesn't include the content. Now let's inspect the size of the first entry:

```
print(len(station_content[0]))
print(sys.getsizeof(station_content[0]))
print(type(station_content[0]))
```

The length is 1,303,981, corresponding to the size of the file. We have a `getsizeof` of 10,431,904. This is around eight times the size of the underlying file. Why eight times? Because each entry is a pointer to a character, and a pointer is 8 bytes in size. At this stage, this looks quite bad, as we have a large data structure, and we haven't yet accounted for the data proper. Let's have a look at a single character:

```
print(sys.getsizeof(station_content[0]))
print(type(station_content[0]))
```

This is colossal in size. The output is 28 with a type of `int`. So every character, which should take only one 1 byte, is represented by 28 bytes. Hence, we have 10,431,904 for the size of the list plus 28 * 1,303,981 (36,511,468) for a grand total of 46,943,372. This is 36 times bigger than the original file! Fortunately, the situation is not as bad as it seems, but we can do much better. We will start by seeing that Python (or rather, CPython) is quite smart with memory allocation.

CPython can allocate objects in a more sophisticated way, and it turns out that our approach to computing memory allocation is quite naive. Let's compute the size of only the inner content, but instead of going through all the integers in our matrix, we will make sure that we are not double-counting. In Python, if an object is used many times, it gets the same `id`. So if we see the same `id` many times, we should only count a single memory allocation:

```
single_file_data = station_content[0]
all_ids = set()
for entry in single_file_data:
    all_ids.add(id(entry))
print(len(all_ids))
```

The id function allows us to get the unique ID of an object.

The previous code gets the unique identifier for all of our numbers. In CPython, that happens to be memory location. CPython is smart enough to see that the same string content is being used over and over again—remember that each ASCII character is represented by an integer between 0 and 127—and, as such, the output of the previous code is 46.

So, dumb allocation of memory would be dreadful, but Python (or, to be more precise, CPython) is much smarter. The memory cost of this solution is just the list infrastructure (10,431,904). Note that in our case, we only have 46 distinct characters; with such a small subset, Python is quite good at smart memory allocation. Do not expect this best-case scenario to occur at all times because it will depend on your data pattern.

Object caching and reuse in Python

Python tries to be as smart as possible with object reuse, but we need to be careful with expectations. The first reason is that this is *implementation-dependent*. CPython is different from other Python implementations in terms of this behavior.

Another reason is that even CPython makes no promises about most of its allocation policies from version to version. What works for your specific version might change in a different version.

Finally, even if you have a fixed version, how things work might not be completely obvious. Consider this code in Python 3.7.3 (this might vary on other versions):

```
s1 = 'a' * 2
s2 = 'a' * 2
s = 2
s3 = 'a' * s
s4 = 'a' * s
print(id(s1))
print(id(s2))
print(id(s3))
print(id(s4))
print(s1 == s4)
```

Here we are getting the string aa by multiplying a times 2.

Here we are getting the string aa by multiplying a times s which is 2.

All these strings are equal in terms of content.

The result will be:

```
140002256425568
140002256425568
140002256425904
140002256425960
True
```

With the size of the string as a variable, the allocator is not able to determine that the content is the same, even if the size is the same. If this simple example works like this, what about more complicated cases? Of course, you can still use knowledge of how the allocator works, and for code, you have control over the Python version, this makes special sense. But adjust your expectations accordingly.

We are using a file representation based on a list of numbers. What if we considered alternative representations?

2.4.2 The memory footprint of some alternative representations

We are now going to consider some simple alternatives to representing a file. Some will be better; some will be worse. The main point here is to understand the underlying cost of each alternative. Instead of using integers to represent each character, we could use strings of length 1—something like this:

```
single_file_str_list = [chr(i) for i in single_file_data]
```

This approach would even be worse than the one that we used before. Just look at the size of each string with a single character:

```
print(sys.getsizeof(single_file_str_list[0]))
```

This returns 50, whereas the representation for our previous integer representation was only 28. This is a step backward, so we won't be doing it.

Python object overheads are quite bad with lots of small objects. Why do small numbers require 28 bytes and single character strings, 50 bytes? It turns out that every Python object requires at least 24 bytes of overhead, and to that overhead, you have to add the overhead of the object type, which will vary from type to type. As we have seen, it's bigger for strings than byte arrays (figure 2.2).

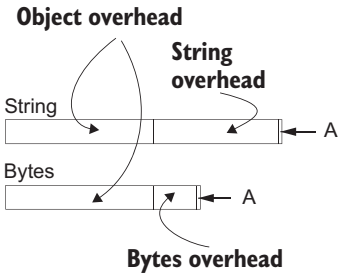

Figure 2.2 Object overhead for strings and bytes

The internal representation of strings and numbers

Python has an efficient internal representation for strings, which can vary and thus confuse expectations about memory allocation:

```
from sys import getsizeof
getsizeof('')
getsizeof('c')
getsizeof('c' * 10000)
getsizeof('ç' * 10000)
getsizeof('ç')
getsizeof('😐')
getsizeof('😐' * 10000)
```

The output would be:

```
49
50
10049
10073
74
80
40076
```

The empty string takes 49 bytes; the `c` string takes 50; 10,000 `c`'s take 10049 bytes. So far so good. But a c with a cedilla takes 74 and 10,000 `ç`'s take 10,073. If you are a bit confused now, know that a single confused smiley takes 80 bytes and 10,000 of those take 40,076 bytes.

Python 3 strings represent Unicode characters, but there is a nuance: the internal representation is optimized as a function of the string being represented. For details, you can see PEP 393–*Flexible String Representation*. For Latin-1 characters (a superset of ASCII), Python uses 1 byte (the c with a cedilla is part of that set), but for other types of characters it can take up to 4 bytes (in the case of our confused emoji). But from our perspective, string sizes are difficult to calculate.

Integers have also an optimized implementation. The precision is arbitrary, but for signed integers fitting 30 bits, we get the smaller representation possibility of 28 bytes (the number 0 is an exception; it is represented by 24 bytes only, which you might recall is the smallest object size possible due to CPython's object overhead).

There is a more obvious representation for a file: instead of using a list of one-character strings, we can use *a string with the whole file*:

```
single_file_str = ''.join(single_file_str_list)
print(sys.getsizeof(single_file_str))
```

This will be a size of 1,304,030—the size of our file plus the string object overhead. While this is an obvious and simple solution, we will continue with the approach of containers of sequences of bytes because, as it turns out, those approaches can still be improved.

2.4.3 Using arrays as a compact representation alternative to lists

Here we will look at how an alternative container to elements might be substantially more efficient in terms of memory: arrays. Let's revisit the implementation of our `get_all_files` function:

```
def get_all_files_clean(stations, start_year, end_year):
    all_files = collections.defaultdict(list)
    for station in stations:
        for year in range(start_year, end_year + 1):
            f = open(TEMPLATE_FILE.format(station=station, year=year), 'rb')
            content = f.read()
            all_files[station].append(content)
            f.close()
    return all_files
```

The original implementation had content = list(f.read()).

The line `content = list(f.read())` was converting the output of the `read` function into a list. Now, we implemented it without the list call, returning a byte array. Let's check the object size:

```
print(type(single_file_data))
print(sys.getsizeof(single_file_data))
```

The type is `bytes`, and the size including data is 1,304,014.

Arrays are of fixed size and can only contain objects of the same type. Hence, their representation can be made much more compact: it can be stored with the object overhead. Recall that for our integers, there was a 28-byte size for a storage of, really, a single byte of data.

> **Memory occupation in lists**
>
> When you allocate a list, Python creates an extra space for potential future additions, so the list will normally have more space than you expect. This makes insertions substantially cheaper because there is no need to allocate memory every time a new element is added—just when the extra space allocated is exhausted. The cost, of course, is the memory overhead. As a rule, such overhead is not serious unless you have lots of tiny lists; that is, the "lots of tiny objects" argument is especially true for lists. While knowing this is interesting, the overhead is normally OK with all other cases.

Much of the code related to array management is available in the `array` module. Except for this chapter, however, we won't be using the `array` module anymore; instead, we will use NumPy, which supersedes it in many ways. But the point here has less to do with the module and more with understanding and getting rid of the object overhead.

At this stage, you should have an insight into the costs and pitfalls of object memory allocation in Python. Finally, we'll now try to understand how to compute the memory usage of Python objects.

2.4.4 Systematizing what we have learned: Estimating memory usage of Python objects

At this stage, you have the basis to understand how memory allocation works. Now that you have a grasp of the underlying principles, we will try to devise some code to allow us to gather all the knowledge of the previous section into a utility function that gives a good approximation of the memory footprint.

We'll now distill all the tidbits that we learned in the rest of the section. In the following discussion, we'll write a function that will return the estimated memory size of an object. It will return both the size of all the objects along with the expenditure on containers. If you look at the following code, you should be able to find ID tracking, container counting (including mapper objects like dictionaries where we need to track both the key and the value), and string and array management.

Computing the size of general objects is a veritable minefield (it is actually not possible in general for external objects using Python only approaches). Our code in listing 2.2 tries to be smart by not double-counting repeated objects and containers/iterators that report the full size of the container and the content (like strings or arrays; the code can be found in `02-python/sec4-memory/compute_allocation.py`).

Listing 2.2 Computing the size of general Python objects

```
from array import array
from collections.abc import Iterable, Mapping
from sys import getsizeof
from types import GeneratorType


def compute_allocation(obj):
    my_ids = set([id(obj)])
    to_compute = [obj]
    allocation_size = 0
    container_allocation = 0
    while len(to_compute) > 0:
        obj_to_check = to_compute.pop()
        allocation_size += getsizeof(obj_to_check)
        if type(obj_to_check) in [str, array]:
            continue
        elif isinstance(obj_to_check, GeneratorType):
            continue
        elif isinstance(obj_to_check, Mapping):
            container_allocation += getsizeof(obj_to_check)
            for ikey, ivalue in obj_to_check.items():
                if id(ikey) not in my_ids:
                    my_ids.add(id(ikey))
                    to_compute.append(id(ikey))
                if id(ivalue) not in my_ids:
                    my_ids.add(ivalue)
                    to_compute.append(id(ivalue))
```

We need to store the IDs of previously seen objects to avoid double-counting them.

We will also return the memory spent in containers like lists or dictionaries.

Strings and arrays are iterables that return the size of their content; we do not want to double-count the content.

We will ignore the content of generators.

For maps, we will need to count the keys and the values.

```
    elif isinstance(obj_to_check, Iterable):
        container_allocation += getsizeof(obj_to_check)
        for inner in obj_to_check:
            if id(inner) not in my_ids:
                my_ids.add(id(inner))
                to_compute.append(inner)
return allocation_size, allocation_size - container_allocation
```

Finally, for other iterators, we will need to check the size.

Here we are using an iterative approach to compute memory allocation. This is a type of algorithm that would have lent itself to a recursive implementation, but due to Python's lack of proper support for good tail call optimization and recursive implementations in general, we will use an iterative approach.

Computing the size of objects from external libraries that are implemented in a system programming language like C or Rust will mostly depend on the implementation making that information available in some form. For those libraries, consult the documentation for details.

> **WARNING** There are memory profiler libraries for Python that you could try to use instead. I have a mixed experience with the reliability of estimates from some of the tools available, which is not shocking due to the minefield that is memory estimation in Python. If you use them, be careful.
>
> There are more lower-level ways to check the memory allocation of Python, but we will discuss those when we use NumPy. In this chapter, we restrict ourselves to Python without external libraries.

2.4.5 The takeaway: Estimating memory usage of Python objects

To summarize, estimating the size of memory objects is not as easy as one might expect. `sys.getsizeof` doesn't report all the object sizes, and as such, extra effort is needed to accurately compute object sizes. In the general case, the problem is not even solvable: libraries written in low-level languages might not report the size of the allocations that they do.

Lean memory allocation has several side advantages. One is allowing the run of more parallel processes in cases where memory is the limiting factor, as it sometimes is. Another advantage is that it may create room for using in-memory algorithms, which are faster than algorithms, which need disk space and are much slower due to disk access.

2.5 Using laziness and generators for big-data pipelining

We now shift our attention to a feature that was extensively introduced with Python 3: lazy semantics. Lazy semantics delays any computation until the data is required and not before. This is extremely helpful to process large amounts of data, as sometimes computation (and related memory allocation) doesn't need to be done or can be spread over time. If you use generators, you are using lazy semantics already. Python 3

is way lazier than Python 2 as functions like `range`, `map`, and `zip` became lazy. A lazy approach will allow you to process more data, typically with substantially less memory, and permit the creation of data pipelines inside the code in a much easier way.

2.5.1 Using generators instead of standard functions

Let's revisit the original code of the first section of this chapter:

```
def get_file_temperatures(file_name):
    with open(file_name, "rt") as f:
        reader = csv.reader(f)
        header = next(reader)
        for row in reader:
            station = row[header.index("STATION")]
            # date = datetime.datetime.fromisoformat(
            ➥ row[header.index('DATE')])
            tmp = row[header.index("TMP")]
            temperature, status = tmp.split(",")
            if status != "1":
                continue
            temperature = int(temperature) / 10
            yield temperature
```

A yield in a definition indicates a generator.

`get_file_temperatures` is a generator (notice the `yield`). Let's run the generator:

```
temperatures = get_file_temperatures(TEMPLATE_FILE.format(
➥ station="01044099999", year=2021))

print(type(temperatures))
print(sys.getsizeof(temperatures))
```

The type reported will be `generator`, and the size of the structure will be 112. In reality, not much was done as generators are lazy. Only when you start iterating through it will the code execute as needed:

```
for temperature in temperatures:
    print(temperature)
```

Every time the for loop is repeated, the generator code will be called to provide a new value.

There are several advantages of this approach. The first, and biggest, one is that you will not need to have memory allocated for all temperatures as each one will be processed in turn. Contrast this with a list where you need memory to maintain all the temperatures at the same time. This can be quite important when a function returns very large data structures with many elements—the difference between having enough memory to execute the code or not.

Second, sometimes we do not need to get all the results, and as such, being eager just spends time in useless computation. Imagine, for example, that you wanted to write a function to see whether there is at least one temperature below zero. You don't need to get all results: computation can stop as soon as a single value is below zero.

It's quite trivial to make an eager version of a generator, as simple as:

```
temperatures = list(temperatures)
```

In this case, you lose the advantage of generators, but there are situations where that might be useful. For example, when the compute time is not long and the memory used by the list representation is tolerable, then in circumstances where you need to visit the results many times, an eager version makes more sense.

> **NOTE** One of the biggest differences between Python 2 and Python 3 is that many built-ins that were eager became lazy. For example, in our case, `zip`, `map`, and `filter` would behave in very different ways in Python 2

Generators can be used to reduce the memory footprint and, in some cases, compute time. So when you are writing code that returns sequences, ask yourself whether it makes sense to convert it to a generator.

Summary

- Detection of performance bottlenecks is not easy to do in an intuitive, nonempirical way. Profiling is the necessary first step to be able to find exactly where performance lacks. "Gut feelings" tend to be wrong when finding performance problems, and empirical approaches almost always win.
- Python's internal profiling system is very useful, but it is sometimes difficult to interpret. Visualization tools like SnakeViz can help us make sense of profiling information.
- Python's internal profiling system has substantial limitations in helping us find the exact spot where a bottleneck occurs. Tools like line_profiler can be substantially more precise at the expense of running very slowly to collect information.
- While CPU performance is typically our first port of call for performance optimization, memory usage is equally important and can have major indirect benefits. For example, a solution that has poor memory optimization and requires an out-of-memory algorithm may sometimes be replaced with a fully in-memory approach, producing enormous time gains.
- Python provides basic data structures that can be used and misused to affect performance. For example, searching for elements in unordered lists can become quite expensive. We have to be mindful of the complexity cost of many operations over Python's basic data structures. These data structures appear in all Python programs and are typically low-hanging fruit that can have a massive effect on performance.
- Having a basic understanding of the computational complexity—Big-O notation—of Python's data structures is crucial for writing efficient code. Be sure to check these from time to time as Python versions change, and sometimes the underlying implementation is replaced, thus changing the performance of the algorithm.

- Lazy programming techniques allow us to develop programs that tend to have smaller memory footprints. They sometimes also make it possible to outright avoid large parts of a computation.
- All the content of this chapter is of wide applicability—both profiling and pure Python optimizations—and it can be used before any techniques discussed in the rest of the book.