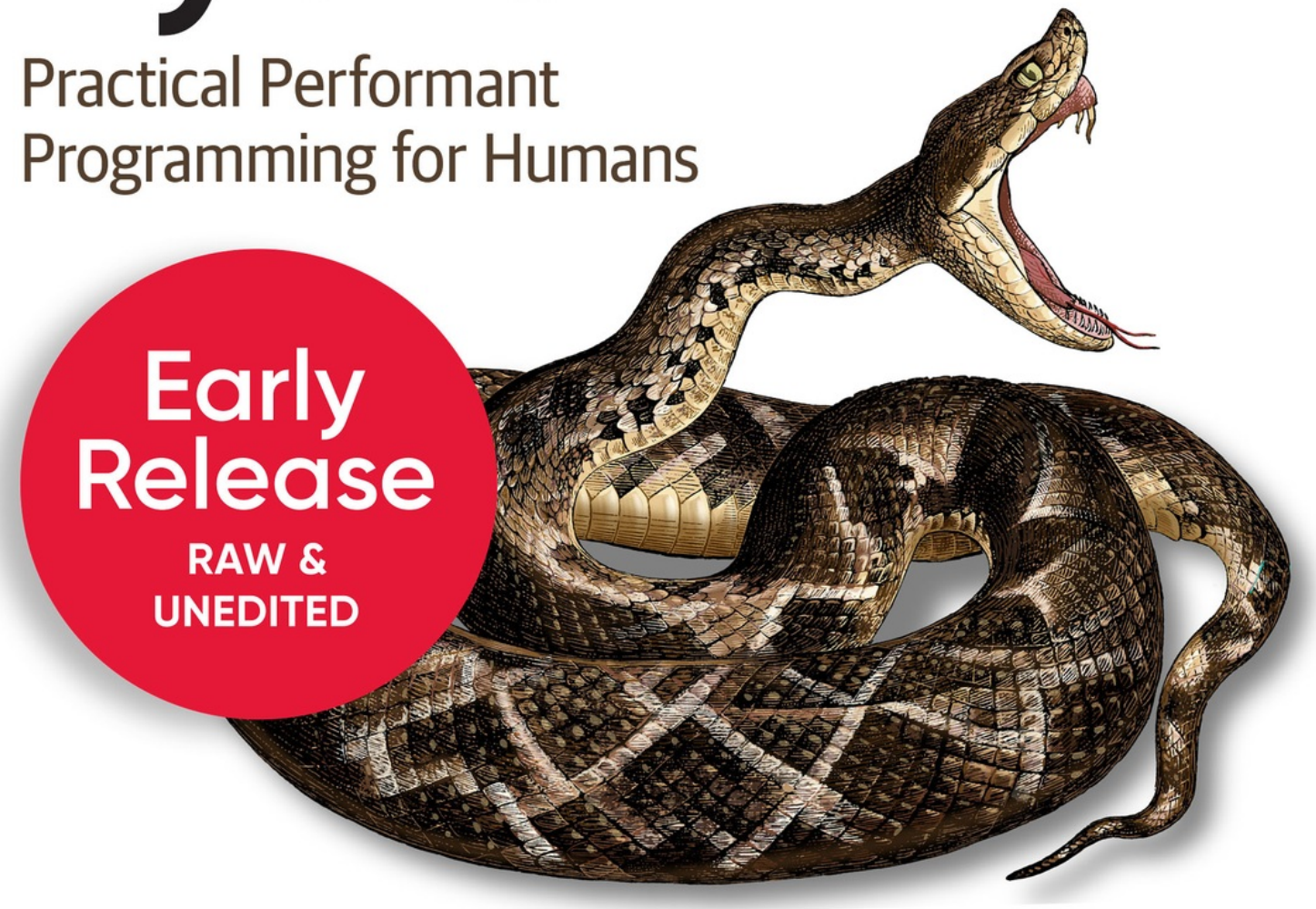
O'REILLY®

Third Edition

High Performance Python

Practical Performant Programming for Humans



Early Release

RAW & UNEDITED

Micha Gorelick & Ian Ozsvald

High Performance Python

THIRD EDITION

Practical Performant Programming for Humans

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Micha Gorelick and Ian Ozsvald

Beijing • Boston • Farnham • Sebastopol • Tokyo

High Performance Python

by Micha Gorelick and Ian Oszvald

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

- Editors: Sara Hunter and Brian Guerin
- Production Editor: Clare Laylock
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- May 2025: Third Edition

Revision History for the Early Release

- 2024-07-26: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098165963 for release details.

978-1-098-16590-1

[TO COME]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Understanding Performant Python

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

QUESTIONS YOU'LL BE ABLE TO ANSWER AFTER THIS CHAPTER

- What are the elements of a computer's architecture?
- What are some common alternate computer architectures?
- How does Python abstract the underlying computer architecture?
- What are some of the hurdles to making performant Python code?
- What strategies can help you become a highly performant programmer?

Programming computers can be thought of as moving bits of data and transforming them in special ways to achieve a particular result. However, these actions have a time cost. Consequently, *high performance programming* can be thought of as the act of minimizing these operations either by reducing the overhead (i.e., writing more efficient code) or by changing the way that we do these operations to make each one more meaningful (i.e., finding a more suitable algorithm).

Let's focus on reducing the overhead in code in order to gain more insight into the actual hardware on which we are moving these bits. This may seem like a futile exercise, since Python works quite hard to abstract away direct interactions with the hardware. However, by understanding both the best way that bits can be moved in the real hardware and the ways that Python's abstractions force your bits to move, you can make progress toward writing high performance programs in Python.

The Fundamental Computer System

The underlying components that make up a computer can be simplified into three basic parts: the computing units, the memory units, and the connections between them. In addition, each of these units has different properties that we can use to understand them. The computational unit has the property of how many computations it can do per second, the memory unit has the properties of how much data it can hold and how fast we can

read from and write to it, and finally, the connections have the property of how fast they can move data from one place to another.

Using these building blocks, we can talk about a standard workstation at multiple levels of sophistication. For example, the standard workstation can be thought of as having a central processing unit (CPU) as the computational unit, connected to both the random access memory (RAM) and the hard drive as two separate memory units (each having different capacities and read/write speeds), and finally a bus that provides the connections between all of these parts. However, we can also go into more detail and see that the CPU itself has several memory units in it: the L1, L2, and sometimes even the L3 and L4 cache, which have small capacities but very fast speeds (from several kilobytes to a dozen megabytes). Furthermore, new computer architectures generally come with new configurations (for example, Intel's SkyLake CPUs replaced the frontside bus with the Intel Ultra Path Interconnect and restructured many connections). Finally, in both of these approximations of a workstation we have neglected the network connection, which is effectively a very slow connection to potentially many other computing and memory units!

To help untangle these various intricacies, let's go over a brief description of these fundamental blocks.

Computing Units

The *computing unit* of a computer is the centerpiece of its usefulness—it provides the ability to transform any bits it receives into other bits or to change the state of the current process. CPUs are the most commonly used computing unit; however, graphics processing units (GPUs) are gaining popularity as auxiliary computing units. They were originally used to speed up computer graphics but are becoming more applicable for numerical applications and are useful thanks to their intrinsically parallel nature, which allows many calculations to happen simultaneously. Regardless of its type, a computing unit takes in a series of bits (for example, bits representing numbers) and outputs another set of bits (for example, bits representing the sum of those numbers). In addition to the basic arithmetic operations on integers and real numbers and bitwise operations on binary numbers, some computing units also provide very specialized operations, such as the "fused multiply add" operation, which takes in three numbers, `A`, `B`, and `C`, and returns the value `A * B + C`.

The main properties of interest in a computing unit are the number of operations it can do in one cycle and the number of cycles it can do in one second. The first value is measured by its instructions per cycle (IPC),[1] while the latter value is measured by its clock speed. These two measures are always competing with each other when new computing units are being made. For example, the Intel Core series has a very high IPC but a lower clock speed, while the Pentium 4 chip has the reverse. GPUs, on the other hand, have a very high IPC and clock speed, but they suffer from other

problems like the slow communications that we discuss in "Communications Layers".

Furthermore, although increasing clock speed almost immediately speeds up all programs running on that computational unit (because they are able to do more calculations per second), having a higher IPC can also drastically affect computing by changing the level of *vectorization* that is possible. Vectorization occurs when a CPU is provided with multiple pieces of data at a time and is able to operate on all of them at once. This sort of CPU instruction is known as single instruction, multiple data (SIMD).

In general, computing units have advanced quite slowly over the past decade (see Figure 1-1). Clock speeds and IPC have both been stagnant because of the physical limitations of making transistors smaller and smaller. As a result, chip manufacturers have been relying on other methods to gain more speed, including simultaneous multithreading (where multiple threads can run at once), more clever out-of-order execution, and multicore architectures.

Hyperthreading presents a virtual second CPU to the host operating system (OS), and clever hardware logic tries to interleave two threads of instructions into the execution units on a single CPU. When successful, gains of up to 30% over a single thread can be achieved. Typically, this works well when the units of work across both threads use different types of

execution units—for example, one performs floating-point operations and the other performs integer operations.

Out-of-order execution enables a compiler to spot that some parts of a linear program sequence do not depend on the results of a previous piece of work, and therefore that both pieces of work could occur in any order or at the same time. As long as sequential results are presented at the right time, the program continues to execute correctly, even though pieces of work are computed out of their programmed order. This enables some instructions to execute when others might be blocked (e.g., waiting for a memory access), allowing greater overall utilization of the available resources.

Finally, and most important for the higher-level programmer, there is the prevalence of multicore architectures. These architectures include multiple CPUs within the same chip, which increases the total capability without running into barriers to making each individual unit faster. This is why it is currently hard to find any machine with fewer than two cores—in this case, the computer has two physical computing units that are connected to each other. While this increases the total number of operations that *can* be done per second, it can make writing code more difficult!

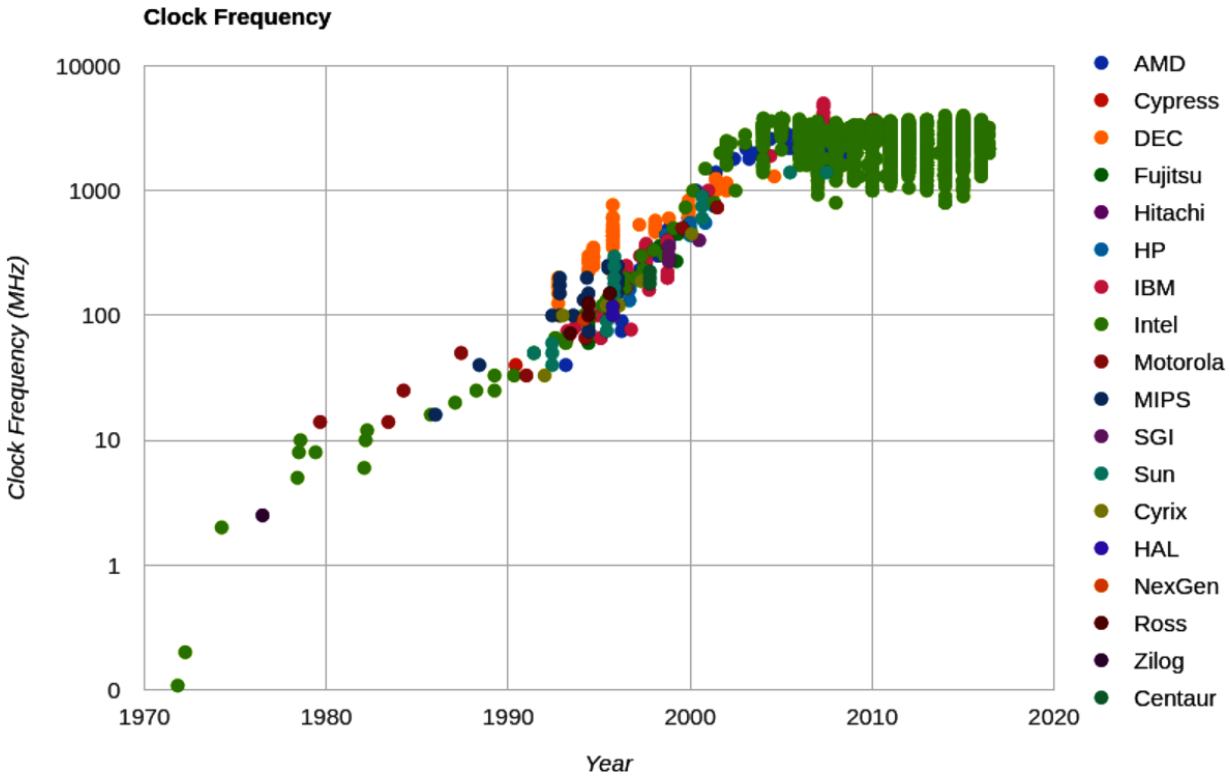


Figure 1-1. Clock speed of CPUs over time (from CPU DB)

Simply adding more cores to a CPU does not always speed up a program's execution time. This is because of something known as *Amdahl's law*. Simply stated, Amdahl's law is this: if a program designed to run on multiple cores has some subroutines that must run on one core, this will be the limitation for the maximum speedup that can be achieved by allocating more cores.

For example, if we had a survey we wanted one hundred people to fill out, and that survey took 1 minute to complete, we could complete this task in 100 minutes if we had one person asking the questions (i.e., this person goes to participant 1, asks the questions, waits for the responses, and then moves to participant 2). This method of having one person asking the

questions and waiting for responses is similar to a serial process. In serial processes, we have operations being satisfied one at a time, each one waiting for the previous operation to complete.

However, we could perform the survey in parallel if we had two people asking the questions, which would let us finish the process in only 50 minutes. This can be done because each individual person asking the questions does not need to know anything about the other person asking questions. As a result, the task can easily be split up without having any dependency between the question askers.

Adding more people asking the questions will give us more speedups, until we have one hundred people asking questions. At this point, the process would take 1 minute and would be limited simply by the time it takes a participant to answer questions. Adding more people asking questions will not result in any further speedups, because these extra people will have no tasks to perform—all the participants are already being asked questions! At this point, the only way to reduce the overall time to run the survey is to reduce the amount of time it takes for an individual survey, the serial portion of the problem, to complete. Similarly, with CPUs, we can add more cores that can perform various chunks of the computation as necessary until we reach a point where the bottleneck is the time it takes for a specific core to finish its task. In other words, the bottleneck in any parallel calculation is always the smaller serial tasks that are being spread out.

However, a major hurdle with utilizing multiple cores in Python is Python's use of a *global interpreter lock* (GIL). The GIL makes sure that a Python process can run only one instruction at a time, regardless of the number of cores it is currently using. This means that even though some Python code has access to multiple cores at a time, only one core is running a Python instruction at any given time. Using the previous example of a survey, this would mean that even if we had 100 question askers, only one person could ask a question and listen to a response at a time. This effectively removes any sort of benefit from having multiple question askers! While this may seem like quite a hurdle, especially if the current trend in computing is to have multiple computing units rather than having faster ones, this problem can be avoided by using other standard library tools, like `multiprocessing` ([Link to Come]), technologies like `numpy` or `numexpr` ([Link to Come]), Cython or Numba ([Link to Come]), or distributed models of computing ([Link to Come]).

NOTE

Python 3.2 also saw a major rewrite of the GIL which made the system much more nimble, alleviating many of the concerns around the system for single-thread performance. Futhermore, there are proposals to make the GIL itself optional (see "Where did the GIL go?"). Although it still locks Python into running only one instruction at a time, the GIL now does better at switching between those instructions and doing so with less overhead.

Memory Units

Memory units in computers are used to store bits. These could be bits representing variables in your program or bits representing the pixels of an image. Thus, the abstraction of a memory unit applies to the registers in your motherboard as well as your RAM and hard drive. The one major difference between all of these types of memory units is the speed at which they can read/write data. To make things more complicated, the read/write speed is heavily dependent on the way that data is being read.

For example, most memory units perform much better when they read one large chunk of data as opposed to many small chunks (this is referred to as *sequential read* versus *random data*). If the data in these memory units is thought of as pages in a large book, this means that most memory units have better read/write speeds when going through the book page by page rather than constantly flipping from one random page to another. While this fact is generally true across all memory units, the amount that this affects each type is drastically different.

In addition to the read/write speeds, memory units also have *latency*, which can be characterized as the time it takes the device to find the data that is being used. For a spinning hard drive, this latency can be high because the disk needs to physically spin up to speed and the read head must move to the right position. On the other hand, for RAM, this latency can be quite small because everything is solid state. Here is a short description of the various memory units that are commonly found inside a standard workstation, in order of read/write speeds:[2]

Spinning hard drive

Long-term storage that persists even when the computer is shut down. Generally has slow read/write speeds because the disk must be physically spun and moved. Degraded performance with random access patterns but very large capacity (20 terabyte range).

Solid-state hard drive

Similar to a spinning hard drive, with faster read/write speeds but smaller capacity (1 terabyte range).

RAM

Used to store application code and data (such as any variables being used). Has fast read/write characteristics and performs well with random access patterns, but is generally limited in capacity (64 gigabyte range).

L1/L2 cache

Extremely fast read/write speeds. Data going to the CPU *must* go through here. Very small capacity (dozens of megabytes range).

Figure 1-2 gives a graphic representation of the differences between these types of memory units by looking at the characteristics of currently available consumer hardware.

A clearly visible trend is that read/write speeds and capacity are inversely proportional—as we try to increase speed, capacity gets reduced. Because of this, many systems implement a tiered approach to memory: data starts in its full state in the hard drive, part of it moves to RAM, and then a much smaller subset moves to the L1/L2 cache. This method of tiering enables programs to keep memory in different places depending on access speed requirements. When trying to optimize the memory patterns of a program, we are simply optimizing which data is placed where, how it is laid out (in order to increase the number of sequential reads), and how many times it is moved among the various locations. In addition, methods such as asynchronous I/O and preemptive caching provide ways to make sure that data is always where it needs to be without having to waste computing time waiting for the I/O to complete —most of these processes can happen independently, while other calculations are being performed! We will discuss these methods in [Link to Come].

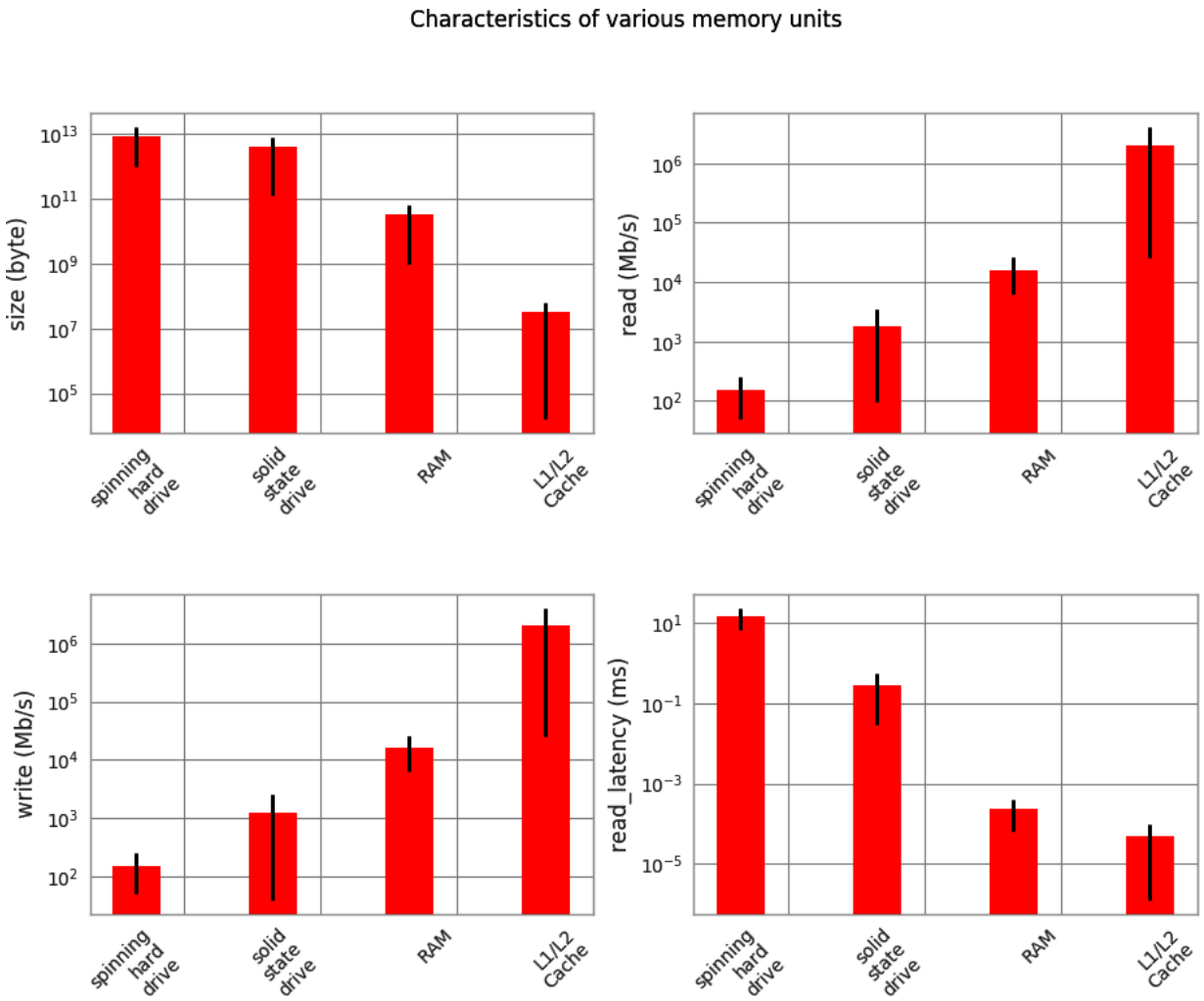


Figure 1-2. Characteristic values for different types of memory units (values from February 2014)

Communications Layers

Finally, let's look at how all of these fundamental blocks communicate with each other. Many modes of communication exist, but all are variants on a thing called a *bus*.

The *frontside bus*, for example, is the connection between the RAM and the L1/L2 cache. It moves data that is ready to be transformed by the processor

into the staging ground to get ready for calculation, and it moves finished calculations out. There are other buses, too, such as the external bus that acts as the main route from hardware devices (such as hard drives and networking cards) to the CPU and system memory. This external bus is generally slower than the frontside bus.

In fact, many of the benefits of the L1/L2 cache are attributable to the faster bus. Being able to queue up data necessary for computation in large chunks on a slow bus (from RAM to cache) and then having it available at very fast speeds from the cache lines (from cache to CPU) enables the CPU to do more calculations without waiting such a long time.

Similarly, many of the drawbacks of using a GPU come from the bus it is connected on: since the GPU is generally a peripheral device, it communicates through the PCI bus, which is much slower than the frontside bus. As a result, getting data into and out of the GPU can be quite a taxing operation. The advent of heterogeneous computing, or computing blocks that have both a CPU and a GPU on the frontside bus, aims at reducing the data transfer cost and making GPU computing more of an available option, even when a lot of data must be transferred.

In addition to the communication blocks within the computer, the network can be thought of as yet another communication block. This block, though, is much more pliable than the ones discussed previously; a network device can be connected to a memory device, such as a network attached storage

(NAS) device or another computing block, as in a computing node in a cluster. However, network communications are generally much slower than the other types of communications mentioned previously. While the frontside bus can transfer dozens of gigabits per second, the network is limited to the order of several dozen megabits.

It is clear, then, that the main property of a bus is its speed: how much data it can move in a given amount of time. This property is given by combining two quantities: how much data can be moved in one transfer (bus width) and how many transfers the bus can do per second (bus frequency). It is important to note that the data moved in one transfer is always sequential: a chunk of data is read off of the memory and moved to a different place. Thus, the speed of a bus is broken into these two quantities because individually they can affect different aspects of computation: a large bus width can help vectorized code (or any code that sequentially reads through memory) by making it possible to move all the relevant data in one transfer, while, on the other hand, having a small bus width but a very high frequency of transfers can help code that must do many reads from random parts of memory. Interestingly, one of the ways that these properties are changed by computer designers is by the physical layout of the motherboard: when chips are placed close to one another, the length of the physical wires joining them is smaller, which can allow for faster transfer speeds. In addition, the number of wires itself dictates the width of the bus (giving real physical meaning to the term!).

Since interfaces can be tuned to give the right performance for a specific application, it is no surprise that there are hundreds of types. Figure 1-3 shows the bitrates for a sampling of common interfaces. Note that this doesn't speak at all about the latency of the connections, which dictates how long it takes for a data request to be responded to (although latency is very computer-dependent, some basic limitations are inherent to the interfaces being used).

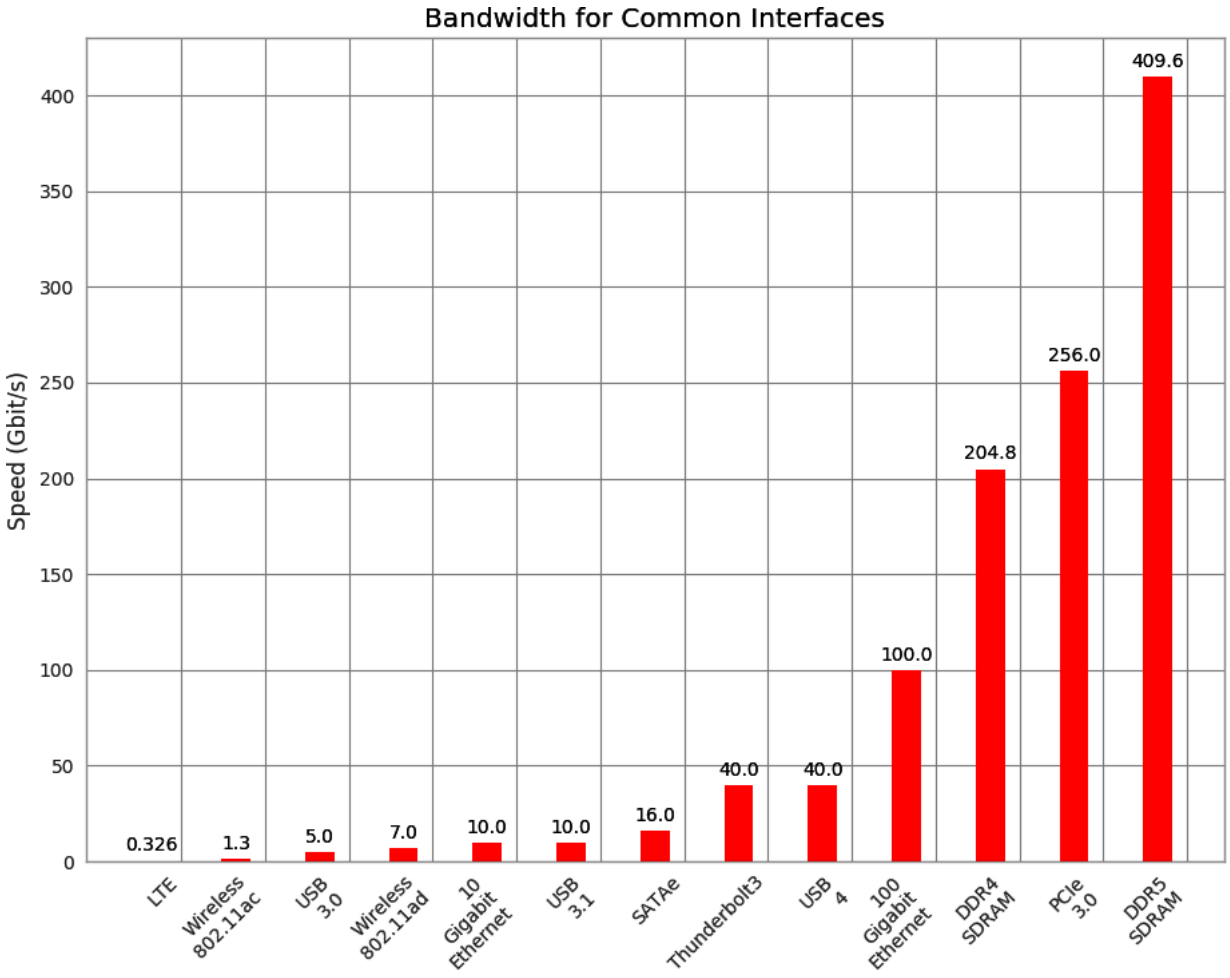


Figure 1-3. Connection speeds of various common interfaces[3]

Putting the Fundamental Elements Together

Understanding the basic components of a computer is not enough to fully understand the problems of high performance programming. The interplay of all of these components and how they work together to solve a problem introduces extra levels of complexity. In this section we will explore some toy problems, illustrating how the ideal solutions would work and how Python approaches them.

A warning: this section may seem bleak—most of the remarks in this section seem to say that Python is natively incapable of dealing with the problems of performance. This is untrue, for two reasons. First, among all of the "components of performant computing," we have neglected one very important component: the developer. What native Python may lack in performance, it gets back right away with speed of development. Furthermore, throughout the book we will introduce modules and philosophies that can help mitigate many of the problems described here with relative ease. With both of these aspects combined, we will keep the fast development mindset of Python while removing many of the performance constraints.

Idealized Computing Versus the Python Virtual Machine

To better understand the components of high performance programming, let's look at a simple code sample that checks whether a number is prime:

```
import math

def check_prime(number):
    sqrt_number = math.sqrt(number)
    for i in range(2, int(sqrt_number) + 1):
        if (number / i).is_integer():
            return False
    return True

print(f"check_prime(10,000,000) = {check_prime(1(
# check_prime(10,000,000) = False
print(f"check_prime(10,000,019) = {check_prime(1(
# check_prime(10,000,019) = True
```

Let's analyze this code using our abstract model of computation and then draw comparisons to what happens when Python runs this code. As with any abstraction, we will neglect many of the subtleties in both the idealized computer and the way that Python runs the code. However, this is generally a good exercise to perform before solving a problem: think about the

general components of the algorithm and what would be the best way for the computing components to come together to find a solution. By understanding this ideal situation and having knowledge of what is actually happening under the hood in Python, we can iteratively bring our Python code closer to the optimal code.

Idealized computing

When the code starts, we have the value of `number` stored in RAM. To calculate `sqrt_number`, we need to send the value of `number` to the CPU. Ideally, we could send the value once; it would get stored inside the CPU's L1/L2 cache, and the CPU would do the calculations and then send the values back to RAM to get stored. This scenario is ideal because we have minimized the number of reads of the value of `number` from RAM, instead opting for reads from the L1/L2 cache, which are much faster. Furthermore, we have minimized the number of data transfers through the frontside bus, by using the L1/L2 cache which is connected directly to the CPU.

TIP

This theme of keeping data where it is needed and moving it as little as possible is very important when it comes to optimization. The concept of "heavy data" refers to the time and effort required to move data around, which is something we would like to avoid.

For the loop in the code, rather than sending one value of `i` at a time to the CPU, we would like to send both `number` and *several* values of `i` to the CPU to check at the same time. This is possible because the CPU vectorizes operations with no additional time cost, meaning it can do multiple independent computations at the same time. So we want to send `number` to the CPU cache, in addition to as many values of `i` as the cache can hold. For each of the `number` / `i` pairs, we will divide them and check if the result is a whole number; then we will send a signal back indicating whether any of the values was indeed an integer. If so, the function ends. If not, we repeat. In this way, we need to communicate back only one result for many values of `i`, rather than depending on the slow bus for every value. This takes advantage of a CPU's ability to *vectorize* a calculation, or run one instruction on multiple data in one clock cycle.

This concept of vectorization is illustrated by the following code:

```
import math

def check_prime(number, V=8):
    sqrt_number = math.sqrt(number)
    numbers = range(2, int(sqrt_number)+1)
    for i in range(0, len(numbers), V):
        # the following line is not valid Python
        result = (number / numbers[i:(i + V)]).is
        if any(result):
```

```
                return False
        return True
```

Here, we set up the processing such that the division and the checking for integers are done on a set of `V` values of `i` at a time. If properly vectorized, the CPU can do this line in one step as opposed to doing a separate calculation for every `i`. Ideally, the `any(result)` operation would also happen in the CPU without having to transfer the results back to RAM. We will talk more about vectorization, how it works, and when it benefits your code in [Link to Come].

Python's virtual machine

The Python interpreter does a lot of work to try to abstract away the underlying computing elements that are being used. At no point does a programmer need to worry about allocating memory for arrays, how to arrange that memory, or in what sequence it is being sent to the CPU. This is a benefit of Python, since it lets you focus on the algorithms that are being implemented. However, it comes at a huge performance cost.

It is important to realize that at its core, Python is indeed running a set of very optimized instructions. The trick, however, is getting Python to perform them in the correct sequence to achieve better performance. For example, it is quite easy to see that, in the following example, `search_fast` will run faster than `search_slow` simply because it

skips the unnecessary computations that result from not terminating the loop early, even though both solutions have runtime `O(n)`. However, things can get complicated when dealing with derived types, special Python methods, or third-party modules. For example, can you immediately tell which function will be faster: `search_unknown1` or `search_unknown2`?

```
def search_fast(haystack, needle):
    for item in haystack:
        if item == needle:
            return True
    return False

def search_slow(haystack, needle):
    return_value = False
    for item in haystack:
        if item == needle:
            return_value = True
    return return_value

def search_unknown1(haystack, needle):
    return any(item == needle for item in haystac

def search_unknown2(haystack, needle):
    return any([item == needle for item in haysta
```

Identifying slow regions of code through profiling and finding more efficient ways of doing the same calculations is similar to finding these useless operations and removing them; the end result is the same, but the number of computations and data transfers is reduced drastically.

```
  The above `search_unknown1` and `search_unknown2
```

One of the impacts of this abstraction layer is that vectorization is not immediately achievable. Our initial prime number routine will run one iteration of the loop per value of `i` instead of combining several iterations. However, looking at the abstracted vectorization example, we see that it is not valid Python code, since we cannot divide a float by a list. External libraries such as `numpy` will help with this situation by adding the ability to do vectorized mathematical operations.

Furthermore, Python's abstraction hurts any optimizations that rely on keeping the L1/L2 cache filled with the relevant data for the next computation. This comes from many factors, the first being that Python objects are not laid out in the most optimal way in memory. This is a consequence of Python being a garbage-collected language—memory is automatically allocated and freed when needed. This creates memory fragmentation that can hurt the transfers to the CPU caches. In addition, at no point is there an opportunity to change the layout of a data structure directly in memory, which means that one transfer on the bus may not

contain all the relevant information for a computation, even though it might have all fit within the bus width.[4]

A second, more fundamental problem comes from Python's dynamic types and the language not being compiled. As many C programmers have learned throughout the years, the compiler is often smarter than you are. When compiling code that is typed and static, the compiler can do many tricks to change the way things are laid out and how the CPU will run certain instructions in order to optimize them. Python, however, is not compiled: to make matters worse, it has dynamic types, which means that inferring any possible opportunities for optimizations algorithmically is drastically harder since code functionality can be changed during runtime. There are many ways to mitigate this problem, foremost being the use of Cython, which allows Python code to be compiled and allows the user to create "hints" to the compiler as to how dynamic the code actually is. Futhermore, Python is on track to having a Just In Time Compiler (JIT) which will allow the code to be compiled and optimized during runtime (more on this in "Does Python have a JIT?").

Finally, the previously mentioned GIL can hurt performance if trying to parallelize this code. For example, let's assume we change the code to use multiple CPU cores such that each core gets a chunk of the numbers from 2 to `sqrtN`. Each core can do its calculation for its chunk of numbers, and then, when the calculations are all done, the cores can compare their calculations. Although we lose the early termination of the loop since each

core doesn't know if a solution has been found, we can reduce the number of checks each core has to do (if we had `M` cores, each core would have to do `sqrtN / M` checks). However, because of the GIL, only one core can be used at a time. This means that we would effectively be running the same code as the unparalleled version, but we no longer have early termination. We can avoid this problem by using multiple processes (with the `multiprocessing` module) instead of multiple threads, or by using Cython or foreign functions.

So Why Use Python?

Python is highly expressive and easy to learn—new programmers quickly discover that they can do quite a lot in a short space of time. Many Python libraries wrap tools written in other languages to make it easy to call other systems; for example, the scikit-learn machine learning system wraps LIBLINEAR and LIBSVM (both of which are written in C), and the `numpy` library includes BLAS and other C and Fortran libraries. As a result, Python code that properly utilizes these modules can indeed be as fast as comparable C code.

Python is described as "batteries included," as many important tools and stable libraries are built in. These include the following:

- `io`

All sorts of IO for bytes, strings and all the encodings you will have to deal with

`array`

Memory-efficient arrays for primitive types

`math`

Basic mathematical operations, including some simple statistics

`sqlite3`

A wrapper around the prevalent SQL file-based storage engine SQLite3

`collections`

A wide variety of objects, including a deque, counter, and dictionary variants

`asyncio`

Concurrent support for I/O-bound tasks using async and await syntax

A huge variety of libraries can be found outside the core language, including these:

`numpy`

A numerical Python library (a bedrock library for anything to do with matrices)

`scipy`

A very large collection of trusted scientific libraries, often wrapping highly respected C and Fortran libraries

`pandas`

A library for data analysis, similar to R's data frames or an Excel spreadsheet, built on `scipy` and `numpy`

`polars`

An alternative to `pandas` with a built-in query planner for faster and parallelized execution of queries

scikit-learn

Rapidly turning into the default machine learning library, built on `scipy`

`tornado`

A library that provides easy bindings for concurrency

PyTorch and TensorFlow

Deep learning frameworks from Facebook and Google with strong Python and GPU support

`NLTK`, `SpaCy`, and `Gensim`

Natural language-processing libraries with deep Python support

Database bindings

For communicating with virtually all databases, including Redis, ElasticSearch, HDF5, and SQL

Web development frameworks

Performant systems for creating websites, such as `aiohttp`, `django`, `pyramid`, `fastapi` or `flask`

`OpenCV`

Bindings for computer vision

API bindings

For easy access to popular web APIs such as Google, Twitter, and LinkedIn

A large selection of managed environments and shells is available to fit various deployment scenarios, including the following:

- The standard distribution, available at *http://python.org*

- `pipenv`, `pyenv`, and `virtualenv` for simple, lightweight, and portable Python environments
- Docker for simple-to-start-and-reproduce environments for development or production
- Anaconda Inc.'s Anaconda, a scientifically focused environment
- IPython, an interactive Python shell heavily used by scientists and developers
- Jupyter Notebook, a browser-based extension to IPython, heavily used for teaching and demonstrations

One of Python's main strengths is that it enables fast prototyping of an idea. Because of the wide variety of supporting libraries, it is easy to test whether an idea is feasible, even if the first implementation might be rather flaky.

If you want to make your mathematical routines faster, look to `numpy`. If you want to experiment with machine learning, try scikit-learn. If you are cleaning and manipulating data, then `pandas` is a good choice.

In general, it is sensible to raise the question, "If our system runs faster, will we as a team run slower in the long run?" It is always possible to squeeze more performance out of a system if enough work-hours are invested, but this might lead to brittle and poorly understood optimizations that ultimately trip up the team.

One example might be the introduction of Cython (see [Link to Come]), a compiler-based approach to annotating Python code with C-like types so the transformed code can be compiled using a C compiler. While the speed gains can be impressive (often achieving C-like speeds with relatively little effort), the cost of supporting this code will increase. In particular, it might be harder to support this new module, as team members will need a certain maturity in their programming ability to understand some of the trade-offs that have occurred when leaving the Python virtual machine that introduced the performance increase.

How to Be a Highly Performant Programmer

Writing high performance code is only one part of being highly performant with successful projects over the longer term. Overall team velocity is far more important than speedups and complicated solutions. Several factors are key to this—good structure, documentation, debuggability, and shared standards.

Let's say you create a prototype. You didn't test it thoroughly, and it didn't get reviewed by your team. It does seem to be "good enough," and it gets pushed to production. Since it was never written in a structured way, it lacks tests and is undocumented. All of a sudden there's an inertia-causing piece

of code for someone else to support, and often management can't quantify the cost to the team.

As this solution is hard to maintain, it tends to stay unloved—it never gets restructured, it doesn't get the tests that'd help the team refactor it, and nobody else likes to touch it, so it falls to one developer to keep it running. This can cause an awful bottleneck at times of stress and raises a significant risk: what would happen if that developer left the project?

Typically, this development style occurs when the management team doesn't understand the ongoing inertia that's caused by hard-to-maintain code. Demonstrating that in the longer-term tests and documentation can help a team stay highly productive and can help convince managers to allocate time to "cleaning up" this prototype code.

In a research environment, it is common to create many Jupyter Notebooks using poor coding practices while iterating through ideas and different datasets. The intention is always to "write it up properly" at a later stage, but that later stage never occurs. In the end, a working result is obtained, but the infrastructure to reproduce it, test it, and trust the result is missing. Once again the risk factors are high, and the trust in the result will be low.

There's a general approach that will serve you well:

Make it work

First you build a good-enough solution. It is very sensible to “build one to throw away” that acts as a prototype solution, enabling a better structure to be used for the second version. It is always sensible to do some up-front planning before coding; otherwise, you’ll come to reflect that “We saved an hour’s thinking by coding all afternoon.” In some fields this is better known as “Measure twice, cut once.”

Make it right

Next, you add a strong test suite backed by documentation and clear reproducibility instructions so that another team member can take it on. This is also a good place to talk about the intention of the code, the challenges that were faced while coming up with the solution, and any notes about the process of building the working version. This will help any future team members when this code needs to be refactored, fixed or rebuilt.

Make it fast

Finally, we can focus on profiling and compiling or parallelization and using the existing test suite to confirm that the new, faster solution still works as expected.

Good Working Practices

There are a few “must haves”—documentation, good structure, and testing are key.

Some project-level documentation will help you stick to a clean structure. It’ll also help you and your colleagues in the future. Nobody will thank you (yourself included) if you skip this part. Writing this up in a *README* file at the top level is a sensible starting point; it can always be expanded into a *docs/* folder later if required.

Explain the purpose of the project, what’s in the folders, where the data comes from, which files are critical, and how to run it all, including how to run the tests.

A *NOTES* file is also a good solution for temporarily storing useful commands, function defaults or other wisdom, tips or tricks for using the code. While this should ideally be put in the documentation, having a scratchpad to keep this information in before it (hopefully) gets into the documentation can be invaluable in not forgetting the important little bits. [5]

Micha recommends also using Docker. A top-level Dockerfile will explain to your future-self exactly which libraries you need from the operating system to make this project run successfully. It also removes the difficulty of running this code on other machines or deploying it to a cloud environment. Often when inheriting new code, simply getting it up and running to play with can be a major hurdle. A Dockerfile removes this

hurdle and lets other developers start interacting with your code immediately.

Add a *tests/* folder and add some unit tests. We prefer `pytest` as a modern test runner, as it builds on Python's built-in `unittest` module. Start with just a couple of tests and then build them up. Progress to using the `coverage` tool, which will report how many lines of your code are actually covered by the tests—it'll help avoid nasty surprises.

If you're inheriting legacy code and it lacks tests, a high-value activity is to add some tests up front. Some "integration tests" that check the overall flow of the project and confirm that with certain input data you get specific output results will help your sanity as you subsequently make modifications.

Every time something in the code bites you, add a test. There's no value to being bitten twice by the same problem.

Docstrings in your code for each function, class, and module will always help you. Aim to provide a useful description of what's *achieved* by the function, and where possible include a short example to demonstrate the expected output. Look at the docstrings inside `numpy` and scikit-learn if you'd like inspiration.

Whenever your code becomes too long—such as functions longer than one screen—be comfortable with refactoring the code to make it shorter. Shorter

code is easier to test and easier to support.

TIP

When you're developing your tests, think about following a test-driven development methodology. When you know exactly what you need to develop and you have testable examples at hand—this method becomes very efficient.

You write your tests, run them, watch them fail, and *then* add the functions and the necessary minimum logic to support the tests that you've written. When your tests all work, you're done. By figuring out the expected input and output of a function ahead of time, you'll find implementing the logic of the function relatively straightforward.

If you can't define your tests ahead of time, it naturally raises the question, do you really understand what your function needs to do? If not, can you write it correctly in an efficient manner? This method doesn't work so well if you're in a creative process and researching data that you don't yet understand well.

Always use source control—you'll only thank yourself when you overwrite something critical at an inconvenient moment. Get used to committing frequently (daily, or even every 10 minutes) and pushing to your repository every day.

Keep to the standard `PEP8` coding standard. Even better, adopt `black` (the opinionated code formatter) on a pre-commit source control hook so it just rewrites your code to the standard for you. Use `flake8` to lint your code to avoid other mistakes.

Creating environments that are isolated from the operating system will make your life easier. Ian prefers Anaconda, while Micha prefers `pyenv` coupled with `virtualenv` or just using Docker. Both are sensible solutions and are significantly better than using the operating system's global Python environment!

Remember that automation is your friend. Doing less manual work means there's less chance of errors creeping in. Automated build systems, continuous integration with automated test suite runners, and automated deployment systems turn tedious and error-prone tasks into standard processes that anyone can run and support. It is never a waste of time to build out your continuous integration toolkit (like running tests automatically when code is checked into your code repository) as it will speed up and streamline future development.

Building libraries is a great way to save on copy-and-paste solutions between early stage projects. It is tempting to copy-and-paste snippets of code because it is quick, but over time you'll have a set of slightly-different but basically the same solutions, each with few or no tests so allowing more bugs and edge cases to impact your work. Sometimes stepping back and identifying opportunities to write a first library can be yield a significant win for a team.

Finally, remember that readability is far more important than being clever. Short snippets of complex and hard-to-read code will be hard for you and

your colleagues to maintain, so people will be scared of touching this code. Instead, write a longer, easier-to-read function and back it with useful documentation showing what it'll return, and complement this with tests to confirm that it *does* work as you expect.

Optimizing for the Team Rather than the Code Block

There are many ways to lose time when building a solution. At worst maybe you're working on the *wrong problem* or with the *wrong approach*, maybe you're on the right track but there are taxes in your development process that slow you down, maybe you haven't estimated the true costs and uncertainties that might get in your way. Or maybe you misunderstand the needs of the stakeholders and spending time building a feature or solving a problem that doesn't actually exist.[6]

Making sure you're solving *a useful problem* is critical. Finding a cool project with cutting edge technology and lots of neat acronyms can be wonderfully fun - but it is unlikely to deliver the value that other project members will appreciate. If you're in an organisation that is trying to cause a positive change, you have to focus on problems that block and can solve that positive change.

Having found potentially-useful problems to solve it is worth reflecting - can we make a *meaningful* change? Just fixing "the tech" behind a problem

won't change the real world. The solution needs to be deployed and maintained and needs to be adopted by human users. If there's resistance or blockage to the technical solution then your work will go nowhere.

Having decided that those blockers aren't a worry - have you estimated the potential impact you can *realistically* have? If you find a part of your problem space where you can have a 100x impact - great! Does that part of the problem represent a meaningful chunk of work for the day to day of your organisation? If you make a 100x impact on a problem that's seen just a few hours a year then the work is (probably) without use. If you can make a 1% improvement on something that hurts the team *every single day* then you'll be a hero.

One way to estimate the value you provide is to think about the cost of the current-state and the potential gain of the future-state (when you've written your solution). How do you quantify the cost and improvement? Tieing estimates down to money (as "time is money" and all of us people burn time) is a great way to figure out what kind of impact you'll have and to be able to communicate it to colleagues. This is also a great way of prioritising potential project options.

When you've found useful and valuable problems to solve next you need to make sure you're solving them in sensible ways. Taking a hard problem and deciding immediately to use a hard solution *might* be sensible, but starting with a simple solution and learning why it does and doesn't work can

quickly yield valuable insights that inform subsequent iterations of your solution. What's the quickest and simplest way you can learn something useful?

Ian has worked with clients with near-release complex NLP pipelines but low confidence that they actually work. After a review it was revealed that a team had built a complex system, but missed the upstream poor-data-annotation problem that was confounding the NLP ML process. By switching to a far simpler solution (without deep neural networks, using old fashion NLP tooling) the issues were identified, the data consistently relabeled, and only then could we build up towards more sophisticated solutions now that up-stream issues had sensibly been removed.

Is your team communicating its results clearly to stakeholders? Are you communicating clearly within your team? A lack of communication is an easy way to add an frustrating cost to your team's progress.

Review your collaborative practices to check that processes such as frequent code reviews are in place. It is so easy to "save some time" by ignoring a code review and forgetting that you're letting colleagues (and yourself) get away with unreviewed code that might be solving the wrong problem or may contain errors that a fresh set of eyes could see before they have a worse and later impact.

The Remote Performant Programmer

Since the COVID-19 Pandemic we've witnessed a switch to fully-remote and hybrid practices. Whilst some organisations have tried to bring teams back on-site, most have adopted hybrid or fully remote practices now that best practices are reasonably well understood.

Remote practices mean we can live anywhere and the hiring and collaborator pool can be far wider - either limited by similar time zones or not limited at all. Some organisations have noticed that open source projects such as Python, Pandas, `scikit-learn` and plenty more are working wonderfully successfully with a globally distributed team who rarely ever meet in person.

Increased communication is critical and often a "documentation first" culture has to be developed. Some teams go as far to say that "if it isn't document on our chat tool (like Slack) then it never happened" - this means that every decision ends up being written down so it is communicated and can be searched for.

It is also easy to feel isolated when working fully remotely for a long time. Having regular checkins with team members, even if you are not working on the same project, and unstructured time where you can talk at a higher level (or just about life!) is important in feeling connected and part of a team.

Some Thoughts on Good Notebook Practice

If you're using Jupyter Notebooks, they're great for visual communication, but they facilitate laziness. If you find yourself leaving long functions inside your Notebooks, be comfortable extracting them out to a Python module and then adding tests.

Consider prototyping your code in IPython or the QTConsole; turn lines of code into functions in a Notebook and then promote them out of the Notebook and into a module complemented by tests. Finally, consider wrapping the code in a class if encapsulation and data hiding are useful.

Liberally spread `assert` statements throughout a Notebook to check that your functions are behaving as expected. You can't easily test code inside a Notebook, and until you've refactored your functions into separate modules, `assert` checks are a simple way to add some level of validation. You shouldn't trust this code until you've extracted it to a module and written sensible unit tests.

Using `assert` statements to check data in your code should be frowned upon. It is an easy way to assert that certain conditions are being met, but it isn't idiomatic Python. To make your code easier to read by other developers, check your expected data state and then raise an appropriate exception if the check fails. A common exception would be `ValueError` if a function encounters an unexpected value. The Pandera

library is an example of a testing framework focused on Pandas and Polars to check that your data meets the specified constraints.

You may also want to add some sanity checks at the end of your Notebook—a mixture of logic checks and `raise` and `print` statements that demonstrate that you've just generated exactly what you needed. When you return to this code in six months, you'll thank yourself for making it easy to see that it worked correctly all the way through!

One difficulty with Notebooks is sharing code with source control systems. nbdime is one of a growing set of new tools that let you diff your Notebooks. It is a lifesaver and enables collaboration with colleagues.

Getting the Joy Back into Your Work

Life can be complicated. In the ten years since your authors wrote the first edition of this book, we've jointly experienced through friends and family a number of life situations, including new children, depression, cancer, home relocations, successful business exits and failures, and career direction shifts. Inevitably, these external events will have an impact on anyone's work and outlook on life.

Remember to keep looking for the joy in new activities. There are always interesting details or requirements once you start poking around. You might ask, "why did they make that decision?" and "how would I do it

differently?" and all of a sudden you're ready to start a conversation about how things might be changed or improved.

Keep a log of things that are worth celebrating. It is so easy to forget about accomplishments and to get caught up in the day-to-day. People get burned out because they're always running to keep up, and they forget how much progress they've made.

We suggest that you build a list of items worth celebrating and note how you celebrate them. Ian keeps such a list—he's happily surprised when he goes to update the list and sees just how many cool things have happened (and might otherwise have been forgotten!) in the last year. These shouldn't just be work milestones; include hobbies and sports, and celebrate the milestones you've achieved. Micha makes sure to prioritize her personal life and spend days away from the computer to work on nontechnical projects or to prioritise rest, relaxation and slowness. It is critical to keep developing your skill set, but it is not necessary to burn out!

Programming, particularly when performance focused, thrives on a sense of curiosity and a willingness to always delve deeper into the technical details. Unfortunately, this curiosity is the first thing to go when you burn out; so take your time and make sure you enjoy the journey, and keep the joy and the curiosity.

The future of Python

Where did the GIL go?

As discussed "Memory Units" the *Global Interpreter Lock (GIL)* is the standard memory locking mechanism that can unfortunately make multi-threaded code run - at worst - at single-thread speeds. The GIL's job is to make sure that only one thread can modify a Python object at a time, so if multiple threads in one program try to modify the same object, they effectively each get to make their modifications one-at-a-time.

This massively simplified the early design of Python but as the processor count has increased, it has added a growing tax to writing multi-core code. The GIL is a core part of Python's reference counting garbage collection machinery.

In 2023 a decision was made to investigate building a GIL-free version of Python which would still support threads in addition to the long-standing GIL build. Since third party libraries (e.g. NumPy, Pandas, `scikit-learn`) have compiled C code which *relies* upon the current GIL implementation, some code gymnastics will be required for external libraries to support both builds of Python and to move to a GIL-less build in the longer term. Nobody wants a repeat of the 10 year Python 2 to Python 3 transition again!

Python Enhancement Proposal `PEP-703` [7] describes the proposal with a focus on scientific and AI applications. The main issue in this domain is

that with CPU-intensive code and 10-100 threads the overhead of the GIL can significantly reduce the parallelization opportunity. By switching to the standard solutions (e.g. `multiprocessing`) described in this book, a significant developer overhead and communications overhead can be introduced. None of these options enable the best use of the machine's resources without significant effort.

This PEP notes the issues with non-atomic object modifications which need to be controlled for along with a new small-object memory allocator that is thread-safe.

We might expect a GIL-less version of Python to be generally available from 2028 - if no significant blockers are discovered during this journey.

Does Python have a JIT?

Starting with Python 3.13 we expect that a just-in-time compiler (JIT) will be built into the main CPython that almost everyone uses.

This JIT follows a 2021 design called "copy and patch" which was first used in the Lua language. As a contrast in technologies such as PyPy and Numba an analyser discovers slow code sections (AKA hot-spots), then compiles a machine-code version that matches this code block with whatever specialisations are available to the CPU on that machine. You get

really fast code, but the compilation process can be expensive on the early passes.

The “copy and patch” process is a little different to the contrasting approach. When the `python` executable is built (normally by the Python Software Foundation) the LLVM compiler toolchain is used to build a set of pre-defined “stencils”. These stencils are semi-compiled versions of critical op-codes from the Python virtual machine. They’re called “stencils” because they have “holes” which are filled in later.

At run time when a hot-spot is identified - typically a loop where the datatypes don’t change - you can take a matching set of stencils that match the op-codes, fill in the “holes” by pasting in the memory addresses of the relevant variables, then the op-codes no longer need to be interpreted as the machine code equivalent is available. This promises to be much faster than compiling each hot spot that’s identified, it may not be as optimal but is hoped to provide significant gains without a slow analysis and compilation pass.

Getting to the point where a JIT is possible has taken a couple of evolutionary stages in major Python releases:

- 3.11 introduced an adaptive type specializing interpreter which provided 10-25% speed-ups

- 3.12 introduced internal clean-ups and a domain specific language for the creation of the interpreter enabling modification at build-time
- 3.13 introduced a hot-spot detector to build on the specialized types with the copy-and-patch JIT

It is worth noting that whilst the introduction of a JIT in Python 3.13 is a great step, it is unlikely to impact any of our Pandas, NumPy and SciPy code as internally these libraries often use C and Cython to pre-compile faster solutions. The JIT will have an impact on anyone writing native Python, particularly numeric Python.

Not to be confused with interprocess communication, which shares the same acronym—we'll look at that topic in [Link to Come].

Speeds in this section are from *https://oreil.ly/pToi7*.

Data is from *https://oreil.ly/7SC8d*.

In [Link to Come], we'll see how we can regain this control and tune our code all the way down to the memory utilization patterns.

Micha generally keeps a notes files open while developing a solution and once things are working, she spends time clearing out the notes file into proper documentation and auxiliary tests and benchmarks.

Micha has, in several occasions, shadowed stakeholders throughout their day to better understand how they work, how they approach problems and what their day to day was like. This "take a developer to work day" approach helped her better adapt her technical solutions to their needs.

- *https://peps.python.org/pep-0703/*

Chapter 2. Profiling to Find Bottlenecks

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

QUESTIONS YOU'LL BE ABLE TO ANSWER AFTER THIS CHAPTER

- How can I identify speed and RAM bottlenecks in my code?
- How do I profile CPU and memory usage?
- What depth of profiling should I use?
- How can I profile a long-running application?
- What's happening under the hood with CPython?
- How do I keep my code correct while tuning performance?

Profiling lets us find bottlenecks so we can do the least amount of work to get the biggest practical performance gain. While we'd like to get huge gains in speed and reductions in resource usage with little work, practically you'll aim for your code to run "fast enough" and "lean enough" to fit your needs. Profiling will let you make the most pragmatic decisions for the least overall effort.

Any measurable resource can be profiled (not just the CPU!). In this chapter we look at both CPU time and memory usage. You could apply similar techniques to measure network bandwidth and disk I/O too.

If a program is running too slowly or using too much RAM, you'll want to fix whichever parts of your code are responsible. You could, of course, skip profiling and fix what you *believe* might be the problem—but be wary, as you'll often end up "fixing" the wrong thing. Rather than using your intuition, it is far more sensible to first profile, having defined a hypothesis, before making changes to the structure of your code.

Sometimes it's good to be lazy. By profiling first, you can quickly identify the bottlenecks that need to be solved, and then you can solve just enough of these to achieve the performance you need. If you avoid profiling and jump to optimization, you'll quite likely do more work in the long run. Always be driven by the results of profiling.

Profiling Efficiently

The first aim of profiling is to test a representative system to identify what's slow (or using too much RAM, or causing too much disk I/O or network I/O). Profiling typically adds an overhead (10× to 100× slowdowns can be typical), and you still want your code to be used in as similar to a real-world situation as possible. Extract a test case and isolate the piece of the system that you need to test. Preferably, it'll have been written to be in its own set of modules already.

The basic techniques that are introduced first in this chapter include the `%timeit` magic in IPython, `time.time()`, and a timing decorator. You can use these techniques to understand the behavior of statements and functions.

Then we will cover `cProfile` ("Using the cProfile Module"), showing you how to use this built-in tool to understand which functions in your code take the longest to run. This will give you a high-level view of the problem so you can direct your attention to the critical functions.

Next, we'll look at `line_profiler` ("Using line_profiler for Line-by-Line Measurements"), which will profile your chosen functions on a line-by-line basis. The result will include a count of the number of times each line is called and the percentage of time spent on each line. This is exactly the information you need to understand what's running slowly and why.

Armed with the results of `line_profiler`, you'll have the information you need to move on to using a compiler ([Link to Come]).

In [Link to Come], you'll learn how to use `perf stat` to understand the number of instructions that are ultimately executed on a CPU and how efficiently the CPU's caches are utilized. This allows for advanced-level tuning of matrix operations. You should take a look at [Link to Come] when you're done with this chapter.

After `line_profiler`, if you're working with long-running systems, then you'll be interested in `py-spy` to peek into already-running Python processes.

To help you understand why your RAM usage is high, we'll show you `memory_profiler` ("Using memory_profiler to Diagnose Memory Usage"). It is particularly useful for tracking RAM usage over time on a labeled chart, so you can explain to colleagues why certain functions use more RAM than expected.

If you'd like to combine CPU and RAM profiling you'll want to read about Scalene ("Combining CPU and Memory Profiling with Scalene"), this combines the jobs of `line_profiler` and `memory_profiler` with a novel low-impact memory allocator and also contains experimental GPU profiling support.

VizTracer ("VizTracer for an interactive time-based call stack") will let you see a time-based view on your code's execution, it presents a call stack down the page with time running from left-to-right. You can click into the call stack and even annotate custom messages and behaviour.

WARNING

Whatever approach you take to profiling your code, you must remember to have adequate unit test coverage in your code. Unit tests help you to avoid silly mistakes and to keep your results reproducible. Avoid them at your peril.

Always profile your code before compiling or rewriting your algorithms. You need evidence to determine the most efficient ways to make your code run faster.

Next, we'll give you an introduction to the Python bytecode inside CPython ("Using the dis Module to Examine CPython Bytecode"), so you can understand what's happening "under the hood." In particular, having an understanding of how Python's stack-based virtual machine operates will help you understand why certain coding styles run more slowly than others. Specialist ("Digging into bytecode specialisation with Specialist") will then helps us see which parts of the bytecode can be identified for performance improvements from Python 3.11 and above.

Before the end of the chapter, we'll review how to integrate unit tests while profiling ("Unit Testing During Optimization to Maintain Correctness") to

preserve the correctness of your code while you make it run more efficiently.

We'll finish with a discussion of profiling strategies ("Strategies to Profile Your Code Successfully") so you can reliably profile your code and gather the correct data to test your hypotheses. Here you'll learn how dynamic CPU frequency scaling and features like Turbo Boost can skew your profiling results, and you'll learn how they can be disabled.

To walk through all of these steps, we need an easy-to-analyze function. The next section introduces the Julia set. It is a CPU-bound function that's a little hungry for RAM; it also exhibits nonlinear behavior (so we can't easily predict the outcomes), which means we need to profile it at runtime rather than analyzing it offline.

Introducing the Julia Set

The Julia set is an interesting CPU-bound problem for us to begin with. It is a fractal sequence that generates a complex output image, named after Gaston Julia.

The code that follows is a little longer than a version you might write yourself. It has a CPU-bound component and a very explicit set of inputs. This configuration allows us to profile both the CPU usage and the RAM usage so we can understand which parts of our code are consuming two of

our scarce computing resources. This implementation is *deliberately* suboptimal, so we can identify memory-consuming operations and slow statements. Later in this chapter we'll fix a slow logic statement and a memory-consuming statement, and in [Link to Come] we'll significantly speed up the overall execution time of this function.

We will analyze a block of code that produces both a false grayscale plot (Figure 2-1) and a pure grayscale variant of the Julia set (Figure 2-3), at the complex point `c=-0.62772-0.42193j`. A Julia set is produced by calculating each pixel in isolation; this is an "embarrassingly parallel problem," as no data is shared between points.

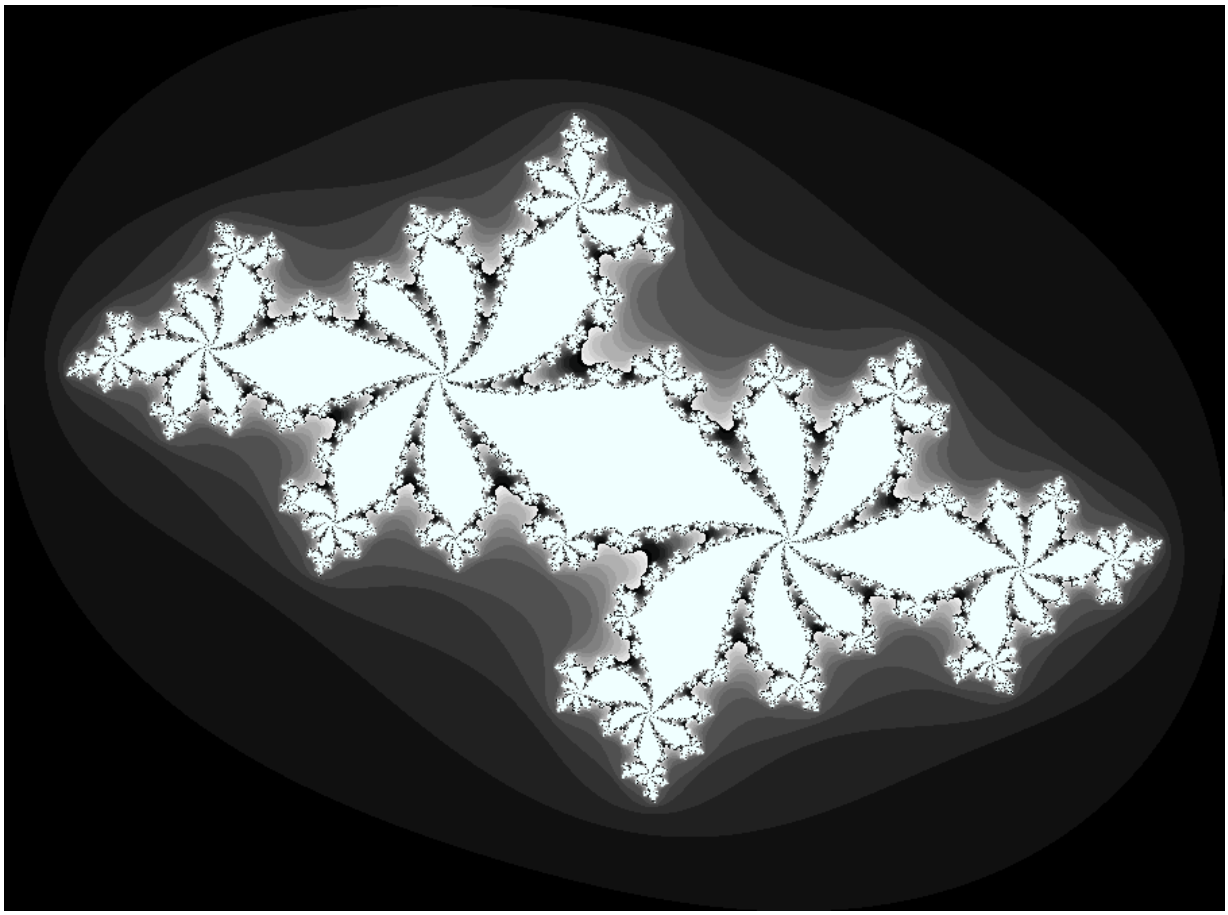


Figure 2-1. Julia set plot with a false gray scale to highlight detail

If we chose a different c, we'd get a different image. The location we have chosen has regions that are quick to calculate and others that are slow to calculate; this is useful for our analysis.

The problem is interesting because we calculate each pixel by applying a loop that could be applied an indeterminate number of times. On each iteration we test to see if this coordinate's value escapes toward infinity, or if it seems to be held by an attractor. Coordinates that cause few iterations are colored darkly in Figure 2-1, and those that cause a high number of

iterations are colored white. White regions are more complex to calculate and so take longer to generate.

We define a set of z coordinates that we'll test. The function that we calculate squares the complex number `z` and adds `c`:

$$f(z) = z^2 + c$$

We iterate on this function while testing to see if the escape condition holds using `abs`. If the escape function is `False`, we break out of the loop and record the number of iterations we performed at this coordinate. If the escape function is never `False`, we stop after `maxiter` iterations. We will later turn this `z`'s result into a colored pixel representing this complex location.

In pseudocode, it might look like this:

```
for z in coordinates:
    for iteration in range(maxiter):  # limited i
        if abs(z) < 2.0:  # has the escape condit
            z = z*z + c
        else:
            break
    # store the iteration count for each z and di
```

To explain this function, let's try two coordinates.

We'll use the coordinate that we draw in the top-left corner of the plot at `-1.8-1.8j`. We must test `abs(z) < 2` before we can try the update rule:

```
z = -1.8-1.8j
print(abs(z))
```

```
2.54558441227
```

We can see that for the top-left coordinate, the `abs(z)` test will be `False` on the zeroth iteration as `2.54 >= 2.0`, so we do not perform the update rule. The `output` value for this coordinate is `0`.

Now let's jump to the center of the plot at `z = 0 + 0j` and try a few iterations:

```
c = -0.62772-0.42193j
z = 0+0j
for n in range(9):
    z = z*z + c
    print(f"{n}: z={z: .5f}, abs(z)={abs(z):0.3f}
```

```
0: z=-0.62772-0.42193j, abs(z)=0.756, c=-0.62772-
1: z=-0.41171+0.10778j, abs(z)=0.426, c=-0.62772-
2: z=-0.46983-0.51068j, abs(z)=0.694, c=-0.62772-
```

```
  3: z=-0.66777+0.05793j, abs(z)=0.670, c=-0.62772-
  4: z=-0.18516-0.49930j, abs(z)=0.533, c=-0.62772-
  5: z=-0.84274-0.23703j, abs(z)=0.875, c=-0.62772-
  6: z= 0.02630-0.02242j, abs(z)=0.035, c=-0.62772-
  7: z=-0.62753-0.42311j, abs(z)=0.757, c=-0.62772-
  8: z=-0.41295+0.10910j, abs(z)=0.427, c=-0.62772-
```

We can see that each update to `z` for these first iterations leaves it with a value where `abs(z) < 2` is `True`. For this coordinate we can iterate 300 times, and still the test will be `True`. We cannot tell how many iterations we must perform before the condition becomes `False`, and this may be an infinite sequence. The maximum iteration (`maxiter`) break clause will stop us from iterating potentially forever.

In Figure 2-2, we see the first 50 iterations of the preceding sequence. For `0+0j` (the solid line with circle markers), the sequence appears to repeat every eighth iteration, but each sequence of seven calculations has a minor deviation from the previous sequence—we can't tell if this point will iterate forever within the boundary condition, or for a long time, or maybe for just a few more iterations. The dashed `cutoff` line shows the boundary at `+2`.

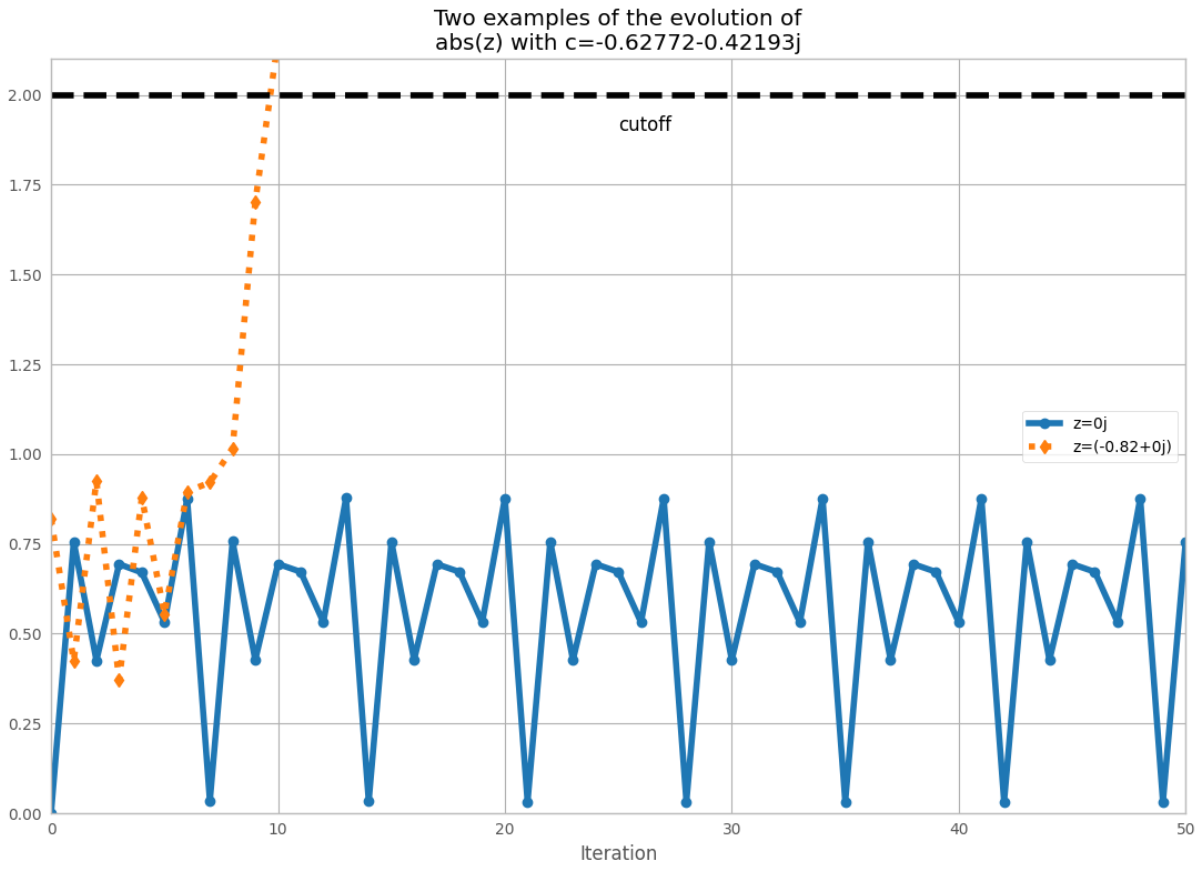


Figure 2-2. Two coordinate examples evolving for the Julia set

For `-0.82+0j` (the dashed line with diamond markers), we can see that after the ninth update, the absolute result has exceeded the `+2` cutoff, so we stop updating this value.

Calculating the Full Julia Set

In this section we break down the code that generates the Julia set. We'll analyze it in various ways throughout this chapter. As shown in Example 2-

1, at the start of our module we import the `time` module for our first profiling approach and define some coordinate constants.

Example 2-1. Defining global constants for the coordinate space

```
"""Julia set generator without optional PIL-based
import time

# area of complex space to investigate
x1, x2, y1, y2 = -1.8, 1.8, -1.8, 1.8
c_real, c_imag = -0.62772, -.42193
```

To generate the plot, we create two lists of input data. The first is `zs` (complex *z* coordinates), and the second is `cs` (a complex initial condition). Neither list varies, and we could optimize `cs` to a single `c` value as a constant. The rationale for building two input lists is so that we have some reasonable-looking data to profile when we profile RAM usage later in this chapter.

To build the `zs` and `cs` lists, we need to know the coordinates for each `z`. In Example 2-2, we build up these coordinates using `xcoord` and `ycoord` and a specified `x_step` and `y_step`. The somewhat verbose nature of this setup is useful when porting the code to other tools (such as `numpy`) and to other Python environments, as it helps to have everything *very* clearly defined for debugging.

Example 2-2. Establishing the coordinate lists as inputs to our calculation function

```
def calc_pure_python(desired_width, max_iteratio
    """Create a list of complex coordinates (zs)
    build Julia set"""
    x_step = (x2 - x1) / desired_width
    y_step = (y1 - y2) / desired_width
    x = []
    y = []
    ycoord = y2
    while ycoord > y1:
        y.append(ycoord)
        ycoord += y_step
    xcoord = x1
    while xcoord < x2:
        x.append(xcoord)
        xcoord += x_step
    # build a list of coordinates and the initial
    # Note that our initial condition is a consta
    # we use it to simulate a real-world scenario
    # function
    zs = []
    cs = []
    for ycoord in y:

        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c real, c imag))
```

```
    print("Length of x:", len(x))
    print("Total elements:", len(zs))
    start_time = time.time()
    output = calculate_z_serial_purepython(max_it
    end_time = time.time()
    secs = end_time - start_time
    print(f"{calculate_z_serial_purepython.__name

    # This sum is expected for a 1000^2 grid with
    # It ensures that our code evolves exactly as
    assert sum(output) == 33219980
```

Having built the `zs` and `cs` lists, we output some information about the size of the lists and calculate the `output` list via `calculate_z_serial_purepython`. Finally, we `sum` the contents of `output` and `assert` that it matches the expected output value. Ian uses it here to confirm that no errors creep into the book.

As the code is deterministic, we can verify that the function works as we expect by summing all the calculated values. This is useful as a sanity check—when we make changes to numerical code, it is *very* sensible to check that we haven't broken the algorithm. Ideally, we would use unit tests and test more than one configuration of the problem.

Next, in Example 2-3, we define the `calculate_z_serial_purepython` function, which expands on the algorithm we discussed earlier. Notably, we also define an `output` list at the start that has the same length as the input `zs` and `cs` lists.

Example 2-3. Our CPU-bound calculation function

```
def calculate_z_serial_purepython(maxiter, zs, cs
    """Calculate output list using Julia update r
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Now we call the calculation routine in Example 2-4. By wrapping it in a `__main__` check, we can safely import the module without starting the calculations for some of the profiling methods. Here, we're not showing the method used to plot the output.

Example 2-4. `__main__` for our code

```
if __name__ == "__main__":
    # Calculate the Julia set using a pure Python
    # reasonable defaults for a laptop
    calc_pure_python(desired_width=1000, max_ite
```

Once we run the code, we see some output about the complexity of the problem:

```
# running the above produces:
Length of x: 1,000
Total elements: 1,000,000
calculate_z_serial_purepython took 5.80 seconds
```

In the false-grayscale plot (Figure 2-1), the high-contrast color changes gave us an idea of where the cost of the function was slow changing or fast changing. Here, in Figure 2-3, we have a linear color map: black is quick to calculate, and white is expensive to calculate.

By showing two representations of the same data, we can see that lots of detail is lost in the linear mapping. Sometimes it can be useful to have various representations in mind when investigating the cost of a function.

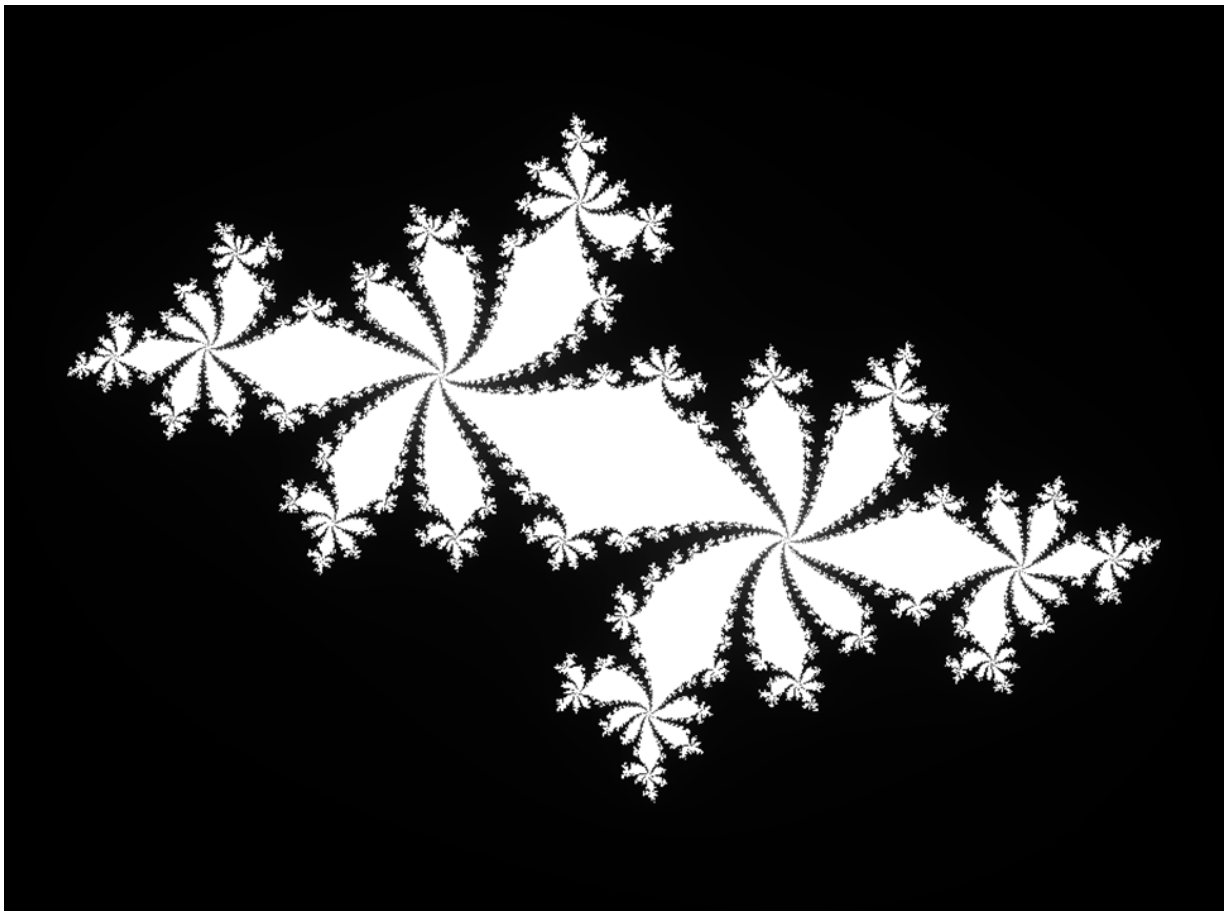


Figure 2-3. Julia plot example using a pure gray scale

Simple Approaches to Timing—print and a Decorator

After Example 2-4, we saw the output generated by several `print` statements in our code. On Ian's laptop, this code takes approximately 5 seconds to run using CPython 3.12. It is useful to note that execution time always varies. You must observe the normal variation when you're timing

your code, or you might incorrectly attribute an improvement in your code to what is simply a random variation in execution time.

Your computer will be performing other tasks while running your code, such as accessing the network, disk, or RAM, and these factors can cause variations in the execution time of your program.

Ian's laptop is a Dell XPS 15 9510 with an Intel Core I7-11800H (2.3 GHz, 24MB Level 3 cache, Eight physical Cores with Hyperthreading) with 64 GB system RAM running Linux Minx 21.2 (based on Ubuntu 22.04).

In `calc_pure_python` (Example 2-2), we can see several `print` statements. This is the simplest way to measure the execution time of a piece of code *inside* a function. It is a basic approach, but despite being quick and dirty, it can be very useful when you're first looking at a piece of code.

Using `print` statements is commonplace when debugging and profiling code. It quickly becomes unmanageable but is useful for short investigations. Try to tidy up the `print` statements when you're done with them, or they will clutter your `stdout`.

A slightly cleaner approach is to use a *decorator*—here, we add one line of code above the function that we care about. Our decorator can be very simple and just replicate the effect of the `print` statements. Later, we can make it more advanced.

In Example 2-5, we define a new function, `timefn`, which takes a function as an argument: the inner function, `measure_time`, takes `*args` (a variable number of positional arguments) and `**kwargs` (a variable number of key/value arguments) and passes them through to `fn` for execution.

Around the execution of `fn`, we capture `time.time()` and then `print` the result along with `fn.__name__`. The overhead of using this decorator is small, but if you're calling `fn` millions of times, the overhead might become noticeable. We use `@wraps(fn)` to expose the function name and docstring to the caller of the decorated function (otherwise, we would see the function name and docstring for the decorator, not the function it decorates).

Example 2-5. Defining a decorator to automate timing measurements

```
from functools import wraps

def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.time()
        result = fn(*args, **kwargs)
        t2 = time.time()
        print(f"@timefn: {fn.__name__} took {(t2
        return result
    return measure time
```

```
@timefn
def calculate_z_serial_purepython(maxiter, zs, cs
    ...
```

When we run this version (we keep the `print` statements from before), we can see that the execution time in the decorated version is ever-so-slightly quicker than the call from `calc_pure_python`. This is due to the overhead of calling a function (the difference is very tiny):

```
Length of x: 1,000
Total elements: 1,000,000
@timefn: calculate_z_serial_purepython took 5.78
calculate_z_serial_purepython took 5.78 seconds
```

NOTE

The addition of profiling information will inevitably slow down your code—some profiling options are very informative and induce a heavy speed penalty. The trade-off between profiling detail and speed will be something you have to consider.

We can use the `timeit` module as another way to get a coarse measurement of the execution speed of our CPU-bound function. More

typically, you would use this when timing different types of simple expressions as you experiment with ways to solve a problem.

WARNING

The `timeit` module temporarily disables the garbage collector. This might impact the speed you'll see with real-world operations if the garbage collector would normally be invoked by your operations. See the Python documentation for help on this.

From the command line, you can run `timeit` as follows:

```
python -m timeit -n 5 -r 1 -s "import julia1_nop
 "julia1_nopil.calc_pure_python(desired_width=10
```

Note that you have to import the module as a setup step using `-s`, as `calc_pure_python` is inside that module. `timeit` has some sensible defaults for short sections of code, but for longer-running functions it can be sensible to specify the number of loops (`-n 5`) and the number of repetitions (`-r 5`) to repeat the experiments. The best result of all the repetitions is given as the answer. Adding the verbose flag (`-v`) shows the cumulative time of all the loops by each repetition, which can help your variability in the results.

By default, if we run `timeit` on this function without specifying `-n` and `-r`, it runs 10 loops with 5 repetitions, and this takes six minutes to

complete. Overriding the defaults can make sense if you want to get your results a little faster.

We're interested only in the best-case results, as other results will probably have been impacted by other processes:

```
Length of x: 1,000
Total elements: 1,000,000
calculate_z_serial_purepython took 5.78 seconds


...
5 loops, best of 1: 6.1 sec per loop
```

Try running the benchmark several times to check if you get varying results—you may need more repetitions to settle on a stable fastest-result time. There is no "correct" configuration, so if you see a wide variation in your timing results, do more repetitions until your final result is stable.

Our results show that the overall cost of calling `calc_pure_python` is 6.1 seconds (as the best case), while single calls to `calc_pure_python` take approximately 5.8 seconds as measured by the `@timefn` decorator. The difference is mainly the time taken to create the `zs` and `cs` lists before `start_time` is recorded.

Inside IPython, we can use the magic `%timeit` in the same way. If you are developing your code interactively in IPython or in a Jupyter Notebook, you can use this:

```
In [1]: import julia1_nopil
In [2]: %timeit julia1_nopil.calc_pure_python(des
                                                max
```

WARNING

Be aware that "best" is calculated differently by the `timeit.py` approach and the `%timeit` approach in Jupyter and IPython. `timeit.py` uses the minimum value seen. IPython in 2016 switched to using the mean and standard deviation. Both methods have their flaws, but generally they're both "reasonably good"; you can't compare between them, though. Use one method or the other; don't mix them.

It is worth considering the variation in load that you get on a normal computer. Many background tasks are running (e.g., Dropbox, backups) that could impact the CPU and disk resources at random. Scripts in web pages can also cause unpredictable resource usage. Figure 2-4 shows the single CPU being used at 100% for some of the timing steps we just performed; the other cores on this machine are each lightly working on other tasks.

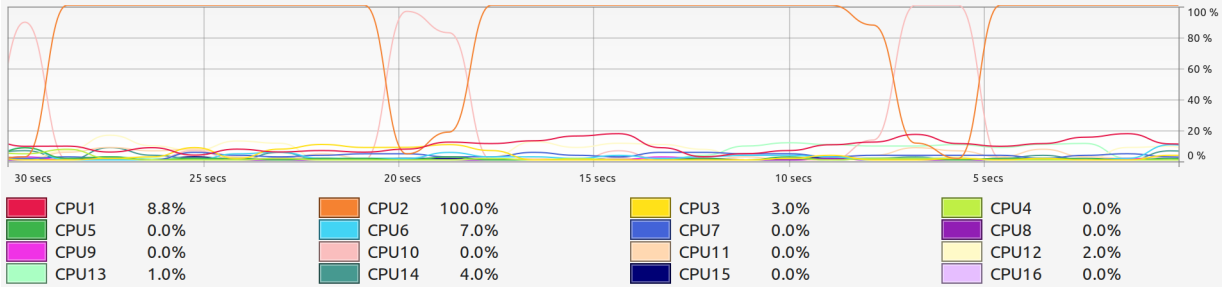


Figure 2-4. System Monitor on Ubuntu showing variation in background CPU usage while we time our function

Occasionally, the System Monitor shows spikes of activity on this machine. It is sensible to watch your System Monitor to check that nothing else is interfering with your critical resources (CPU, disk, network).

Simple Timing Using the Unix time Command

We can step outside of Python for a moment to use a standard system utility on Unix-like systems. The following will record various views on the execution time of your program, and it won't care about the internal structure of your code:

```
$ /usr/bin/time -p python julia1_nopil.py
Length of x: 1,000
Total elements: 1,000,000
calculate_z_serial_purepython took 5.71 seconds
real 6.02
user 5.96
```

```
  sys 0.05
```

Note that we specifically use `/usr/bin/time` rather than `time` so we get the system's `time` and not the simpler (and less useful) version built into our shell. If you try `time --verbose` and you get an error, you're probably looking at the shell's built-in `time` command and not the system command.

Using the `-p` portability flag, we get three results:

- `real` records the wall clock or elapsed time.
- `user` records the amount of time the CPU spent on your task outside of kernel functions.
- `sys` records the time spent in kernel-level functions.

By adding `user` and `sys`, you get a sense of how much time was spent in the CPU. The difference between this and `real` might tell you about the amount of time spent waiting for I/O; it might also suggest that your system is busy running other tasks that are distorting your measurements.

`time` is useful because it isn't specific to Python. It includes the time taken to start the `python` executable, which might be significant if you start lots of fresh processes (rather than having a long-running single process). If you often have short-running scripts where the startup time is a

significant part of the overall runtime, then `time` can be a more useful measure.

We can add the `--verbose` flag to get even more output:

```
Length of x: 1,000
Total elements: 1,000,000
calculate_z_serial_purepython took 5.76 seconds
        Command being timed: "python julia1_nopil
        User time (seconds): 6.01
        System time (seconds): 0.05
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:s
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 9843
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 2
        Voluntary context switches: 1
        Involuntary context switches: 37
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
```

```
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

One useful indicator is `Maximum resident set size`, this indicates the maximum amount of RAM used during execution - if it nears the phsical RAM you have available, you'll be close to either running out of RAM or using disk-swap which is very slow. This execution cost 98 MB at its worst.

Another useful indicator here is `Major (requiring I/O) page faults`, this indicates whether the operating system is having to load pages of data from the disk because the data no longer resides in RAM. This will cause a speed penalty, here it doesn't as it records 0 page faults.

In our example, the code and data requirements are small, so no page faults occur. If you have a memory-bound process, or several programs that use variable and large amounts of RAM, you might find that this gives you a clue as to which program is being slowed down by disk accesses at the operating system level because parts of it have been swapped out of RAM to disk.

Using the cProfile Module

`cProfile` is a built-in profiling tool in the standard library. It hooks into the virtual machine in CPython to measure the time taken to run every function that it sees. This introduces a greater overhead, but you get correspondingly more information. Sometimes the additional information can lead to surprising insights into your code.

`cProfile` is one of two profilers in the standard library, alongside `profile`. `profile` is the original and slower pure Python profiler; `cProfile` has the same interface as `profile` and is written in `C` for a lower overhead. If you're curious about the history of these libraries, see Armin Rigo's 2005 request to include `cProfile` in the standard library.

A good practice when profiling is to generate a *hypothesis* about the speed of parts of your code before you profile it. Ian likes to print out the code snippet in question and annotate it. Forming a hypothesis ahead of time means you can measure how wrong you are (and you will be!) and improve your intuition about certain coding styles.

WARNING

You should never avoid profiling in favor of a gut instinct (we warn you—you *will* get it wrong!). It is definitely worth forming a hypothesis ahead of profiling to help you learn to spot possible slow choices in your code, and you should always back up your choices with evidence.

Always be driven by results that you have measured, and always start with some quick-and-dirty profiling to make sure you're addressing the right

area. There's nothing more humbling than cleverly optimizing a section of code only to realize (hours or days later) that you missed the slowest part of the process and haven't really addressed the underlying problem at all.

Let's hypothesize that `calculate_z_serial_purepython` is the slowest part of the code. In that function, we do a lot of dereferencing and make many calls to basic arithmetic operators and the `abs` function. These will probably show up as consumers of CPU resources.

Here, we'll use the `cProfile` module to run a variant of the code. The output is spartan but helps us figure out where to analyze further.

The `-s cumulative` flag tells `cProfile` to sort by cumulative time spent inside each function; this gives us a view into the slowest parts of a section of code. The `cProfile` output is written to screen directly after our usual `print` results:

```
$ python -m cProfile -s cumulative julia1_nopil.p
Length of x: 1,000
Total elements: 1,000,000
calculate_z_serial_purepython took 13.15 seconds
         36221995 function calls in 14.301 second

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall fil
        1    0.000    0.000   14.301   14.301 {bu
```

```
         1    0.035    0.035   14.301   14.301 jul
         1    0.803    0.803   14.267   14.267 jul
                                                (ca
         1    8.420    8.420   13.150   13.150 jul
                                                (ca
  34219980    4.730    0.000    4.730    0.000 {bu
   2002000    0.306    0.000    0.306    0.000 {me
                                                 ob

         1    0.007    0.007    0.007    0.007 {bu
         3    0.000    0.000    0.000    0.000 {bu
         1    0.000    0.000    0.000    0.000 {me
                                                 '_
         2    0.000    0.000    0.000    0.000 {bu
         4    0.000    0.000    0.000    0.000 {bu
```

Sorting by cumulative time gives us an idea about where the majority of execution time is spent. This result shows us that 36,221,995 function calls occurred in just over 13 seconds (this time includes the overhead of using `cProfile`). Previously, our code took around 5 seconds to execute—we've just added a 8-second penalty by measuring how long each function takes to execute.

We can see that the entry point to the code `julia1_nopil.py` on line 1 takes a total of 14 seconds. This is just the `__main__` call to

`calc_pure_python`. `ncalls` is 1, indicating that this line is executed only once.

Inside `calc_pure_python`, the call to `calculate_z_serial_purepython` consumes 13 seconds. Both functions are called only once. We can derive that approximately 1 second is spent on lines of code inside `calc_pure_python`, separate to calling the CPU-intensive `calculate_z_serial_purepython` function. However, we can't derive *which* lines take the time inside the function using `cProfile`.

Inside `calculate_z_serial_purepython`, the time spent on lines of code (without calling other functions) is 8 seconds. This function makes 34,219,980 calls to `abs`, which take a total of 4 seconds, along with other calls that do not cost much time.

What about the `{abs}` call? This line is measuring the individual calls to the `abs` function inside `calculate_z_serial_purepython`. While the per-call cost is negligible (it is recorded as 0.000 seconds), the total time for 34,219,980 calls is 4 seconds. We couldn't predict in advance exactly how many calls would be made to `abs`, as the Julia function has unpredictable dynamics (that's why it is so interesting to look at).

At best we could have said that it will be called a minimum of 1 million times, as we're calculating `1000*1000` pixels. At most it will be called

300 million times, as we calculate 1,000,000 pixels with a maximum of 300 iterations. So 34 million calls is roughly 10% of the worst case.

If we look at the original grayscale image (Figure 2-3) and, in our mind's eye, squash the white parts together and into a corner, we can estimate that the expensive white region accounts for roughly 10% of the rest of the image.

The next line in the profiled output, `{method 'append' of 'list' objects}`, details the creation of 2,002,000 list items.

TIP

Why 2,002,000 items? Before you read on, think about how many list items are being constructed.

This creation of 2,002,000 items is occurring in `calc_pure_python` during the setup phase.

The `zs` and `cs` lists will be `1000*1000` items each (generating 1,000,000 * 2 calls), and these are built from a list of 1,000 *x* and 1,000 *y* coordinates. In total, this is 2,002,000 calls to append.

It is important to note that this `cProfile` output is not ordered by parent functions; it is summarizing the expense of all functions in the executed block of code. Figuring out what is happening on a line-by-line basis is very

hard with `cProfile`, as we get profile information only for the function calls themselves, not for each line within the functions.

Inside `calculate_z_serial_purepython`, we can account for `{abs}`, and in total this function costs approximately 4.7 seconds. We know that `calculate_z_serial_purepython` costs 13.1 seconds in total.

The final line of the profiling output refers to `lsprof`; this is the original name of the tool that evolved into `cProfile` and can be ignored.

To get more control over the results of `cProfile`, we can write a statistics file and then analyze it in Python:

```
$ python -m cProfile -o profile.stats julia1_nop
```

We can load this into Python as follows, and it will give us the same cumulative time report as before:

```
In [1]: import pstats
In [2]: p = pstats.Stats("profile.stats")
In [3]: p.sort_stats("cumulative")
Out[3]: <pstats.Stats at 0x7fe8747e8470>

In [4]: p.print_stats()
Thu Feb 22 20:38:55 2024    profile.stats
```

```
         36221995 function calls in 14.398 second

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall fil
        1    0.000    0.000   14.398   14.398 {bu
        1    0.036    0.036   14.398   14.398 jul
        1    0.799    0.799   14.363   14.363 jul
                                               (ca
        1    8.453    8.453   13.252   13.252 jul
                                               (ca
 34219980    4.799    0.000    4.799    0.000 {bu
  2002000    0.304    0.000    0.304    0.000 {me
                                                ob
        1    0.008    0.008    0.008    0.008 {bu
        3    0.000    0.000    0.000    0.000 {bu
        1    0.000    0.000    0.000    0.000 {me
                                                '_
        2    0.000    0.000    0.000    0.000 {bu
        4    0.000    0.000    0.000    0.000 {bu
```

To trace which functions we're profiling, we can print the caller information. In the following two listings we can see that `calculate_z_serial_purepython` is the most expensive function, and it is called from one place. If it were called from many places,

these listings might help us narrow down the locations of the most expensive parents:

```
In [5]: p.print_callers()
   Ordered by: cumulative time

Function                                      was called b
                                                   ncalls
{built-in method builtins.exec}               <-
julia1_nopil.py:1(<module>)                   <-       1
                                               {built-in m
julia1_nopil.py:23(calc_pure_python)          <-       1
                                               :1(<module>
julia1_nopil.py:9(...)                        <-       1
                                               :23(calc_pu
{built-in method builtins.abs}                <- 34219980
                                               :9(calculat
{method 'append' of 'list' objects}           <- 2002000
                                               :23(calc_pu
{built-in method builtins.sum}                <-       1
                                               :23(calc_pu
{built-in method builtins.print}              <-       3
                                               :23(calc_pu
{built-in method time.time}                   <-       2
                                               :23(calc_pu
{built-in method builtins.len}                <-       2
                                               :9(calculat
```

We can flip this around the other way to show which functions call other functions:

```
In [6]: p.print_callees()
   Ordered by: cumulative time

Function
                                                  ncalls
{built-in method builtins.exec}                ->      1
                                                julia1_nopi
julia1_nopil.py:1(<module>)                    ->      1
                                                julia1_nopi
julia1_nopil.py:23(calc_pure_python)           ->      1
                                                julia1_nopi
                                                        2
                                                {built-in m
                                                        3
                                                {built-in m
                                                        1
                                                {built-in m
                                                        2

                                                {built-in m
                                                   2002000
                                                {method 'ap
julia1_nopil.py:9(...)                         -> 34219980
                                                {built-in m
                                                        2
```

```
                                                {built-in r
  {built-in method builtins.abs}        ->
  {method 'append' of 'list' objects}   ->
  {built-in method builtins.sum}        ->
  {built-in method builtins.print}      ->
  {built-in method time.time}           ->
  {built-in method builtins.len}        ->
```

`cProfile` is rather verbose, and you need a side screen to see it without lots of word wrapping. Since it is built in, though, it is a convenient tool for quickly identifying bottlenecks. Tools like `line_profiler` and `memory_profiler`, which we discuss later in this chapter, will then help you to drill down to the specific lines that you should pay attention to.

Visualizing cProfile Output with SnakeViz

`snakeviz` is a visualizer that draws the output of `cProfile` as a diagram in which larger boxes are areas of code that take longer to run. It replaces the older `runsnake` tool.

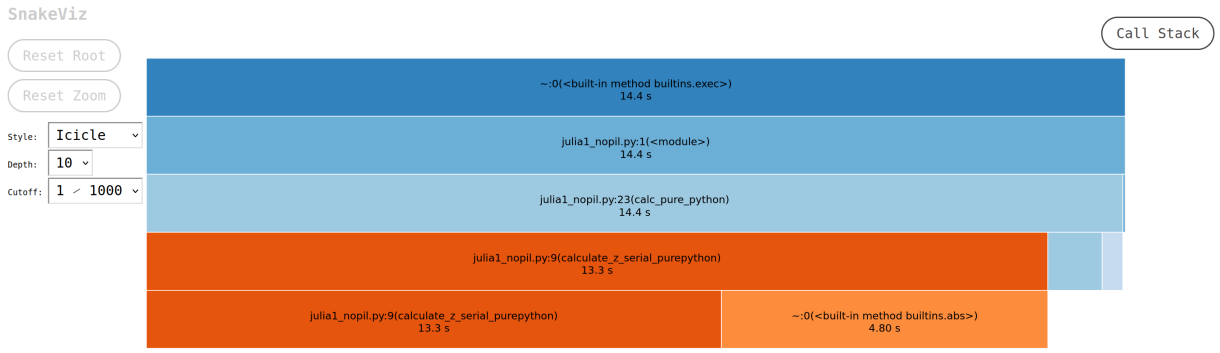
Use `snakeviz` to get a high-level understanding of a `cProfile` statistics file, particularly if you're investigating a new project for which you have little intuition. The diagram will help you visualize the CPU-

usage behavior of the system, and it may highlight areas that you hadn't expected to be expensive.

To install SnakeViz, use `pip install snakeviz`.

In Figure 2-5 we have the visual output of the *profile.stats* file we've just generated. The entry point for the program is shown at the top of the diagram. Each layer down is a function called from the function above.

The width of the diagram represents the entire time taken by the program's execution. The fourth layer shows that the majority of the time is spent in `calculate_z_serial_purepython`. The fifth layer breaks this down some more—the unannotated block to the right occupying approximately 33% of that layer represents the time spent in the `abs` function. Seeing these larger blocks quickly brings home how the time is spent inside your program.



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	8.453	8.453	13.25	13.25	julia1_nopil.py:9(calculate_z_serial_purepython)
1	0.7986	0.7986	14.36	14.36	julia1_nopil.py:23(calc_pure_python)
1	0.03564	0.03564	14.4	14.4	julia1_nopil.py:1(<module>)
1	0.007733	0.007733	0.007733	0.007733	~:0(<built-in method builtins.sum>)
1	5.096e-05	5.096e-05	5.096e-05	5.096e-05	~:0(<method 'disable' of '_lsprof.Profiler' objects>)
1	8.719e-06	8.719e-06	14.4	14.4	~:0(<built-in method builtins.exec>)
2	3.593e-06	1.797e-06	3.593e-06	1.797e-06	~:0(<built-in method time.time>)
3	7.513e-05	2.504e-05	7.513e-05	2.504e-05	~:0(<built-in method builtins.print>)
4	2.693e-06	6.733e-07	2.693e-06	6.733e-07	~:0(<built-in method builtins.len>)
2002000	0.3042	1.519e-07	0.3042	1.519e-07	~:0(<method 'append' of 'list' objects>)
34219980	4.799	1.402e-07	4.799	1.402e-07	~:0(<built-in method builtins.abs>)

Figure 2-5. `snakeviz` visualizing profile.stats

The next section down shows a table that is a pretty-printed version of the statistics we've just been looking at, which you can sort by `cumtime` (cumulative time), `percall` (cost per call), or `ncalls` (number of calls altogether), among other categories. Starting with `cumtime` will tell you which functions cost the most overall. They're a pretty good place to start your investigations.

If you're comfortable looking at tables, the console output for `cProfile` may be adequate for you. To communicate to others, we strongly suggest you use diagrams—such as this output from `snakeviz`—to help others quickly understand the point you're making.

Using line_profiler for Line-by-Line Measurements

In Ian's opinion, Robert Kern's `line_profiler` is the strongest tool for identifying the cause of CPU-bound problems in Python code. It works by profiling individual functions on a line-by-line basis, so you should start with `cProfile` and use the high-level view to guide which functions to profile with `line_profiler`.

It is worthwhile printing and annotating versions of the output from this tool as you modify your code, so you have a record of changes (successful or not) that you can quickly refer to. Don't rely on your memory when you're working on line-by-line changes.

To install `line_profiler`, issue the command `pip install line_profiler`.

A decorator (`@profile`) is used to mark the chosen function. The `kernprof` script is used to execute your code, and the CPU time and other statistics for each line of the chosen function are recorded.

The arguments are `-l` for line-by-line (rather than function-level) profiling and `-v` for verbose output. Without `-v`, you receive an *.lprof* output that

you can later analyze with the `line_profiler` module. In Example 2-6, we'll do a full run on our CPU-bound function.

Example 2-6. Running `kernprof` with line-by-line output on a decorated function to record the CPU cost of each line's execution

```
$ kernprof -l -v julia1_lineprofiler.py
...
Wrote profile results to julia1_lineprofiler.py.l
Timer unit: 1e-06 s

Total time: 31.0996 s
File: julia1_lineprofiler.py
Function: calculate_z_serial_purepython at line 1

Line #      Hits Per Hit % Time  Line Contents
==============================================
    16                                @profile
    17                                def calculate_z_

    18                                    """Calculate
                                          Julia upd
    19         1  3720.4    0.0      output = [0]
    20   1000001     0.3    0.8      for i in ran
    21   1000000     0.2    0.6          n = 0
    22   1000000     0.2    0.7          z = zs[i
    23   1000000     0.2    0.7          c = cs[i
    24  34219980     0.4   44.7          while ab
```

```
    25  33219980     0.3   29.2              z =
    26  33219980     0.2   22.5              n +=
    27   1000000     0.2    0.7         output[:
    28         1     3.6    0.0      return outpu
```

Introducing `kernprof.py` adds a substantial amount to the runtime. In this example, `calculate_z_serial_purepython` takes 31 seconds; this is up from 5 seconds using simple `print` statements and 13 seconds using `cProfile`. The gain is that we get a line-by-line breakdown of where the time is spent inside the function.

The `% Time` column is the most helpful—we can see that 4% of the time is spent on the `while` testing. We don't know whether the first statement (`abs(z) < 2`) is more expensive than the second (`n < maxiter`), though. Inside the loop, we see that the update to `z` is also fairly expensive. Even `n += 1` is expensive! Python's dynamic lookup machinery is at work for every loop, even though we're using the same types for each variable in each loop—this is where compiling and type specialization ([Link to Come]) give us a massive win. The creation of the `output` list and the updates on line 19 are relatively cheap compared to the cost of the `while` loop.

If you haven't thought about the complexity of Python's dynamic machinery before, do think about what happens in that `n += 1` operation. Python has to check that the `n` object has an `__add__` function (and if it

didn't, it'd walk up any inherited classes to see if they provided this functionality), and then the other object (`1` in this case) is passed in so that the `__add__` function can decide how to handle the operation. Remember that the second argument might be a `float` or other object that may or may not be compatible. This all happens dynamically.

The obvious way to further analyze the `while` statement is to break it up. While there has been some discussion in the Python community around the idea of rewriting the *.pyc* files with more detailed information for multipart, single-line statements, we are unaware of any production tools that offer a more fine-grained analysis than `line_profiler`.

In Example 2-7, we break the `while` logic into several statements. This additional complexity will increase the runtime of the function, as we have more lines of code to execute, but it *might* also help us understand the costs incurred in this part of the code.

TIP

Before you look at the code, do you think we'll learn about the costs of the fundamental operations this way? Might other factors complicate the analysis?

Example 2-7. Breaking the compound `while` statement into individual statements to record the cost of each part of the original statement

```
$ kernprof -l -v julia1_lineprofiler2.py
...
Wrote profile results to julia1_lineprofiler2.py
Timer unit: 1e-06 s

Total time: 63.3558 s
File: julia1_lineprofiler2.py
Function: calculate_z_serial_purepython at line 1

Line #      Hits Per Hit % Time  Line Contents
==============================================
    15                                @profile
    16                                def calculate_z_

    17                                    """Calculate
                                             Julia upd
    18         1  3862.9    0.0      output = [0]
    19   1000001     0.3    0.5      for i in ran
    20   1000000     0.3    0.4          n = 0
    21   1000000     0.3    0.5          z = zs[i
    22   1000000     0.3    0.4          c = cs[i
    23  34219980     0.2   12.3          while Tr
    24  34219980     0.4   21.4              not_
    25  34219980     0.3   14.8              iter
    26  34219980     0.3   18.6              if n
    27  33219980     0.3   15.7
    28  33219980     0.3   14.6
    29                                        else
```

```
    30   1000000      0.2    0.4
    31   1000000      0.3    0.4          output[:
    32         1      3.0    0.0      return outpu
```

This version takes 63 seconds to execute, while the previous version took 31 seconds. Other factors *did* complicate the analysis. In this case, having extra statements that have to be executed 34,219,980 times each slows down the code. If we hadn't used `kernprof.py` to investigate the line-by-line effect of this change, we might have drawn other conclusions about the reason for the slowdown, as we'd have lacked the necessary evidence.

At this point it makes sense to step back to the earlier `timeit` technique to test the cost of individual expressions:

```
Python 3.12.0 | packaged by Anaconda, Inc. | (ma
Type 'copyright', 'credits' or 'license' for more
IPython 8.21.0 -- An enhanced Interactive Python
In [1]: z = 0+0j
In [2]: %timeit abs(z) < 2
66.8 ns ± 2.07 ns per loop (mean ± std. dev. of
In [3]: n = 1
In [4]: maxiter = 300
In [5]: %timeit n < maxiter
20.7 ns ± 0.0611 ns per loop (mean ± std. dev. o
```

From this simple analysis, it looks as though the logic test on `n` is more than three times faster than the call to `abs`. Since the order of evaluation for Python statements is both left to right and opportunistic, it makes sense to put the cheapest test on the left side of the equation. On 1 in every 301 tests for each coordinate, the `n < maxiter` test will be `False`, so Python wouldn't need to evaluate the other side of the `and` operator.

We never know whether `abs(z) < 2` will be `False` until we evaluate it, and our earlier observations for this region of the complex plane suggest it is `True` around 10% of the time for all 300 iterations. If we wanted to have a strong understanding of the time complexity of this part of the code, it would make sense to continue the numerical analysis. In this situation, however, we want an easy check to see if we can get a quick win.

We can form a new hypothesis stating, "By swapping the order of the operators in the `while` statement, we will achieve a reliable speedup." We *can* test this hypothesis using `kernprof`, but the additional overheads of profiling this way might add too much noise. Instead, we can use an earlier version of the code, running a test comparing `while abs(z) < 2 and n < maxiter:` against `while n < maxiter and abs(z) < 2:`, which we see in Example 2-8.

Running the two variants *outside* of `line_profiler` means they run at similar speeds, on the same problem size that we've been using (with `x==1000`).

The overheads of `line_profiler` also confuse the result, and the results on line 23 for both versions are similar and the timing for this final result is a little slower at 33s. We might reject the hypothesis that in Python 3.12 changing the order of the logic results in a consistent speedup—there's no clear evidence for this.

However if the problem is made harder by setting `x==5000`, the version with the swapped `if` statement is consistently slightly faster. Ian measures 143s for the original order and 142s for the swapped order of operations.

Using a more suitable approach to solve this problem (e.g., swapping to using Cython or PyPy, as described in [Link to Come]) would yield greater gains.

We can be confident in our result because of the following:

- We stated a hypothesis that was easy to test.
- We changed our code so that only the hypothesis would be tested (never test two things at once!).
- We gathered enough evidence to support our conclusion.

For completeness, we can run a final `kernprof` on the two main functions including our optimization to confirm that we have a full picture of the overall complexity of our code.

Example 2-8. Swapping the order of the compound `while` statement makes the function fractionally faster

```
$ kernprof -l -v julia1_lineprofiler3.py
...
Wrote profile results to julia1_lineprofiler3.py
Timer unit: 1e-06 s

Total time: 33.9135 s
File: julia1_lineprofiler3.py
Function: calculate_z_serial_purepython at line 1

Line #      Hits Per Hit % Time  Line Contents
==============================================
    15                                @profile
    16                                def calculate_z_

    17                                    """Calculate
                                          update ru
    18         1  4015.8    0.0      output = [0]
    19   1000001     0.3    1.0      for i in ran
    20   1000000     0.2    0.7          n = 0
    21   1000000     0.3    0.8          z = zs[i
    22   1000000     0.2    0.7          c = cs[i
    23  34219980     0.4   44.1          while n
    24  33219980     0.3   29.0              z =
    25  33219980     0.2   22.9              n +=
```

```
     26   1000000      0.3    0.8           output[:
     27         1      2.1    0.0       return outpu
```

As expected, we can see from the output in Example 2-9 that `calculate_z_serial_purepython` takes most (98%) of the time of its parent function. The list-creation steps are minor in comparison.

Example 2-9. Testing the line-by-line costs of the setup routine

```
Total time: 64.0333 s
File: julia1_lineprofiler3.py
Function: calc_pure_python at line 30

Line #      Hits Per Hit % Time  Line Contents
==============================================
    30                           @profile
    31                           def calc_pure_py



...
    52      1001      0.3    0.0    for ycoord in
    53   1001000      0.3    0.5        for xcoord
    54   1000000      0.4    0.6            zs.app
    55   1000000      0.4    0.6            cs.app
    56
    57         1     56.2    0.0    print(f"Length
    58         1      7.4    0.0    print(f"Total
```

```
     59          1       5.1      0.0    start_time = t
     60          1    6e+07     98.3    output = calcu
                                                max_i
     61          1       6.0      0.0    end_time = tim
     62          1       1.3      0.0    secs = end_tim
     63          1      48.6      0.0    print(f"{calcu
                                                took {
     64
     65          1    7129.7      0.0    assert sum(out
                                         is expected fo
```

`line_profiler` gives us a great insight into the cost of lines inside loops and expensive functions; even though profiling adds a speed penalty, it is a great boon to scientific developers. Remember to use representative data to make sure you're focusing on the lines of code that'll give you the biggest win.

Using memory_profiler to Diagnose Memory Usage

Just as Robert Kern's `line_profiler` package measures CPU usage, the `memory_profiler` module by Fabian Pedregosa and Philippe Gervais measures memory usage on a line-by-line basis. Understanding the

memory usage characteristics of your code allows you to ask yourself two questions:

- Could we use *less* RAM by rewriting this function to work more efficiently?
- Could we use *more* RAM and save CPU cycles by caching?

`memory_profiler` operates in a very similar way to `line_profiler` but runs far more slowly. If you install the `psutil` package (optional but recommended), `memory_profiler` will run faster. Memory profiling may easily make your code run 10 to 100 times slower. In practice, you will probably use `memory_profiler` occasionally and `line_profiler` (for CPU profiling) more frequently.

Install `memory_profiler` with the command `pip install memory_profiler` (and optionally with `pip install psutil`).

As mentioned, the implementation of `memory_profiler` is not as performant as the implementation of `line_profiler`. It may therefore make sense to run your tests on a smaller problem that completes in a useful amount of time. Overnight runs might be sensible for validation, but you need quick and reasonable iterations to diagnose problems and hypothesize solutions. The code in Example 2-10 uses the full 1,000 × 1,000 grid, and the statistics took about two hours to collect on Ian's laptop.

NOTE

The requirement to modify the source code is a minor annoyance. As with `line_profiler`, a decorator (`@profile`) is used to mark the chosen function. This will break your unit tests unless you make a dummy decorator—see "No-op @profile Decorator".

When dealing with memory allocation, you must be aware that the situation is not as clear-cut as it is with CPU usage. Generally, it is more efficient to overallocate memory in a process that can be used at leisure, as memory allocation operations are relatively expensive. Furthermore, garbage collection is not instantaneous, so objects may be unavailable but still in the garbage collection pool for some time.

The outcome of this is that it is hard to really understand what is happening with memory usage and release inside a Python program, as a line of code may not allocate a deterministic amount of memory *as observed from outside the process*. Observing the gross trend over a set of lines is likely to lead to better insight than would be gained by observing the behavior of just one line.

Let's take a look at the output from `memory_profiler` in Example 2-10. Inside `calculate_z_serial_purepython` on line 18, we see that the allocation of 1,000,000 items causes approximately 7 MB of RAM to be added to this process.[1] This does not mean that the `output` list is definitely 7 MB in size, just that the process grew by approximately 7 MB during the internal allocation of the list.

In the parent function on line 52, we see that the allocation of the `zs` and `cs` lists changes the `Mem usage` column from 47 MB to 123 MB (a change of +76 MB). Again, it is worth noting that this is not necessarily the true size of the arrays, just the size that the process grew by after these lists had been created.

At the time of writing, the `memory_usage` module exhibits a bug—the `Increment` column does not always match the change in the `Mem usage` column. During the first edition of this book, these columns were correctly tracked; you might want to check the status of this bug on GitHub. We recommend you use the `Mem usage` column, as this correctly tracks the change in process size per line of code.

Example 2-10. `memory_profiler`'s result on both of our main functions, showing an unexpected memory use in `calculate_z_serial_purepython`

```
$ python -m memory_profiler julia1_memoryprofile
...

Line #    Mem usage    Increment   Line Contents
================================================
    15  123.086 MiB  123.086 MiB  @profile
    16                            def calculate_

    17                                """Calculat
    18  130.711 MiB    7.625 MiB      output = [(
```

```
    19  133.461 MiB    0.000 MiB      for i in ra
    20  133.461 MiB    0.000 MiB          n = 0
    21  133.461 MiB    0.000 MiB          z = zs
    22  133.461 MiB    0.000 MiB          c = cs
    23  133.461 MiB    2.750 MiB          while r
    24  133.461 MiB    0.000 MiB              z =
    25  133.461 MiB    0.000 MiB              n +
    26  133.461 MiB    0.000 MiB          output
    27  133.461 MiB    0.000 MiB      return outp
...

Line #    Mem usage    Increment   Line Contents
================================================
    30   47.137 MiB   47.137 MiB  @profile
    31                             def calc_pure_p


    32                                 """Create a
    33   47.137 MiB    0.000 MiB      x_step = (x
    34   47.137 MiB    0.000 MiB      y_step = (y
    35   47.137 MiB    0.000 MiB      x = []
    36   47.137 MiB    0.000 MiB      y = []
    37   47.137 MiB    0.000 MiB      ycoord = y2
    38   47.137 MiB    0.000 MiB      while ycoor

    39   47.137 MiB    0.000 MiB          y.apper
    40   47.137 MiB    0.000 MiB          ycoord
    41   47.137 MiB    0.000 MiB      xcoord = x1
    42   47.137 MiB    0.000 MiB      while xcoor
```

```
    43   47.137 MiB    0.000 MiB           x.apper
    44   47.137 MiB    0.000 MiB           xcoord
    50   47.137 MiB    0.000 MiB       zs = []
    51   47.137 MiB    0.000 MiB       cs = []
    52  123.086 MiB   -1.055 MiB       for ycoord
    53  123.086 MiB -893.922 MiB           for xc
    54  123.086 MiB -900.219 MiB               zs
    55  123.086 MiB -853.844 MiB               cs
    56
    57  123.086 MiB    0.000 MiB       print("Leng
    58  123.086 MiB    0.000 MiB       print("Tota
    59  123.086 MiB    0.000 MiB       start_time
    60  133.461 MiB  133.461 MiB       output = ca
    61  133.461 MiB    0.000 MiB       end_time =
    62  133.461 MiB    0.000 MiB       secs = end_
    63  133.461 MiB    0.000 MiB       print(calcu
                                               + " t
    64
    66  133.461 MiB    0.000 MiB       assert sum
```

Another way to visualize the change in memory use is to sample over time and plot the result. `memory_profiler` has a utility called `mprof`, used once to sample the memory usage and a second time to visualize the samples. It samples by time and not by line, so it barely impacts the runtime of the code.

Figure 2-6 is created using `mprof run julia1_memoryprofiler.py`. This writes a statistics file that is then visualized using `mprof plot`. Our two functions are bracketed: this shows where in time they are entered, and we can see the growth in RAM as they run. Inside `calculate_z_serial_purepython`, we can see the steady increase in RAM usage throughout the execution of the function; this is caused by all the small objects (`int` and `float` types) that are created.

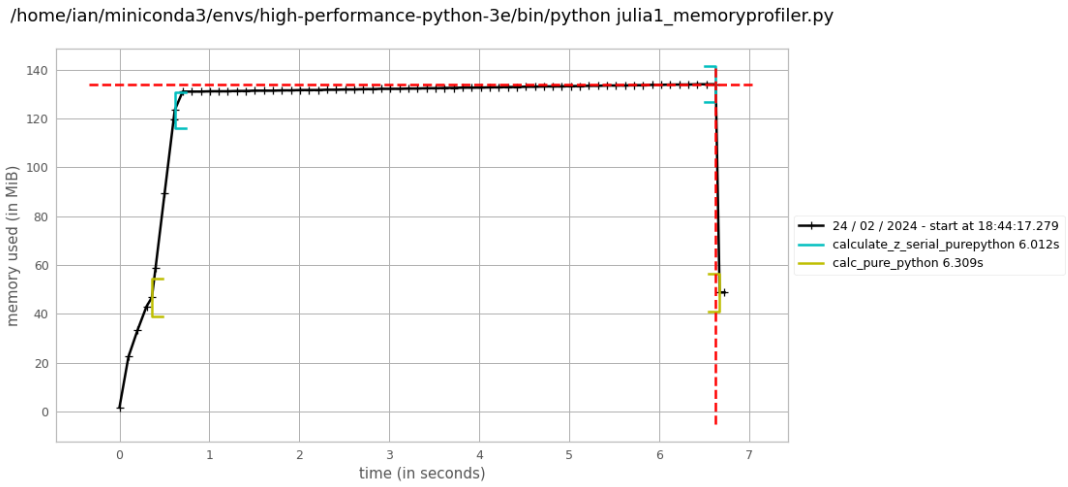


Figure 2-6. `memory_profiler` report using `mprof`

In addition to observing the behavior at the function level, we can add labels using a context manager. The snippet in Example 2-11 is used to generate the graph in Figure 2-7. We can see the `create_output_list` label: it appears momentarily at around 1.5 seconds after `calculate_z_serial_purepython` and results in the process being allocated more RAM. We then pause for a second;

`time.sleep(1)` is an artificial addition to make the graph easier to understand.

Example 2-11. Using a context manager to add labels to the `mprof` graph

```
@profile
def calculate_z_serial_purepython(maxiter, zs, cs
    """Calculate output list using Julia update r
    with profile.timestamp("create_output_list")
        output = [0] * len(zs)
    time.sleep(1)
    with profile.timestamp("calculate_output"):
        for i in range(len(zs)):
            n = 0
            z = zs[i]
            c = cs[i]
            while n < maxiter and abs(z) < 2:
                z = z * z + c
                n += 1
            output[i] = n
    return output
```

In the `calculate_output` block that runs for most of the graph, we see a very slow, linear increase in RAM usage. This will be from all of the temporary numbers used in the inner loops. Using the labels really helps us

to understand at a fine-grained level where memory is being consumed. Interestingly, we see the "peak RAM usage" line—a dashed vertical line just before the 10-second mark—occurring before the termination of the program. Potentially this is due to the garbage collector recovering some RAM from the temporary objects used during `calculate_output`.

What happens if we simplify our code and remove the creation of the `zs` and `cs` lists? We then have to calculate these coordinates inside `calculate_z_serial_purepython` (so the same work is performed), but we'll save RAM by not storing them in lists. You can see the code in Example 2-12.

In Figure 2-8, we see a major change in behavior—the overall envelope of RAM usage drops from 140 MB to 60 MB, reducing our RAM usage by half!

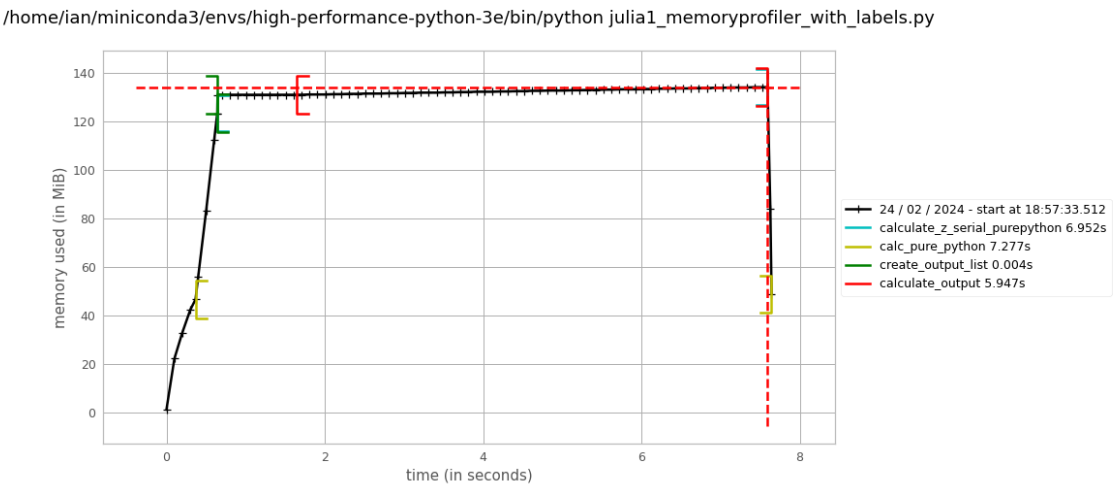


Figure 2-7. `memory_profiler` report using `mprof` with labels

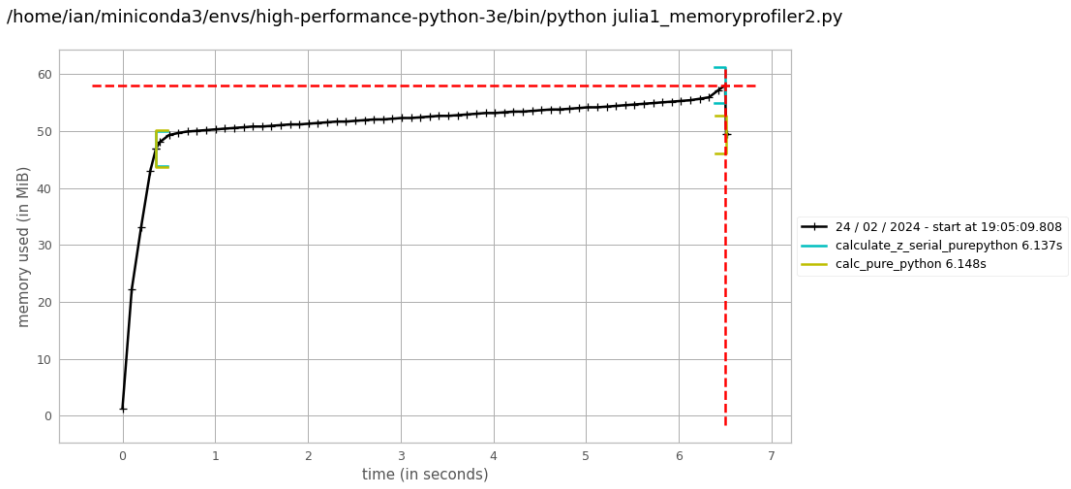


Figure 2-8. `memory_profiler` after removing two large lists

Example 2-12. Creating complex coordinates on the fly to save RAM

```
@profile
def calculate_z_serial_purepython(maxiter, x, y)
    """Calculate output list using Julia update r
    output = []
    for ycoord in y:
        for xcoord in x:
            z = complex(xcoord, ycoord)
            c = complex(c_real, c_imag)
            n = 0
            while n < maxiter and abs(z) < 2:
                z = z * z + c
                n += 1


            output.append(n)
    return output
```

If we want to measure the RAM used by several statements, we can use the IPython magic `%memit`, which works just like `%timeit`. In [Link to Come], we will look at using `%memit` to measure the memory cost of lists and discuss various ways of using RAM more efficiently.

`memory_profiler` offers an interesting aid to debugging a large process via the `--pdb-mmem=XXX` flag. The `pdb` debugger will be activated after the process exceeds `XXX` MB. This will drop you in directly at the point in your code where too many allocations are occurring, if you're in a space-constrained environment.

Combining CPU and Memory Profiling with Scalene

Scalene combines both CPU and memory profiling and adds GPU profiling, into one easy-to-run package. It runs on all platforms and is installed with `pip install scalene`.

Scalene uses its own lightweight profiler library so it has very little impact on execution speed (unlike both `memory_profiler` and `line_profiler`). We can invoke it with `scalene`

`julia1_memoryprofiler.py` and it will profile the whole program without needing any modification.

It has options to restrict profiling with an `@profile` decorator (similar to `line_profiler` and `memory_profiler`) and to filter in or out certain filenames. A magic extension is provided for use in Jupyter Notebooks with `%%scalene` to profile a whole cell.

In the screenshot in Figure 2-9, looking at the left-most "TIME" column we see that the `while` loop occupies 85% of the execution time, shown using a large horizontal bar. This bar has three blue colour components, the longest (and darkest) shows native Python time, two smaller blue chunks on the right edge (both with lighter shades) represent native and system execution time.

Native Python time refers to the time spent executing Python instructions, native time refers to libraries (i.e. libraries writtin in C++ and compiled) which the developer is unlikely to optimise. System times refers to operating system calls such as I/O which again probably can't be optimised by the developer, but may highlight I/O bottlenecks which can be approached in a different way.

The second "MEMORY" column shows the peak memory allocation per line. We can see that `zs.append(complex(xcoord, ycoord))` uses 40MB whilst `cs.append(complex(c_real, c_imag))` uses

31MB. Different runs on this produce different peak-allocation numbers and sometimes `cs` takes more than `zs`, so be cautious about forming a strong opinion without making multiple runs.

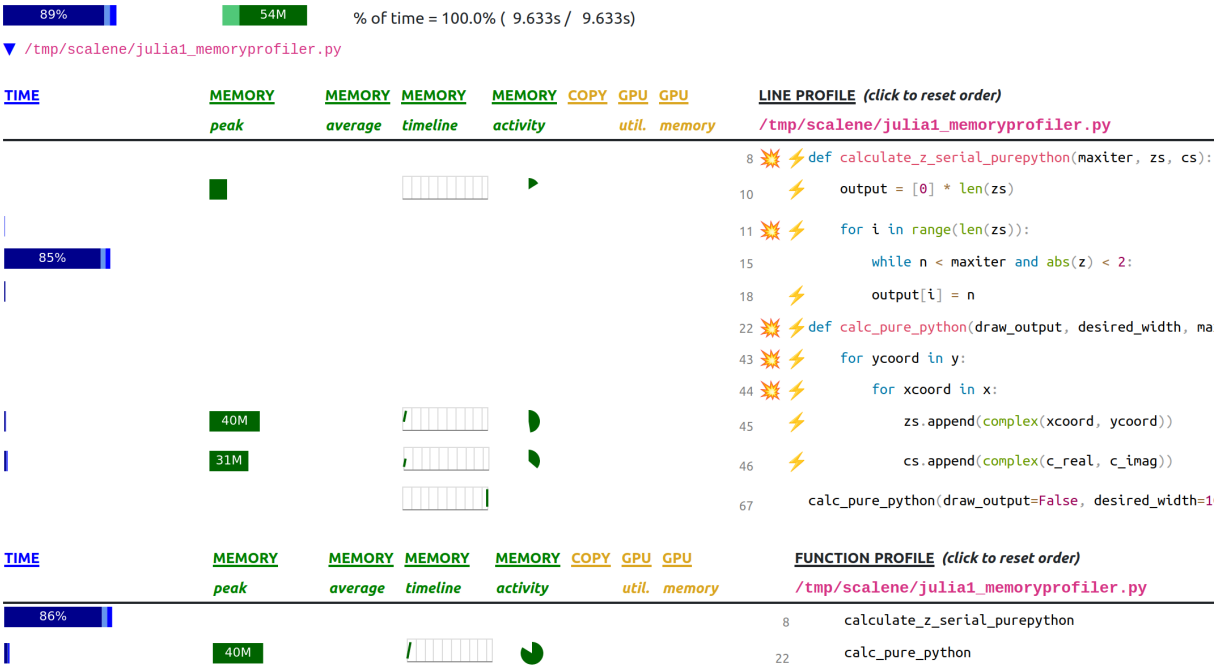


Figure 2-9. Combined CPU and Memory Profiler output with Scalene

Many of the lines have either an “explosion icon” or a “lightning icon”, Scalene creates calls to ChatGPT if you provide an API key so that optimisations can be automatically proposed.

Scalene can generate both a console-only (text) output as well as an interactive HTML report. The snippet below shows the console report with two extra lines added making NumPy arrays from the pure-Python list containers.

Both cause a duplication in the underlying data into `complex128` (8 byte float * 2) 16 byte `dtype`. With 1,000,000 elements in each, these lines cause a 16MB copy of each array to be constructed. A user might not realise this as they code (or during a code review may not appreciate what's happening) whilst a profiler clearly brings out the memory dupliation that's occurring.

Example 2-13. Console output from Scalene with two NumPy arrays added

```
  Line ... Python |peak |timeline/%
  ...
  49   |...   20%  | 19M |▁  15% | zs_np = np.array
  50   |...        | 15M |▁  12% | cs_np = np.array
```

You should expect to see different results to any you record with `line_profiler` or `memory_profiler`. Each profiling tools has a different impact and typically looks at CPU and memory usage in slightly different ways and with different resolutions - the big-scale picture will be similar but the details will be different.

Scalene's lightweight profiling library should add an overhead of under 20% to your execution speed and it should also reasonably measure the actual time and memory cost - the authors claim that it is potentially far more accurate than other profilers.

Introspecting an Existing Process with PySpy

`py-spy` is an intriguing new sampling profiler—rather than requiring any code changes, it introspects an already-running Python process and reports in the console with a `top`-like display. Being a sampling profiler, it has almost no runtime impact on your code. It is written in Rust and requires elevated privileges to introspect another process. It can also profile subprocesses and native compiled extensions.

Note that at the time of writing PySpy only runs up to Python 3.11, Python 3.12 support is forthcoming [2].

This tool could be very useful in a production environment with long-running processes or complicated installation requirements. It supports Windows, Mac, and Linux. Install it using `pip install py-spy` (note the dash in the name—there's a separate `pyspy` project that isn't related).

If your process is already running, you'll want to use `ps` to get its process identifier (the PID); then this can be passed into `py-spy` as shown in Example 2-14. `py-spy` needs `sudo` privileges which would run it in the superuser's environment, so we pass in our login's `PATH` using `sudo`

`env "PATH=$PATH"` when calling `py-spy` along with the process identifier to monitor.

Example 2-14. Running PySpy at the command line

```
$ ps -A -o pid,rss,cmd | ack python
...
95671 94984 python julia1_nopil.py
...
$ sudo env "PATH=$PATH" py-spy top --pid 95671
```

In Figure 2-10, you'll see a static picture of a `top`-like display in the console; this updates every second to show which functions are currently taking most of the time.

```
Collecting samples from 'python julia1_nopil.py' (python v3.11.7)
Total Samples 7800
GIL: 100.00%, Active: 100.00%, Threads: 1

  %Own   %Total  OwnTime  TotalTime  Function (filename:line)
 49.00%  49.00%   42.48s    42.48s   calculate_z_serial_purepython (julia1_nopil.py:16)
 33.00%  33.00%   24.33s    24.33s   calculate_z_serial_purepython (julia1_nopil.py:17)
 18.00%  18.00%   11.18s    11.18s   calculate_z_serial_purepython (julia1_nopil.py:18)
  0.00%   0.00%   0.010s    0.010s   calculate_z_serial_purepython (julia1_nopil.py:14)
  0.00% 100.00%   0.000s    78.00s   <module> (julia1_nopil.py:64)
  0.00% 100.00%   0.000s    78.00s   calc_pure_python (julia1_nopil.py:50)
```

Figure 2-10. Introspecting a Python process using PySpy

PySpy can also export a flame chart. Here, we'll run that option while asking PySpy to run our code directly without requiring a PID using `$ py-spy record -o profile.svg -- python`

`julia1_nopil.py`. The `--` tells `py-spy` to take the command that follows (`python julia1_nopil.py`), which might have its own command line arguments, and ignore any command line switches - it'll execute that command from inside `py-spy`.

You'll see in Figure 2-11 that the width of the display represents the entire program's runtime, and each layer moving down the image represents functions called from above.

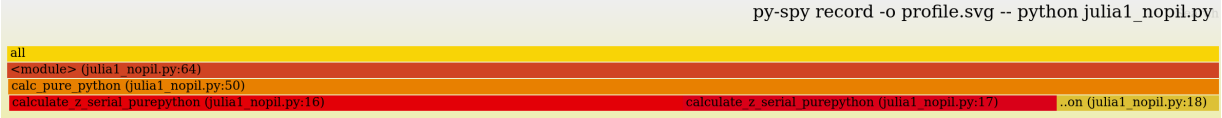


Figure 2-11. Part of a flame chart for PySpy

PySpy also has the useful ability to hook into an already-running Python process so if you have a web server or a long-running extract-transform-load process, you can connect to that process and introspect where in the program the time seems to be being spent. This allows for in-situ profiling when something unexpected has occurred, without having to setup a new profiling run.

VizTracer for an interactive time-based call stack

VizTracer is a two-part tool to profile and then view a time-based view of Python code execution. This tool is very powerful, but can be a little tricky to learn. The fact that it displays time left-to-right and enables you to zoom in on a flame-like chart gives you a birds-eye view of the entire process, so you can get a feel for how long parts of the execution path take.

It can track the execution, arguments and return values and time of every function call. It allows custom logging information, it supports multiprocesses and threads, it doesn't require source code changes, it has an API and it provides an interactive graphical tool.

We install it with `pip install viztracer` and then we can run the same code we've used previously with `viztracer julia1_memoryprofiler.py`.

Having profiled code using `viztracer` a `results.json` file is output. We next visualise this with `vizviewer results.json`, for our Julia code we get a result like that shown in Figure 2-12. The screen has the following sections:

- The very top is an interactive timeline with grab bars to slide and zoom the view to a specific point in time
- "MainProcess" shows a horizontal timeline of the code's execution and the function calls hanging down the screen in bars

- “Current Selection” gives details of anything that’s clicked in “MainProcess”

VizTracer can track the execution of multiple threads and multiple processes, these would be displayed alongside “MainProcess”. VizTracer can also log other time-based events such a `print` statements and calls to the garbage collector, enabling you to annotate specific events that you’d like to observe.

The two long bars in the centre of the screenshot under “MainProcess” detail the time based view on `calc_pure_python` and its call into `calcualte_z_serial_purepython`, below this are many tiny spikes - these are some of the individual calls to `abs`.

Having clicked `calculate_z_serial_purepython` the “Details” section in the bottom-left shows the function name and a start-time and duration, plus thread-based details. To the right of this is a view of the source code that was executed.

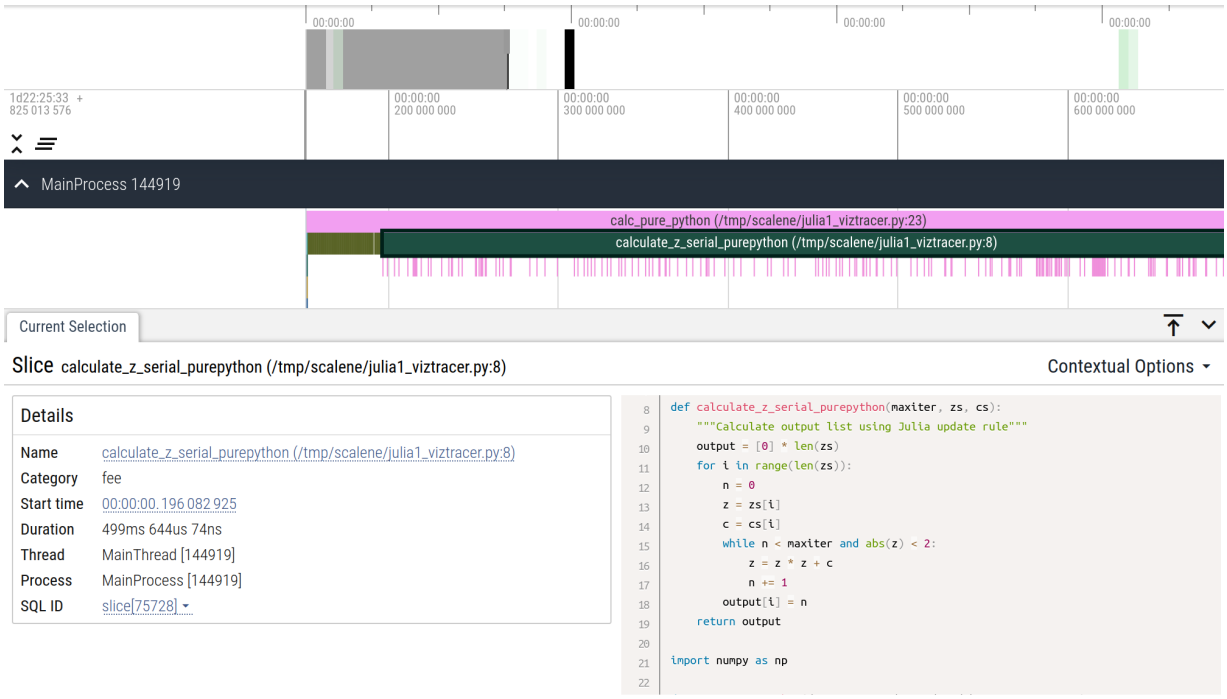


Figure 2-12. Overview of the Julia execution laid out through time

Due to the high level of detail it can track it can potentially generate unfeasibly large information dumps which can't be viewed, this can generate a "Circular buffer is full" error.

VizTracer provides various tools to filter the information that's collected, for the Julia example we had to:

- reduce the `calc_pure_python` argument from `1000` to a `desired_width=200`
- filter shorter calls when profiling with `viztracer --min_duration 0.1 julia1_viztracer.py`

Other filtering options can include `--ignore_c_function` which includes all built-in functions (which would ignore every `abs` call) or limiting the stack depth with `--max_stack_depth 10`. In this case we aren't witnessing a deep stack but instead a large number of calls to the fast `abs` function.

In order to get enough detail, but not overload the circular buffer, here we limited the profiling to functions taking longer than 0.1ms - interestingly the calls to `abs` could sometimes take longer (but frequently were much faster), so we caught some in this trace. Without this filter we'd have millions to track and that was the cause of this particular "Circular buffer is full" error.

Author Ian has found this tool very useful to dig into the execution path in libraries like Pandas, so we can see how long certain parts of the functions take and how many other functions are called behind the scenes.

Bytecode: Under the Hood

So far we've reviewed various ways to measure the cost of Python code (for both CPU and RAM usage). We haven't yet looked at the underlying bytecode used by the virtual machine, though. Understanding what's going on "under the hood" helps to build a mental model of what's happening in

slow functions, and it'll help when you come to compile your code. So let's introduce some bytecode.

Using the dis Module to Examine CPython Bytecode

The `dis` module lets us inspect the underlying bytecode that we run inside the stack-based CPython virtual machine. Having an understanding of what's happening in the virtual machine that runs your higher-level Python code will help you to understand why some styles of coding are faster than others. It will also help when you come to use a tool like Cython, which steps outside of Python and generates C code.

The `dis` module is built in. You can pass it code or a module, and it will print out a disassembly. In Example 2-15, we disassemble the outer loop of our CPU-bound function.

TIP

You should try to disassemble one of your own functions and to follow *exactly* how the disassembled code matches to the disassembled output. Can you match the following `dis` output to the original function?

Example 2-15. Using the built-in `dis` to understand the underlying stack-based virtual machine that runs our Python code

```
In [1]: import dis
In [2]: import julia1_nopil
In [3]: dis.dis(julia1_nopil.calculate_z_serial_p
  9           0 RESUME                   0

 11           2 LOAD_CONST               1 (0)
              4 BUILD_LIST               1
              6 LOAD_GLOBAL              1 (NULL
             16 LOAD_FAST                1 (zs)
             18 CALL                     1
             26 BINARY_OP                5 (*)
             30 STORE_FAST               3 (outpu

 12          32 LOAD_GLOBAL              3 (NULL
             42 LOAD_GLOBAL              1 (NULL
             52 LOAD_FAST                1 (zs)
             54 CALL                     1
             62 CALL                     1
             70 GET_ITER
        >>   72 FOR_ITER                71 (to 21
             76 STORE_FAST               4 (i)

 13          78 LOAD_CONST               1 (0)
             80 STORE_FAST               5 (n)

...

 16         166 LOAD_GLOBAL              5 (NULL
            176 LOAD_FAST                6 (z)
```

```
              178 CALL                 1
              186 LOAD_CONST           2 (2)
              188 COMPARE_OP           2 (<)
              192 POP_JUMP_IF_FALSE    6 (to 20
              194 LOAD_FAST            5 (n)
              196 LOAD_FAST            0 (maxit
              198 COMPARE_OP           2 (<)
              202 POP_JUMP_IF_FALSE    1 (to 20
              204 JUMP_BACKWARD       33 (to 14

   19     >>  206 LOAD_FAST            5 (n)
              208 LOAD_FAST            3 (outpu
              210 LOAD_FAST            4 (i)
              212 STORE_SUBSCR
              216 JUMP_BACKWARD       73 (to 72

   12     >>  218 END_FOR

   20         220 LOAD_FAST            3 (outpu
              222 RETURN_VALUE
```

The output is fairly straightforward, if terse. The first column contains line numbers that relate to our original file. The second column contains several `>>` symbols; these are the destinations for jump points elsewhere in the code. The third column is the operation address; the fourth has the operation name. The fifth column contains the parameters for the operation. The sixth

column contains annotations to help line up the bytecode with the original Python parameters.

Refer back to Example 2-3 to match the bytecode to the corresponding Python code. The bytecode starts on Python line 11 by putting the constant value 0 onto the stack, and then it builds a single-element list. Next, it searches the namespaces to find the `len` function, puts it on the stack, searches the namespaces again to find `zs`, and then puts that onto the stack. Now a multiply can be performed, consuming the stack items, which are then stored in `output`. That's the first line of our Python function now dealt with. Follow the next block of bytecode to understand the behavior of the second line of Python code (the outer `for` loop).

TIP

The jump points (`>>`) match to instructions like `JUMP_BACKWARD` and `POP_JUMP_IF_FALSE`. Go through your own disassembled function and match the jump points to the jump instructions.

Digging into bytecode specialisation with Specialist

When Python 3.11 runs your code it attempts to identify "hot" code which is run frequently enough to warrant optimization. When possible it will use "specialised" bytecode instructions to replace the more general bytecode

that has been used up to Python 3.10. The specialised bytecode can run faster by doing less work (perhaps avoiding redundant checks or via other optimisations). It can be installed with `pip install specialist`.

Specialisations hold true as long as the same types are encountered - if an addition is performed on two integers in a loop, that operation can be specialised (and would be coloured green). If however the addition sees different types (maybe integers or floating point types depending on the iteration) then it'll stay in an "adaptive" state which is coloured red.

Specialist colour codes the "hot" regions of code which Python 3.11+ identifies during execution, allowing us to review if our code is running as fast as might be possible. Using the Julia example Figure 2-13 we can see:

- lines in white - no specialisation has been attempted
- lines in green - successfully specialised, they're running fast
- sections in orange - part specialised which sometimes fail and revert to being adaptive
- in our example none are colour red (indicating a lack of specialisation)

The `abs(z) < 2` can be made "more green" (i.e. more specialised) by changing it to `abs(z) < 2.0`, as Python 3.12 currently prefers identical numerical types for certain primitive operations. The complex-type operation `z * z + c` refused to be improved.

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Figure 2-13. Viewing coloured output from Specialist

In this case making the `2.0` change did improve performance a little. In the current version of Python the Specialist tool is more for education about what's happening behind the scenes, but experiments may reveal opportunity to improve execution speed.

Having introduced bytecode, we can now ask: what's the bytecode and time cost of writing a function out explicitly versus using built-ins to perform the same task?

Different Approaches, Different Complexity

> *There should be one—and preferably only one—obvious way to do it. Although that way may not be obvious at first unless you're Dutch.*[3]
>
> —Tim Peters, The Zen of Python

There will be various ways to express your ideas using Python. Generally, the most sensible option should be clear, but if your experience is primarily with an older version of Python or another programming language, you may have other methods in mind. Some of these ways of expressing an idea may be slower than others.

You probably care more about readability than speed for most of your code, so your team can code efficiently without being puzzled by performant but opaque code. Sometimes you will want performance, though (without sacrificing readability). Some speed testing might be what you need.

Take a look at the two code snippets in Example 2-16. Both do the same job, but the first generates a lot of additional Python bytecode, which will cause more overhead.

Example 2-16. A naive and a more efficient way to solve the same summation problem

```
def fn_expressive(upper=1_000_000):
    total = 0
    for n in range(upper):
        total += n
    return total

def fn_terse(upper=1_000_000):
    return sum(range(upper))
```

```
  assert fn_expressive() == fn_terse(), "Expect ide
```

Both functions calculate the sum of a range of integers. A simple rule of thumb (but one you *must* back up using profiling!) is that more lines of bytecode will execute more slowly than fewer equivalent lines of bytecode that use built-in functions. In Example 2-17, we use IPython's `%timeit` magic function to measure the best execution time from a set of runs. `fn_terse` runs over twice as fast as `fn_expressive`!

Example 2-17. Using `%timeit` to test our hypothesis that using built-in functions should be faster than writing our own functions

```
In [2]: %timeit fn_expressive()
45.6 ms ± 963 µs per loop (mean ± std. dev. of 7

In [3]: %timeit fn_terse()
16.4 ms ± 226 µs per loop (mean ± std. dev. of 7
```

If we use the `dis` module to investigate the code for each function, as shown in Example 2-18, we can see that the virtual machine has 17 lines to execute with the more expressive function and only 7 to execute with the very readable but terser second function.

Example 2-18. Using `dis` to view the number of bytecode instructions involved in our two functions

```
In [4]: import dis

In [5]: dis.dis(fn_expressive)
  1           0 RESUME                   0

  2           2 LOAD_CONST               1 (0)
              4 STORE_FAST               1 (total

  3           6 LOAD_GLOBAL              1 (NULL
             16 LOAD_FAST                0 (upper
             18 CALL                     1
             26 GET_ITER
        >>   28 FOR_ITER                 7 (to 46
             32 STORE_FAST               2 (n)

  4          34 LOAD_FAST                1 (total
             36 LOAD_FAST                2 (n)
             38 BINARY_OP               13 (+=)
             42 STORE_FAST               1 (total
             44 JUMP_BACKWARD            9 (to 28

  3     >>   46 END_FOR

  5          48 LOAD_FAST                1 (total
             50 RETURN_VALUE


In [6]: dis.dis(fn_terse)
```

```
  7           0 RESUME                   0

  8           2 LOAD_GLOBAL              1 (NULL
             12 LOAD_GLOBAL              3 (NULL
             22 LOAD_FAST                0 (uppe
             24 CALL                     1
             32 CALL                     1
             40 RETURN_VALUE
```

The difference between the two code blocks is striking. Inside `fn_expressive()`, we maintain two local variables and iterate over a list using a `for` statement. The `for` loop will be checking to see if a `StopIteration` exception has been raised on each loop. Each iteration applies the `total.__add__` function, which will check the type of the second variable (`n`) on each iteration. These checks all add a little expense.

Inside `fn_terse()`, we call out to an optimized C list comprehension function that knows how to generate the final result without creating intermediate Python objects. This is much faster, although each iteration must still check for the types of the objects that are being added together (in Chapter 4, we look at ways of fixing the type so we don't need to check it on each iteration).

As noted previously, you *must* profile your code—if you just rely on this heuristic, you will inevitably write slower code at some point. It is definitely worth learning whether a shorter and still readable way to solve

your problem is built into Python. If so, it is more likely to be easily readable by another programmer, and it will *probably* run faster.

Unit Testing During Optimization to Maintain Correctness

If you aren't already unit testing your code, you are probably hurting your longer-term productivity. Ian (blushing) is embarrassed to note that he once spent a day optimizing his code, having disabled unit tests because they were inconvenient, only to discover that his significant speedup result was due to breaking a part of the algorithm he was improving. You do not need to make this mistake even once.

TIP

Add unit tests to your code for a saner life. You'll be giving your current self and your colleagues faith that your code works, and you'll be giving a present to your future-self who has to maintain this code later. You really will save a lot of time in the long term by adding tests to your code.

In addition to unit testing, you should also strongly consider using `coverage.py`. It checks to see which lines of code are exercised by your tests and identifies the sections that have no coverage. This quickly lets you figure out whether you're testing the code that you're about to

optimize, such that any mistakes that might creep in during the optimization process are quickly caught.

No-op @profile Decorator

Your unit tests will fail with a `NameError` exception if your code uses `@profile` from `line_profiler` or `memory_profiler`. The reason is that the unit test framework will not be injecting the `@profile` decorator into the local namespace. The no-op decorator shown here solves this problem. It is easiest to add it to the block of code that you're testing and remove it when you're done.

With the no-op decorator, you can run your tests without modifying the code that you're testing. This means you can run your tests after every profile-led optimization you make so you'll never be caught out by a bad optimization step.

Let's say we have the trivial `ex.py` module shown in Example 2-19. It has a test (for `pytest`) and a function that we've been profiling with either `line_profiler` or `memory_profiler`.

Example 2-19. Simple function and test case where we wish to use `@profile`

```
  import time
```

```
def test_some_fn():
    """Check basic behaviors for our function"""
    assert some_fn(2) == 4
    assert some_fn(1) == 1
    assert some_fn(-1) == 1


@profile
def some_fn(useful_input):
    """An expensive function that we wish to both
    # artificial "we're doing something clever ar
    time.sleep(1)
    return useful_input ** 2


if __name__ == "__main__":
    print(f"Example call `some_fn(2)` == {some_fr
```

If we run `pytest` on our code, we'll get a `NameError`, as shown in Example 2-20.

Example 2-20. A missing decorator during testing breaks out tests in an unhelpful way!

```
$ pytest test_utility.py
== test session starts ==
```

```
platform linux -- Python 3.12.0, pytest-8.0.1, pl
rootdir: .../ch02/noop_profile_decorator
collected 0 items / 1 error

== ERRORS ==
__ ERROR collecting test_utility.py __
test_utility.py:1: in <module>
    from utility import some_fn
utility.py:20: in <module>
    @profile
E   NameError: name 'profile' is not defined. Did
== short test summary info ==
ERROR test_utility.py - NameError: name 'profile
Did you forget to import 'profile'
!! Interrupted: 1 error during collection !!
== 1 error in 0.12s ==
```

The solution is to add a no-op decorator at the start of our module (you can remove it after you're done with profiling). If the `@profile` decorator is not found in one of the namespaces (because `line_profiler` or `memory_profiler` is not being used), the no-op version we've written is added. If `line_profiler` or `memory_profiler` has injected the new function into the namespace, our no-op version is ignored.

For both `line_profiler` and `memory_profiler`, we can add the code in Example 2-21.

Example 2-21. Add a no-op `@profile` decorator to the namespace while unit testing

```
# check for line_profiler or memory_profiler in t
# are injected by their respective tools or they
# if these tools aren't being used (in which case
# a dummy @profile decorator)
if 'line_profiler' not in dir() and 'profile' not
    def profile(func):
        def inner(*args, **kwargs):
            return func(*args, **kwargs)
        return inner
```

Having added the no-op decorator, we can now run our `pytest` successfully, as shown in Example 2-22, along with our profilers—with no additional code changes.

Example 2-22. With the no-op decorator, we have working tests, and both of our profilers work correctly

```
$ pytest utility.py
== test session starts ==
platform linux -- Python 3.12.0, pytest-8.0.1, pl
rootdir: ...
collected 1 item

utility.py .
```

```
== 1 passed in 3.01s ==

$ kernprof -l -v utility.py
Example call `some_fn(2)` == 4
Wrote profile results to utility.py.lprof
Timer unit: 1e-06 s

Total time: 1.00018 s
File: utility.py
Function: some_fn at line 20

Line # Hits Per Hit % Time  Line Contents
==============================================
    20                       @profile
    21                       def some_fn(useful_i
    22                           """An expensive
                                    to both test a
    23                           # artificial 'we
                                   clever and expe
    24    1   1e+06  100.0      time.sleep(1)
    25    1     6.5    0.0      return useful_inp



$ python -m memory_profiler utility.py
Example call `some_fn(2)` == 4
Filename: utility.py
```

```
Line #  Mem usage  Increment  Line Contents
================================================
    20 46.898 MiB 46.898 MiB @profile
    21                       def some_fn(useful_i
    22                           """An expensive
                                    to both test
    23                           # artificial 'we
                                   clever and exp
    24 46.898 MiB  0.000 MiB     time.sleep(1)
    25 46.898 MiB  0.000 MiB     return useful_in
```

You can save yourself a few minutes by avoiding the use of these decorators, but once you've lost hours making a false optimization that breaks your code, you'll want to integrate this into your workflow.

Strategies to Profile Your Code Successfully

Profiling requires some time and concentration. You will stand a better chance of understanding your code if you separate the section you want to test from the main body of your code. You can then unit test the code to preserve correctness, and you can pass in realistic fabricated data to exercise the inefficiencies you want to address.

Do remember to disable any BIOS-based accelerators, as they will only confuse your results. On Ian's laptop, the Intel Turbo Boost feature can temporarily accelerate a CPU above its normal maximum speed if it is cool enough. Running the Julia demo code from this chapter with Turbo Boost enabled on a cold CPU took 3.3 seconds and the CPU would temporarily have become hotter, if the code had been run repeatedly then as the CPU heated the execution time would have slowed. With Turbo Boost disabled that same piece of code took 5.6 seconds. In this example with Turbo Boost enabled a single run took 60% of the time compared to disabling it, and that gap would have closed as the CPU heated. Your operating system may also control the clock speed—a laptop on battery power is likely to more aggressively control CPU speed than a laptop on AC power.

You can imagine that this could confuse your bechmarking as you repeatedly run complex code! AMD has Turbo Boost, a similar technology. This applies to laptops, server machines may simply run at a maximum fixed speed (trading off complexity for increased power consumption) so this might not apply to servers.

To create a more stable benchmarking configuration, we do the following:

- Disable Turbo Boost in the BIOS.
- Disable the operating system's ability to override the SpeedStep (you will find this in your BIOS if you're allowed to control it).
- Use only AC power (never battery power).

- Disable background tools like backups and Dropbox while running experiments.
- Run the experiments many times to obtain a stable measurement.
- Possibly drop to run level 1 (Unix) so that no other tasks are running.
- Reboot and rerun the experiments to double-confirm the results.

Try to hypothesize the expected behavior of your code and then validate (or disprove!) the hypothesis with the result of a profiling step. Your choices will not change (you should only drive your decisions by using the profiled results), but your intuitive understanding of the code will improve, and this will pay off in future projects as you will be more likely to make performant decisions. Of course, you will verify these performant decisions by profiling as you go.

Do not skimp on the preparation. If you try to performance test code deep inside a larger project without separating it from the larger project, you are likely to witness side effects that will sidetrack your efforts. It is likely to be harder to unit test a larger project when you're making fine-grained changes, and this may further hamper your efforts. Side effects could include other threads and processes impacting CPU and memory usage and network and disk activity, which will skew your results.

Naturally, you're already using source code control (e.g., Git or Mercurial), so you'll be able to run multiple experiments in different branches without

ever losing “the versions that work well.” If you’re *not* using source code control, do yourself a huge favor and start to do so!

For web servers, investigate `dowser` and `dozer`; you can use these to visualize in real time the behavior of objects in the namespace. Definitely consider separating the code you want to test out of the main web application if possible, as this will make profiling significantly easier.

Make sure your unit tests exercise all the code paths in the code that you’re analyzing. Anything you don’t test that is used in your benchmarking may cause subtle errors that will slow down your progress. Use `coverage.py` to confirm that your tests are covering all the code paths.

Unit testing a complicated section of code that generates a large numerical output may be difficult. Do not be afraid to output a text file of results to run through `diff` or to use a `pickled` object. For numeric optimization problems, Ian likes to create long text files of floating-point numbers and use `diff`—minor rounding errors show up immediately, even if they’re rare in the output.

If your code might be subject to numerical rounding issues due to subtle changes, you are better off with a large output that can be used for a before-and-after comparison. One cause of rounding errors is the difference in floating-point precision between CPU registers and main memory. Running your code through a different code path can cause subtle rounding errors

that might later confound you—it is better to be aware of this as soon as they occur.

Obviously, it makes sense to use a source code control tool while you are profiling and optimizing. Branching is cheap, and it will preserve your sanity.

Wrap-Up

Having looked at profiling techniques, you should have all the tools you need to identify bottlenecks around CPU and RAM usage in your code. Next, we'll look at how Python implements the most common containers, so you can make sensible decisions about representing larger collections of data.

`memory_profiler` measures memory usage according to the International Electrotechnical Commission's MiB (mebibyte) of 2^{20} bytes. This is slightly different from the more common but also more ambiguous MB (megabyte has two commonly accepted definitions!). 1 MiB is equal to 1.048576 (or approximately 1.05) MB. For our discussion, unless we're dealing with very specific amounts, we'll consider the two equivalent.

Python 3.12 release information: *https://github.com/benfred/py-spy/issues/633*

The language creator Guido van Rossum is Dutch, and not everyone has agreed with his "obvious" choices, but on the whole we like the choices that Guido makes!

Chapter 3. Lists and Tuples

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

QUESTIONS YOU'LL BE ABLE TO ANSWER AFTER THIS CHAPTER

- What are lists and tuples good for?
- What is the complexity of a lookup in a list/tuple?
- How is that complexity achieved?
- What are the differences between lists and tuples?
- How does appending to a list work?
- When should I use lists and tuples?

One of the most important things in writing efficient programs is understanding the guarantees of the data structures you use. In fact, a large part of performant programming is knowing what questions you are trying to ask of your data and picking a data structure that can answer these questions quickly. In this chapter we will talk about the kinds of questions that lists and tuples can answer quickly, and how they do it.

Lists and tuples are a class of data structures called *arrays*. An array is a flat list of data with some intrinsic ordering. Usually in these sorts of data structures, the relative ordering of the elements is as important as the elements themselves! In addition, this *a priori* knowledge of the ordering is incredibly valuable: by knowing that data in our array is at a specific position, we can retrieve it in `O(1)`![1] There are also many ways to implement arrays, and each solution has its own useful features and guarantees. This is why in Python we have two types of arrays: lists and tuples. *Lists* are dynamic arrays that let us modify and resize the data we are storing, while *tuples* are static arrays whose contents are fixed and immutable.

Let's unpack these previous statements a bit. System memory on a computer can be thought of as a series of numbered buckets, each capable of holding a number. Python stores data in these buckets *by reference*, which means the number itself simply points to, or refers to, the data we actually care about. As a result, these buckets can store any type of data we

want (as opposed to `numpy` arrays, which have a static type and can store only that type of data).[2]

When we want to create an array (and thus a list or tuple), we first have to allocate a block of system memory (where every section of this block will be used as an integer-sized pointer to actual data). This involves going to the system kernel and requesting the use of `N` *consecutive* buckets. Figure 3-1 shows an example of the system memory layout for an array (in this case, a list) of size 6.

NOTE

In Python, lists also store how large they are, so of the six allocated blocks, only five are usable—the zeroth element is the length.

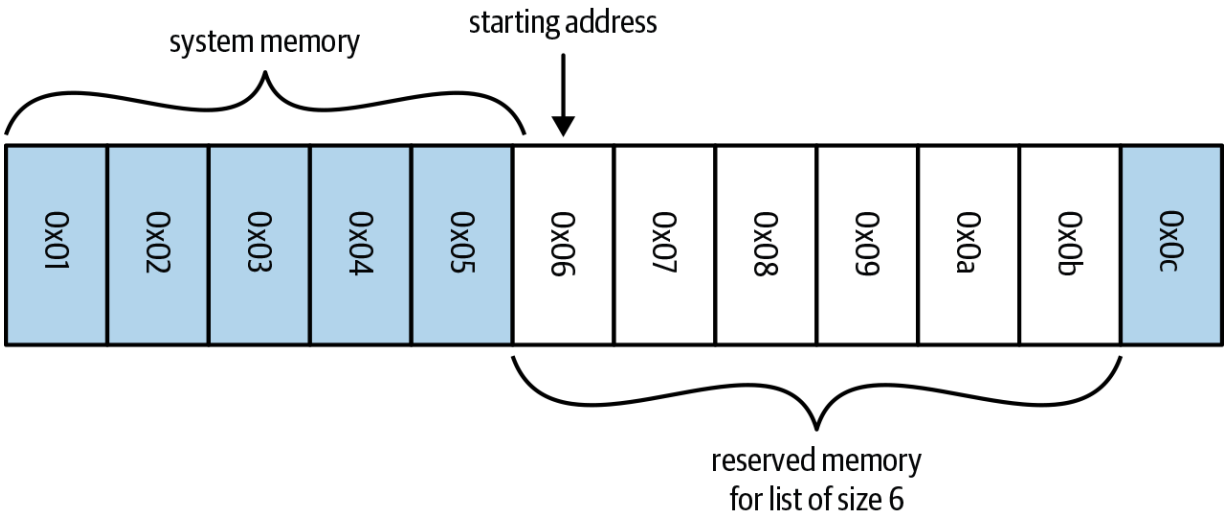


Figure 3-1. Example of system memory layout for an array of size 6

In order to look up any specific element in our list, we simply need to know which element we want and remember which bucket our data started in. Since all of the data will occupy the same amount of space (one "bucket," or, more specifically, one integer-sized pointer to the actual data), we don't need to know anything about the type of data that is being stored to do this calculation.

TIP

If you knew where in memory your list of `N` elements started, how would you find an arbitrary element in the list?

If, for example, we needed to retrieve the zeroth element in our array, we would simply go to the first bucket in our sequence, `M`, and read out the value inside it. If, on the other hand, we needed the fifth element in our array, we would go to the bucket at position `M + 5` and read its content. In general, if we want to retrieve element `i` from our array, we go to bucket `M + i`. So, by having our data stored in consecutive buckets, and having knowledge of the ordering of our data, we can locate our data by knowing which bucket to look at in one step (or `O(1)`), regardless of how big our array is (Example 3-1).

Example 3-1. Timings for lookups in lists of different sizes

```
>>> %%timeit l = list(range(10))
                 - ---
```

```
          ...: l[5]
          ...:
14 ns ± 0.182 ns per loop (mean ± std. dev. of 7

>>> %%timeit l = list(range(10_000_000))
          ...: l[100_000]
          ...:
13.9 ns ± 0.123 ns per loop (mean ± std. dev. of
```

What if we were given an array with an unknown order and wanted to retrieve a particular element? If the ordering were known, we could simply look up that particular value. However, in this case, we must do a `search` operation. The most basic approach to this problem is called a *linear search*, where we iterate over every element in the array and check if it is the value we want, as seen in Example 3-2.

Example 3-2. A linear search through a list

```
def linear_search(needle, array):
    for i, item in enumerate(array):
        if item == needle:
            return i
    return -1
```

This algorithm has a worst-case performance of `O(n)`. This case occurs when we search for something that isn't in the array. In order to know that

the element we are searching for isn't in the array, we must first check it against every other element. Eventually, we will reach the final `return -1` statement. In fact, this algorithm is exactly the algorithm that `list.index()` uses.

The only way to increase the speed is by having some other understanding of how the data is placed in memory, or of the arrangement of the buckets of data we are holding. For example, hash tables ("How Do Dictionaries and Sets Work?"), which are a fundamental data structure powering dictionaries and sets, solve this problem in `O(1)` by adding extra overhead to insertions/retrievals and enforcing a strict and peculiar sorting of the item. Alternatively, if your data is sorted so that every item is larger (or smaller) than its neighbor to the left (or right), then specialized search algorithms can be used that can bring your lookup time down to `O(log n)`. This may seem like a huge performance penalty compared to the constant time *lookups* of lists and tuples, however sometimes using these search algorithms are the best solution (especially since search algorithms are flexible and allow you to define searches in creative ways).

EXERCISE

Given the following data, write an algorithm to find the index of the value `61`:

```
[9, 18, 18, 19, 29, 42, 56, 61, 88, 95]
```

Since you know the data is ordered, how can you do this faster?

Hint: If you split the array in half, you know all the values on the left are smaller than the smallest element in the right set. You can use this!

A More Efficient Search

As alluded to previously, we can achieve better search performance if we first sort our data so that all elements to the left of a particular item are smaller (or larger) than that item. The comparison is done through the `__eq__` and `__lt__` magic functions of the object and can be user-defined if using custom objects.

NOTE

Without the `__eq__` and `__lt__` methods, a custom object will compare only to objects of the same type, and the comparison will be done using the instance's placement in memory. With those two magic functions defined, you can use the `functools.total_ordering` decorator from the standard library to automatically define all the other ordering functions, albeit at a small performance penalty.

The two ingredients necessary are the sorting algorithm and the searching algorithm. Python lists have a built-in sorting algorithm that uses Tim sort. Tim sort can sort through a list in `O(n)` in the best case (and in `O(n log n)` in the worst case). It achieves this performance by utilizing multiple types of sorting algorithms and using heuristics to guess which algorithm will perform the best, given the data (more specifically, it hybridizes insertion and merge sort algorithms).

Once a list has been sorted, we can find our desired element using a binary search (Example 3-3), which has an average case complexity of `O(log n)`. It achieves this by first looking at the middle of the list and comparing this value with the desired value. If this midpoint's value is less than our desired value, we consider the right half of the list, and we continue halving the list like this until the value is found, or until the value is known not to occur in the sorted list. As a result, we do not need to read all values in the list, as was necessary for the linear search; instead, we read only a small subset of them.

Example 3-3. Efficient searching through a sorted list—binary search

```
def binary_search(needle, haystack):
    imin, imax = 0, len(haystack)
    while True:
        if imin > imax:
            return -1
        midpoint = (imin + imax) // 2
        if haystack[midpoint] > needle:
            imax = midpoint
        elif haystack[midpoint] < needle:
            imin = midpoint+1
        else:
            return midpoint
```

This method allows us to find elements in a list without resorting to the potentially heavyweight solution of a dictionary. This is especially true when the list of data that is being operated on is intrinsically sorted. It is more efficient to do a binary search on the list to find an object rather than first converting your data to a dictionary and then doing a single lookup on it. Although a dictionary lookup takes only `O(1)`, converting the data to a dictionary takes `O(n)` (and a dictionary's restriction of no repeating keys may be undesirable). On the other hand, the binary search will take `O(log n)`.

In addition, the `bisect` module from Python's standard library simplifies much of this process by giving easy methods to add elements into a list while maintaining its sorting, in addition to finding elements using a heavily optimized binary search. It does this by providing alternative functions that add the element into the correct sorted placement. With the list always being sorted, we can easily find the elements we are looking for (examples of this can be found in the documentation for the `bisect` module). In addition, we can use `bisect` to find the closest element to what we are looking for very quickly (Example 3-4). This can be extremely useful for comparing two datasets that are similar but not identical.

Example 3-4. Finding close values in a list with the `bisect` module

```
import bisect
import random

def find_closest(haystack, needle):
    # bisect.bisect_left will return the first va
    # that is greater than the needle
    i = bisect.bisect_left(haystack, needle)
    if i == len(haystack):
        return i - 1
    elif haystack[i] == needle:
        return i
    elif i > 0:
        j = i - 1
        # since we know the value is larger than
```

```
            # value at j), we don't need to use absol
            if haystack[i] - needle > needle - haysta
                return j
        return i

important_numbers = []
for i in range(10):
    new_number = random.randint(0, 1000)
    bisect.insort(important_numbers, new_number)

# important_numbers will already be in order beca
# with bisect.insort
print(important_numbers)
# > [14, 265, 496, 661, 683, 734, 881, 892, 973,

closest_index = find_closest(important_numbers, -
print(f"Closest value to -250: {important_numbers
# > Closest value to -250: 14

closest_index = find_closest(important_numbers, 5
print(f"Closest value to 500: {important_numbers
# > Closest value to 500: 496

closest_index = find_closest(important_numbers, 1
print(f"Closest value to 1100: {important_numbers
# > Closest value to 1100: 992
```

In general, this touches on a fundamental rule of writing efficient code: pick the right data structure and stick with it! Although there may be more efficient data structures for particular operations, the cost of converting to those data structures may negate any efficiency boost.

Lists Versus Tuples

If lists and tuples both use the same underlying data structure, what are the differences between the two? Summarized, the main differences are as follows:

- Lists are *dynamic* arrays; they are mutable and allow for resizing (changing the number of elements that are held).
- Tuples are *static* arrays; they are immutable, and the data they reference cannot change once the tuple has been created. [3]
- Tuples are cached by the Python runtime, which means that we don't need to talk to the kernel to reserve memory every time we want to use one.

These differences outline the philosophical difference between the two: tuples are for describing multiple properties of one unchanging thing, and lists can be used to store collections of data about completely disparate objects. For example, the parts of a telephone number are perfect for a tuple: they won't change, and if they do, they represent a new object or a

different phone number. Similarly, the coefficients of a polynomial fit a tuple, since different coefficients represent a different polynomial. On the other hand, the names of the people currently reading this book are better suited for a list: the data is constantly changing both in content and in size but is still always representing the same idea.

It is important to note that both lists and tuples can take mixed types. This can, as you will see, introduce quite a bit of overhead and reduce some potential optimizations. This overhead can be removed if we force all our data to be of the same type. In [Link to Come], we will talk about reducing both the memory used and the computational overhead by using `numpy`. In addition, tools like the standard library module `array` can reduce these overheads for other, nonnumerical situations. This alludes to a major point in performant programming that we will touch on in later chapters: generic code will be much slower than code specifically designed to solve a particular problem.

In addition, the immutability of a tuple as opposed to a list, which can be resized and changed, makes it a lightweight data structure. This means that there isn't much overhead in memory when storing tuples, and operations with them are quite straightforward. With lists, as you will learn, their mutability comes at the price of extra memory needed to store them and extra computations needed when using them.

EXERCISE

For the following example datasets, would you use a tuple or a list? Why?

1. First 20 prime numbers
2. Names of programming languages
3. A person's age, weight, and height
4. A person's birthday and birthplace
5. The result of a particular game of pool
6. The results of a continuing series of pool games

Solution:

1. Tuple, since the data is static and will not change.
2. List, since this dataset is constantly growing.
3. List, since the values will need to be updated.
4. Tuple, since that information is static and will not change.
5. Tuple, since the data is static.
6. List, since more games will be played. (In fact, we could use a list of tuples since each individual game's results will not change, but we will need to add more results as more games are played.)

Lists as Dynamic Arrays

Once we create a list, we are free to change its contents as needed:

```
>>> numbers = [5, 8, 1, 3, 2, 6]
>>> numbers[2] = 2 * numbers[0]   ❶
>>> numbers
[5, 8, 10, 3, 2, 6]
```

❶ As described previously, this operation is `O(1)` because we can find the data stored within the zeroth and second elements immediately.

In addition, we can append new data to a list and grow its size:

```
>>> len(numbers)
6
>>> numbers.append(42)
>>> numbers
[5, 8, 10, 3, 2, 6, 42]
>>> len(numbers)
7
```

This is possible because dynamic arrays support a `resize` operation that increases the capacity of the array. When a list of size `N` is first appended to, Python must create a new list that is big enough to hold the original `N` items in addition to the extra one that is being appended. However, instead of allocating exactly `N + 1` items, `M` items are actually allocated, where

`M > N`, in order to provide extra headroom for future appends. Then the data from the old list is copied to the new list, and the old list is destroyed.

The philosophy is that one append is probably the beginning of many appends, and by requesting extra space, we can reduce the number of total times this allocation must happen and thus the total number of memory copies that are necessary. This is important since memory copies can be quite expensive, especially when list sizes start growing. Figure 3-2 shows what this overallocation looks like in Python 3.7. The formula dictating this growth is given in Example 3-5.[4]

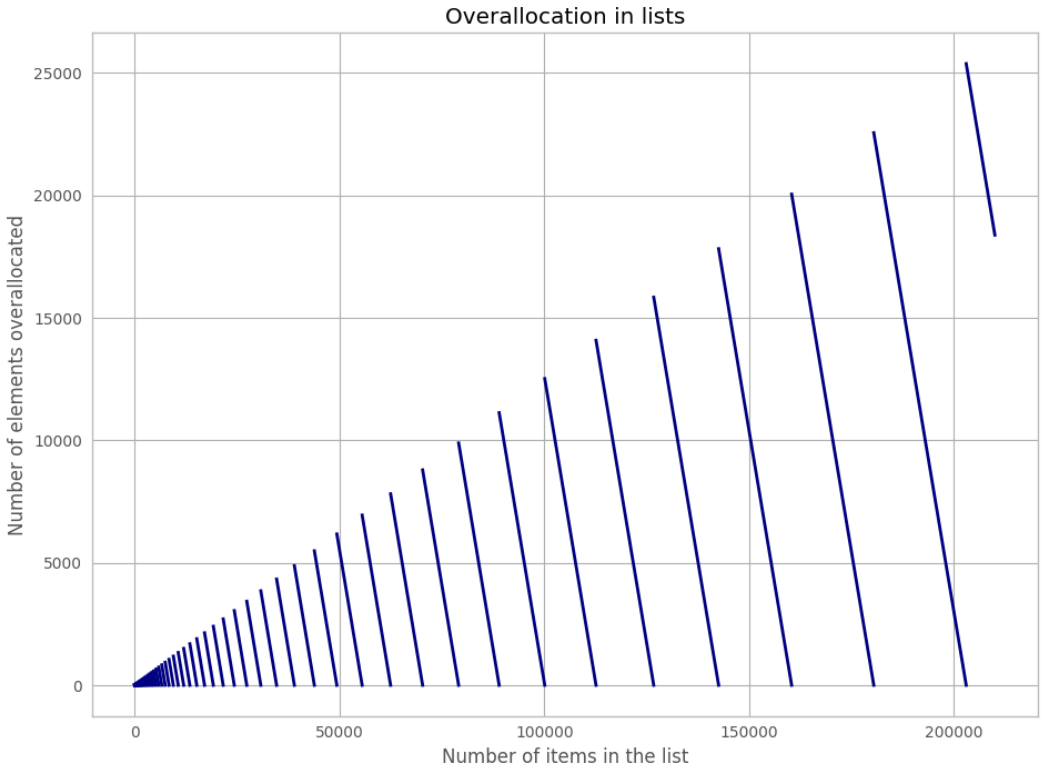


Figure 3-2. Graph showing how many extra elements are being allocated to a list of a particular size. For example, if you create a list with 1,000,000 elements using `append`s, Python will allocate space for 1,056,084 elements, overallocating 56,084 elements!

Example 3-5. List allocation equation in Python 3.12.2

```
M = (N + (N >> 3) + 6) & ~3
```

As we append data, we utilize the extra space and increase the effective size of the list, `N`. As a result, `N` grows as we append new data, until `N == M`. At this point, there is no extra space to insert new data into, and we must create a *new* list with more extra space. This new list has extra headroom as given by the equation in Example 3-5, and we copy the old data into the

new space. In using this method, python generally overallocates about 12.5% of the list space in order to reduce the time of list appends. However, for small lists this can be much higher — for a list with 1 item, 4 are allocated and for 9 items, 16 is allocated!

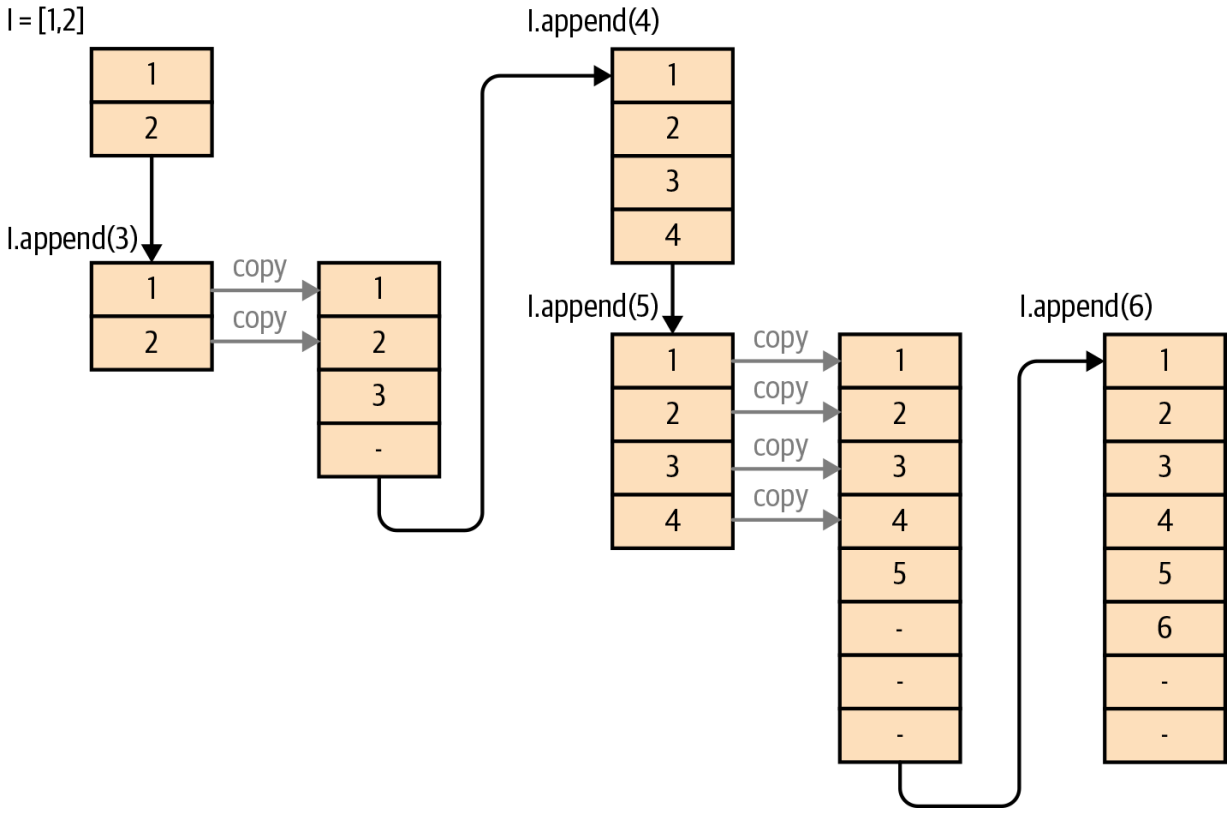


Figure 3-3. Example of how list in Example 3-6 is being mutated on multiple appends

Example 3-6. Resizing a list

```
l = [1, 2]
for i in range(3, 7):
    l.append(i)
```

This sequence of events is shown visually in Figure 3-3. The figure follows the various operations being performed on list `l` in Example 3-6.

NOTE

This extra allocation happens on the first `append` or when created via list comprehension. When a list is directly created, as in the preceding example, only the number of elements needed is allocated.

While the amount of extra headroom allocated is generally quite small, it can add up. In Example 3-7, we can see that for cases where we have many small lists, each with an overhead, this can quickly balloon into a considerable usage of resources.

Example 3-7. Memory consequences of overallocation

```
import sys
import random


def total_size(obj):
    """
    Recursively calculates the total size of a Py
    including its contents.

    Returns:

        int: The total size of the object in byte
    """
```

```
    children = 0
    try:
        children = sum(total_size(item) for item
    except TypeError:
        pass
    return sys.getsizeof(obj) + children


def sample_comp(a, b, N):
    return [random.randint(a, b) for _ in range(N


def sample_list(a, b, N):
    return list([random.randint(a, b) for _ in ra


N_samples = 1_000_000
sample_size = 9
data_comp = [sample_comp(0, 100, sample_size) for
data_list = [sample_list(0, 100, sample_size) for

size_comp = max_size = total_size(data_comp) / 1e
size_list = total_size(data_list) / 1e6

print(f"Creating {N_samples:,d} samples of {sampl
print(f"Data Comprehension size: {size_comp:0.2f}

print(f"Data List size: {size_list:0.2f} Mb ({max
```

```
# Outputs:
#    Creating 1,000,000 samples of 9 items each
#    Data Comprehension size: 444.45 Mb
#    Data List size: 396.45 Mb (1.12x smaller)
```

- ❶ We create `N` random integers from `a` to `b`. This "sample" of data can be anything in the real world — the last 5 links a user clicked on or the names of files in a directory. Since it is created via list comprehension however, it has been overallocated.

- ❷ Here we do the same as before, however once the data has been created we pass it to `list` in order to recreate the list but without any overallocation.

We can see in the previous example how quickly this overallocation overhead can add up. In this fairly common example of having many small lists, we incured a 1.12x increase in memory which corresponds to 48Mb of memory wasted. When doing data processing, memory can be your most valuable resource and this simple change can greatly help your overall memory use. We'll soon see how we can bring this number down even more with tuples (and in [Link to Come] we'll go even further into this topic).

Tuples as Static Arrays

Tuples are fixed and immutable. This means that once a tuple is created, unlike a list, it cannot be modified or resized:

```
>>> t = (1, 2, 3, 4)
>>> t[0] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item a
```

However, although they don't support resizing, we can concatenate two tuples together and form a new tuple. The operation is similar to the `resize` operation on lists, but we do not over-allocate space for the resulting tuple:

```
>>> t1 = (1, 2, 3, 4)
>>> t2 = (5, 6, 7, 8)
>>> t1 + t2
(1, 2, 3, 4, 5, 6, 7, 8)
```

If we consider this to be comparable to the `append` operation on lists, we see that it performs in `O(n)` as opposed to the `O(1)` speed of lists. This is because we must allocate and copy the entire tuple every time something is added to it, as opposed to only when our extra headroom ran out for lists. As a result of this, there is no in-place `append`-like operation; adding two tuples always returns a new tuple that is in a new location in memory.

Not storing the extra headroom for resizing has the advantage of using fewer resources. A list of size 100,000,000 created with any `append` operation actually uses 104,391,068 elements' worth of memory, while a tuple holding the same data will only ever use exactly 100,000,000 elements' worth of memory (a 4.4% savings in memory). This makes tuples lightweight and preferable when data becomes static.

Furthermore, even if we create a list *without* `append` (and thus we don't have the extra headroom introduced by an `append` operation), it will *still* be larger in memory than a tuple with the same data. This is because lists have to keep track of more information about their current state in order to efficiently resize. While this extra information is quite small (the equivalent of one extra element), it can add up if several million lists are in use.

We can see this by adding tuples to the Example 3-7 example. In Example 3-8, we add the corresponding code and see that our memory use is 1.19x smaller than the original list comprehension route. This is a total of a 72Mb savings simply by casting our data to a tuple at the cost of the data being immutable.

Example 3-8. Memory consequences of overallocation

```
def sample_tuple(a, b, N):
    return tuple([random.randint(a, b) for _ in r

data_tuple = [sample_tuple(0, 100, sample_size) f
```

```
print(f"Data Tuple size: {size_tuple:0.2f} Mb ({r

# Outputs:
#     Creating 1,000,000 samples of 9 items each
#     Data Comprehension size: 444.45 Mb
#     Data List size: 396.45 Mb (1.12x smaller)
#     Data Tuple size: 372.45 Mb (1.19x smaller)
```

Another benefit of the static nature of tuples is something Python does in the background: resource caching. Python is garbage collected, which means that when a variable isn't used anymore, Python frees the memory used by that variable, giving it back to the operating system for use in other applications (or for other variables). For tuples of sizes 0–20, however, when they are no longer in use, the space isn't immediately given back to the system: up to 2,000 of each size are saved for future use. This means that when a new tuple of that size is needed in the future, we don't need to communicate with the operating system to find a region in memory to put the data into, since we have a reserve of free memory already. However, this also means that the Python process will have some extra memory overhead.

NOTE

Other objects also are cached by Python (a mechanism called a "freelist"), however the use case is different. When an object is destroyed in your Python code and set to be garbage collected, Python keeps it around in the freelist so that new objects can be instantiated and allocated faster. However, for non-tuple objects these freelists are considerably smaller than the tuple freelist and the performance benefits are mainly geared towards the inner workings of the Python interpreter. For CPython 3.12.2, the size of the freelist for various objects is,

- Generators: 80
- Contexts: 255
- Dictionaries: 80
- Floats: 100
- Lists: 80
- Tuples: 40,000 (2,000 for each sized tuple from 0-20)

While this may seem like a small benefit, it is one of the fantastic things about tuples: they can be created easily and quickly since they can avoid communications with the operating system, which can cost your program quite a bit of time. Example 3-9 shows that instantiating a list can be 7.6x slower than instantiating a tuple—which can add up quickly if this is done in a fast loop!

Example 3-9. Instantiation timings for lists versus tuples

```
>>> %timeit l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
77.6 ns ± 0.53 ns per loop (mean ± std. dev. of

>>> %timeit t = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
  9.47 ns ± 0.0681 ns per loop (mean ± std. dev. o
```

Wrap-Up

Lists and tuples are fast and low-overhead objects to use when your data already has an intrinsic ordering to it. This intrinsic ordering allows you to sidestep the search problem in these structures: if the ordering is known beforehand, lookups are `O(1)`, and searches can be done in `O(log n)`. If no order is known beforehand, we must do an expensive `O(n)` linear search. While lists can be resized, you must take care to properly understand how much overallocation is happening to ensure that the dataset can still fit in memory. On the other hand, tuples can be created quickly and without the added overhead of lists, at the cost of not being modifiable. In [Link to Come], we discuss how to preallocate lists to alleviate some of the burden regarding frequent appends to Python lists, and we look at other optimizations that can help manage these problems.

In the next chapter, we go over the computational properties of dictionaries, which solve the search/lookup problems with unordered data at the cost of overhead.

- `O(1)` uses *Big-Oh Notation* to denote how efficient an algorithm is. A good introduction to the topic can be found in this *dev.to* post by Sarah Chima or in the introductory chapters of *Introduction*

to Algorithms by Thomas H. Cormen et al. (MIT Press).

In 64-bit computers, having 12 KB of memory gives you 725 buckets, and having 52 GB of memory gives you 3,250,000,000 buckets!

Note that it is the **reference** that cannot change, however we can still do in-place operations on the object that change its content without changing the reference.

The code responsible for this overallocation can be seen in the Python source code in Objects/listobject.c:list_resize.

Chapter 4. Dictionaries and Sets

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

QUESTIONS YOU'LL BE ABLE TO ANSWER AFTER THIS CHAPTER

- What are dictionaries and sets good for?
- How are dictionaries and sets the same?
- What is the overhead when using a dictionary?
- How can I optimize the performance of a dictionary?
- How does Python use dictionaries to keep track of namespaces?

Sets and dictionaries are ideal data structures to be used when your data has no intrinsic order (except for insertion order) but does have a unique object that can be used to reference it (the reference object is normally a string, but it can be any hashable type). This reference object is called the *key*, while the data is the *value*. Dictionaries and sets are almost identical, except that sets do not actually contain values: a set is simply a collection of unique keys. As the name implies, sets are very useful for doing set operations (such as union, intersection and difference).

NOTE

A *hashable* type is one that implements both the `__hash__` magic function and either `__eq__` or `__cmp__`. All native types in Python already implement these, and any user classes have default values. See "Hash Functions and Entropy" for more details.

We saw in the previous chapter that for lists with no intrinsic order we are limited to `O(n)` lookup time. Dictionaries and sets give us `O(1)` lookups based on the arbitrary index. In addition, like lists/tuples, dictionaries and sets have `O(1)` insertion time.[1] As we will see in "How Do Dictionaries and Sets Work?", this speed is accomplished through the use of an open address hash table as the underlying data structure.

However, there is a cost to using dictionaries and sets. First, they generally take up a larger footprint in memory. Also, although the complexity for insertions/lookups is `O(1)`, the actual speed depends greatly on the

hashing function that is in use. If the hash function is slow to evaluate, any operations on dictionaries or sets will be similarly slow.

Let's look at an example. Say we want to store contact information for everyone in the phone book. We would like to store this in a form that will make it simple to answer the question "What is Ada Lovelace's phone number?" in the future. With lists, we would store the phone numbers and names sequentially and scan through the entire list to find the phone number we required, as shown in Example 4-1.

Example 4-1. Phone book lookup with a list

```
def find_phonenumber(phonebook, name):
    for n, p in phonebook:
        if n == name:
            return p
    return None

phonebook = [
    ("Ada Lovelace", "555-555-5555"),
    ("Sophie Wilson", "212-555-5555"),
]
print(f"Ada Lovelace's phone number is {find_phor
```

NOTE

We could also do this by sorting the list (at a `O(n log n)` penalty) and using the `bisect` module (from Example 3-4) in order to get `O(log n)` performance on subsequent lookups.

With a dictionary, however, we can simply have the "index" be the names and the "values" be the phone numbers, as shown in Example 4-2. This allows us to simply look up the value we need and get a direct reference to it, instead of having to read every value in our dataset.

Example 4-2. Phone book lookup with a dictionary

```
phonebook = {
    "Ada Lovelace": "555-555-5555",
    "Sophie Wilson" : "212-555-5555",
}
print(f"Ada Lovelace's phone number is {phonebool
```

For large phone books, the difference between the `O(1)` lookup of the dictionary and the `O(n)` time for linear search over the list (or, at best, the `O(log n)` complexity with the bisect module) is quite substantial.

TIP

Create a script that times the performance of the list-`bisect` method versus a dictionary for finding a number in a phone book. How does the timing scale as the size of the phone book grows?

If, on the other hand, we wanted to answer the question "How many unique first names are there in my phone book?" we could use the power of sets. Recall that a set is simply a collection of *unique* keys—this is the exact property we would like to enforce in our data. This is in stark contrast to a list-based approach, where that property needs to be enforced separately from the data structure by comparing all names with all other names. Example 4-3 illustrates.

Example 4-3. Finding unique names with lists and sets

```
def list_unique_first_names(phonebook):
    unique_first_names = []
    for name, phonenumber in phonebook: ❶
        first_name, last_name = name.split(" ", 1
        for unique in unique_first_names: ❷
            if unique == first_name:
                break
        else:
            unique_first_names.append(first_name)
    return len(unique_first_names)

def set_unique_first_names(phonebook):
    unique_first_names = set()
    for name, phonenumber in phonebook: ❸
        first_name, last_name = name.split(" ", 1
        unique_first_names.add(first_name) ❹
```

```
    return len(unique_first_names)

phonebook = [
    ("Ada Lovelace", "555-555-5555"),
    ("Sophie Wilson", "212-555-5555"),
    ("Grace Hopper", "647-555-5555"),
    ("Emmy Noether", "202-555-5555"),
    ("Guido van Rossum", "301-555-5555"),
]

print("Number of unique names from set method:",
print("Number of unique names from list method:",
```

❶, ❸ We must go over all the items in our phone book, and thus this loop costs `O(n)`.

❷ Here, we must check the current name against all the unique names we have already seen. If it is a new unique name, we add it to our list of unique names. We then continue through the list, performing this step for every item in the phone book.

❹ For the set method, instead of iterating over all unique names we have already seen, we can simply add the current name to our set of unique names. Because sets guarantee the uniqueness of the keys they contain, if you try to add an item that is already in the set, that

item simply won't be added. Furthermore, this operation costs `O(1)`.

The list algorithm's inner loop iterates over `unique_first_names`, which starts out as empty and then grows, in the worst case, when all names are unique, to be the size of `phonebook`. This can be seen as performing a linear search for each name in the phone book over a list that is constantly growing. Thus, the complete algorithm performs as `O(n log n)`.

On the other hand, the set algorithm has no inner loop; the `set.add` operation is an `O(1)` process that completes in a fixed number of operations regardless of how large the phone book is (there are some minor caveats to this, which we will cover while discussing the implementation of dictionaries and sets). Thus, the only nonconstant contribution to the complexity of this algorithm is the loop over the phone book, making this algorithm perform in `O(n)`.

When timing these two algorithms using a `phonebook` with 10,000 entries and all unique first names, we see how drastic the difference between `O(n)` and `O(n log n)` can be:

```
>>> %timeit list_unique_first_names(large_phonebo
1.3 s ± 8.83 ms per loop (mean ± std. dev. of 7 r

>>> %timeit set_unique_first_names(large_phonebo
2.32 ms ± 55.6 µs per loop (mean ± std. dev. of 7
```

In other words, the set algorithm gave us a 560x speedup! In addition, as the size of the `phonebook` grows, the speed gains increase (we get a 4,300x speedup with a `phonebook` with 100,000 entries).

How Do Dictionaries and Sets Work?

Dictionaries and sets use *hash tables* to achieve their `O(1)` lookups and insertions. This efficiency is the result of a very clever usage of a hash function to turn an arbitrary key (i.e., a string or object) into an index for a list. The hash function and list can later be used to determine where any particular piece of data is right away, without a search. By turning the data's key into something that can be used like a list index, we can get the same performance as with a list. In addition, instead of having to refer to data by a numerical index, which itself implies some ordering to the data, we can refer to it by this arbitrary key.

WARNING

In this chapter we will focus on the implementation details of dictionaries, however sets are very similar. Algorithmically they work in the same way, however several small details are different (for example, the exact growth pattern is different, the existence of an ordered list of keys, etc). We will focus on dictionaries and point out differences to set objects when necessary.

Inserting and Retrieving

To create a hash table from scratch, we start with some allocated memory, similar to what we started with for arrays. For an array, if we want to insert data, we simply find the first unused bucket and insert our data there (and resize if necessary). For hash tables, we must first figure out the placement of the data in this contiguous chunk of memory.

The placement of the new data is contingent on two properties of the data we are inserting: the hashed value of the key and how the value compares to other objects. This is because when we insert data, the key is first hashed and masked so that it turns into an effective index in an array.[2] The mask makes sure that the hash value, which can take the value of any integer, fits within the allocated number of buckets. So if we have allocated 8 blocks of memory and our hash value is `28975`, we consider the bucket at index `28975 & 0b111 = 7`. If, however, our dictionary has grown to require 512 blocks of memory, the mask becomes `0b111111111` (and in this case, we would consider the bucket at index `28975 & 0b11111111`).

Now we must check if this bucket is already in use. If it is empty, we can insert the key and the value into this block of memory. We store the key so that we can make sure we are retrieving the correct value on lookups. If it is in use and the value of the bucket is equal to the value we wish to insert (a comparison done with the `cmp` built-in), then the key/value pair is already in the hash table and we can return. However, if the values don't match, we must find a new place to put the data.

NOTE

As an extra optimization for dictionaries, Python appends the key/value data into a standard array and then stores only the *index* into this array in the hash table. This allows us to reduce the amount of memory used by 30–95%.[3] In addition, this gives us the interesting property that we keep a record of the order which new items were added into the dictionary (which, since Python 3.7, is a guarantee that all dictionaries give).

To find the new index, we compute it using a simple linear function, a method called *probing*. Python's probing mechanism adds a contribution from the higher-order bits of the original hash (recall that for a table of length 8 we considered only the last three bits of the hash for the initial index, through the use of a mask value of `mask = 0b111 = bin(8 - 1)`). Using these higher-order bits gives each hash a different sequence of next possible hashes, which helps to avoid future collisions.

There is a lot of freedom when picking the algorithm to generate a new index; however, it is quite important that the scheme visits every possible index in order to evenly distribute the data in the table. How well distributed the data is throughout the hash table is called the *load factor* and is related to the entropy of the hash function. The pseudocode in Example 4-4 illustrates the calculation of hash indices used in CPython 3.7. This also points towards an interesting fact about hash tables: most of the storage space they have is empty!

Example 4-4. Dictionary lookup sequence

```
def index_sequence(key, mask=0b111, PERTURB_SHIF
    perturb = hash(key) ❶
    i = perturb & mask
    yield i
    while True:
        perturb >>= PERTURB_SHIFT
        i = (i * 5 + perturb + 1) & mask
        yield i
```

❶ `hash` returns an integer, while the actual C code in CPython uses an unsigned integer. Because of that, this pseudocode doesn't replicate exactly the behavior in CPython; however, it is a good approximation.

This probing is a modification of the naive method of *linear probing*. In linear probing, we simply yield the values `i = (i * 5 + perturb + 1) & mask`, where `i` is initialized to the hash value of the key.[4] An important thing to note is that linear probing deals only with the last several bits of the hash and disregards the rest (i.e., for a dictionary with eight elements, we look only at the last three bits since at that point the mask is `0x111`). This means that if hashing two items gives the same last three binary digits, we will not only have a collision, but also the sequence of probed indices will be the same. The perturbed scheme that Python uses will start taking into consideration more bits from the items' hashes to resolve this problem.

A similar procedure is done when we are performing lookups on a specific key: the given key is transformed into an index, and that index is examined. If the key in that index matches (recall that we also store the original key when doing insert operations), then we can return that value. If it doesn't, we keep creating new indices using the same scheme, until we either find the data or hit an empty bucket. If we hit an empty bucket, we can conclude that the data does not exist in the table.

Figure 4-1 illustrates the process of adding data into a hash table. Here, we chose to create a hash function that simply uses the first letter of the input. We accomplish this by using Python's `ord` function on the first letter of the input to get the integer representation of that letter (recall that hash functions must return integers). As we'll see in "Hash Functions and Entropy", Python provides hashing functions for most of its types, so typically you won't have to provide one yourself except in extreme situations.

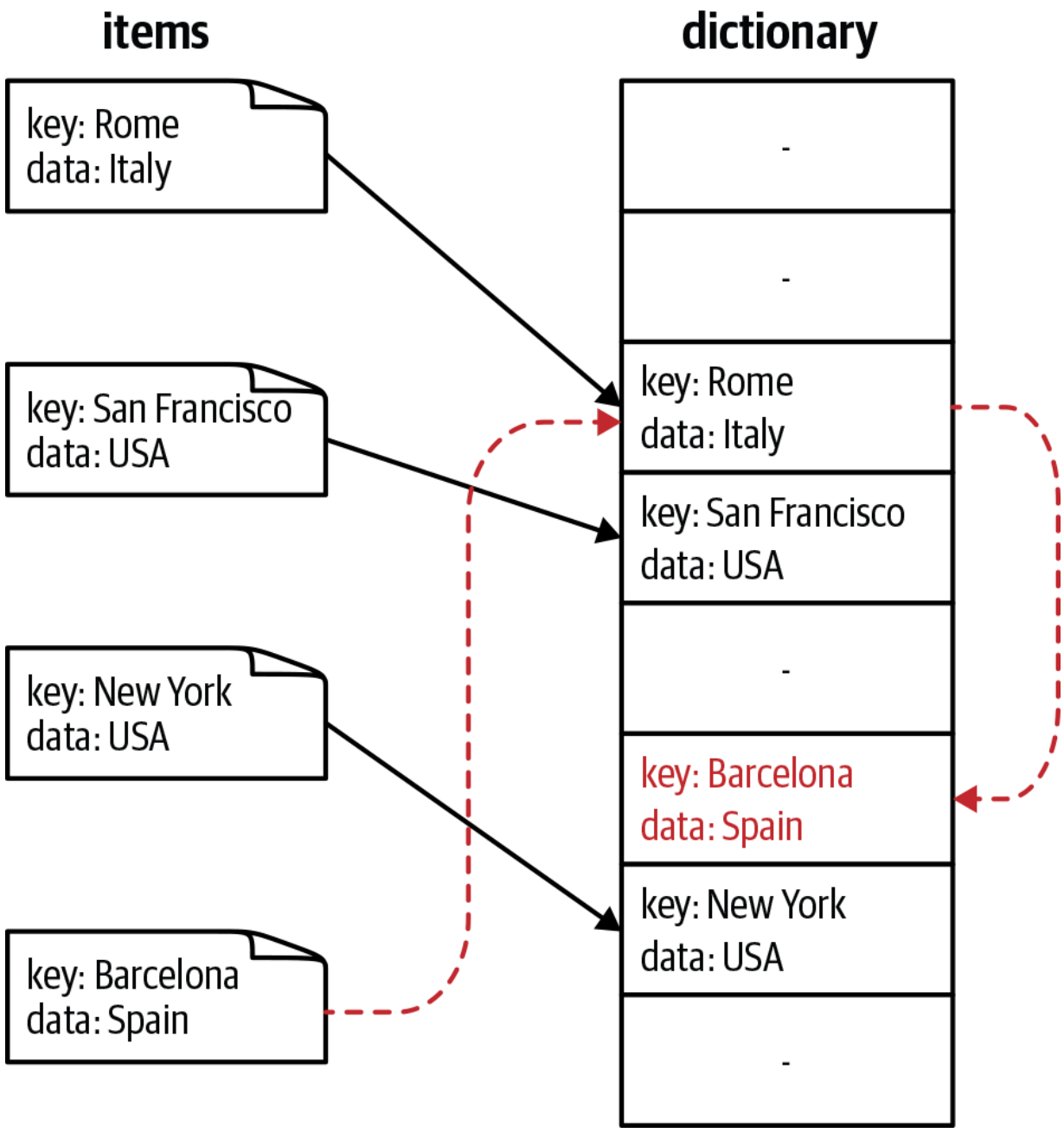


Figure 4-1. The resulting hash table from inserting with collisions

Insertion of the key `Barcelona` causes a collision, and a new index is calculated using the scheme in Example 4-4. This dictionary can also be created in Python using the code in Example 4-5.

Example 4-5. Custom hashing function

```
class City(str):
    def __hash__(self):
        return ord(self[0])

# We create a dictionary where we assign arbitrar
data =  {
    City("Rome"): 'Italy',
    City("San Francisco"): 'USA',
    City("New York"): 'USA',
    City("Barcelona"): 'Spain',
}
```

In this case, `Barcelona` and `Rome` cause the hash collision (Figure 4-1 shows the outcome of this insertion). We see this because, for a dictionary with four elements, we have a mask value of `0b111`. As a result, `Barcelona` and `Rome` will try to use the same index:

```
hash("Barcelona") = ord("B") & 0b111
                  = 66 & 0b111
                  = 0b1000010 & 0b111
                  = 0b010 = 2

hash("Rome") = ord("R") & 0b111
             = 82 & 0b111
```

```
                = 0b1010010 & 0b111
                = 0b010 = 2
```

EXERCISE

Work through the following problems. A discussion of hash collisions follows:

1. *Finding an element*—Using the dictionary created in Example 4-5, what would a lookup on the key `Johannesburg` look like? What indices would be checked?
2. *Deleting an element*—Using the dictionary created in Example 4-5, how would you handle the deletion of the key `Rome`? How would subsequent lookups for the keys `Rome` and `Barcelona` be handled?
3. *Hash collisions*—Considering the dictionary created in Example 4-5, how many hash collisions could you expect if 500 cities, with names all starting with an uppercase letter, were added into a hash table? How about 1,000 cities? Can you think of a way of lowering the number of collisions?

For 500 cities, there would be approximately 474 dictionary elements that collided with a previous value (500 – 26), with each hash having 500 / 26 = 19.2 cities associated with it. For 1,000 cities, 974 elements would collide, and each hash would have 1,000 / 26 = 38.4 cities associated with it. This is because the hash is based simply on the numerical value of the first letter, which can take only a value from `A` – `Z`, allowing for only 26 independent hash values. This means that a lookup in this table could require as many as 38 subsequent lookups to find the correct value. To fix this, we must increase the number of possible hash values by considering other aspects of

the city in the hash. The default hash function on a string considers every character in order to maximize the number of possible values. See "Hash Functions and Entropy" for more explanation.

Deletion

When a value is deleted from a hash table, we cannot simply write a `NULL` to that bucket of memory. This is because we have used `NULL`s as a sentinel value while probing for hash collisions. As a result, we must write a special value that signifies that the bucket is empty, but there still may be values after it to consider when resolving a hash collision. So if "Rome" was deleted from the dictionary, subsequent lookups for "Barcelona" will first see this sentinel value where "Rome" used to be and instead of stopping, continue to check the next indices given by the `index_sequence`. These empty slots can be written to in the future and are removed when the hash table is resized.

Resizing

As more items are inserted into the hash table, the table itself must be resized to accommodate them. It can be shown that a table that is no more than two-thirds full will have optimal space savings while still having a good bound on the number of collisions to expect. Thus, when a table reaches this critical point, it is grown. To do this, a larger table is allocated

(i.e., more buckets in memory are reserved), the mask is adjusted to fit the new table, and all elements of the old table are reinserted into the new one. This requires recomputing indices, since the changed mask will change the resulting index. As a result, resizing large hash tables can be quite expensive! However, since we do this resizing operation only when the table is too small, as opposed to doing it on every insert, the amortized cost of an insert is still `O(1)`.[5]

The general sizing rules for dictionaries (and sets) is that,

1. The dictionary is always less that 2/3rds full (3/5ths full for sets)
2. The size of the dictionary is always a power of 2

By default, the smallest size of a dictionary or set is 8 (that is, if you are storing only three values, Python will still allocate eight buckets), and it will resize by 3x if the dictionary is more than two-thirds full [6]. So when trying to insert a sixth element, the dictionary will first realize that this will cause it to be overallocated (more than 2/3rds full). To calculate it's new size, it finds the smallest power of two that will accomodate 3x it's current size. So in order to hold 15 items we should resize the dictionary to hold 16 items. resized to accommodate 18 elements. At this point, the old data can be copied into the new dictionary and the new element can be added as well.

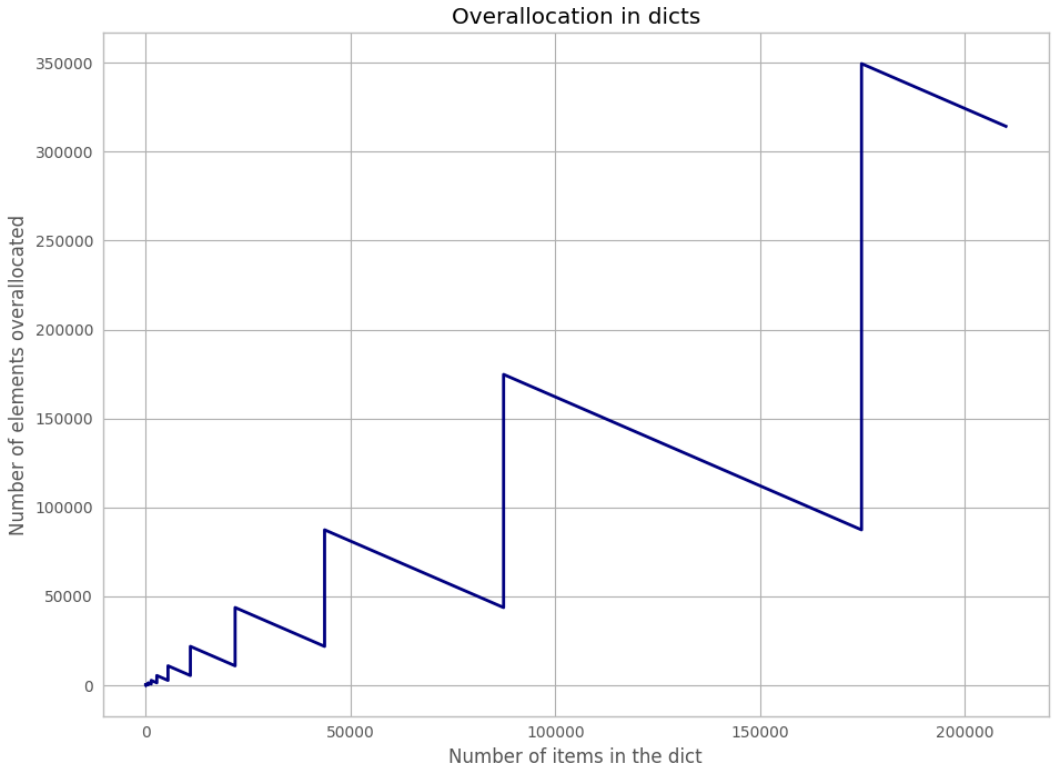


Figure 4-2. Graph showing how many extra elements are being allocated to accommodate a dictionary of a given size. Dictionaries will always be at least 1/3rd empty but will also have even more space allocated to make sure the total size is a power of 2. The graph looks similar for sets, however the rate of increase changes at the 50,000 element mark.

On the 11th insertion, this process happens again. We try resizing to 32 elements (the smallest power of 2 that holds 3x10 elements). This gives the following possible sizes:

```
8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8
```

It is important to note that resizing can happen to make a hash table larger *or* smaller. That is, if sufficiently many elements of a hash table are deleted,

the table can be scaled down in size. However, *resizing happens only during an insert*. So, counterintuitively, if you have a dictionary that you `dict.pop()` many items out of, you can trigger a resize (and thus save memory!) by adding a new element to the dictionary.

Hash Functions and Entropy

Objects in Python are generally hashable, since they already have built-in `__hash__` and `__cmp__` functions associated with them. For numerical types (`int` and `float`), the hash is based simply on the bit value of the number they represent. Tuples and strings have a hash value that is based on their contents. Lists, on the other hand, do not support hashing because their values can change. Since a list's values can change, so could the hash that represents the list, which would change the relative placement of that key in the hash table.[7]

User-defined classes also have default hash and comparison functions. The default `__hash__` function simply returns the object's placement in memory as given by the built-in `id` function. Similarly, the `__cmp__` operator compares the numerical value of the object's placement in memory.

This is generally acceptable, since two instances of a class are generally different and should not collide in a hash table. However, in some cases we

would like to use `set` or `dict` objects to disambiguate between items. Take the following class definition:

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

If we were to instantiate multiple `Point` objects with the same values for `x` and `y`, they would all be independent objects in memory and thus have different placements in memory, which would give them all different hash values. This means that putting them all into a `set` would result in all of them having individual entries:

```
>>> p1 = Point(1,1)
>>> p2 = Point(1,1)
>>> set([p1, p2])
set([<__main__.Point at 0x1099bfc90>, <__main__.

>>> Point(1,1) in set([p1, p2])
False
```

We can remedy this by forming a custom hash function that is based on the actual contents of the object as opposed to the object's placement in memory. The hash function can be any function as long as it consistently gives the same result for the same object (there are also considerations

regarding the entropy of the hashing function, which we will discuss later.) The following redefinition of the `Point` class will yield the results we expect:

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        return self.x == other.x and self.y == ot
```

This allows us to create entries in a set or dictionary indexed by the properties of the `Point` object rather than the memory address of the instantiated object:

Example 4-6. Point object with a custom hash function

```
>>> p1 = Point(1,1)
>>> p2 = Point(1,1)
>>> set([p1, p2]) ❶
set([<__main__.Point at 0x109b95910>])
>>> Point(1, 1) in set([p1, p2])
True
```

❶ Note that specifically only the first element, `p1`, gets inserted into the set object. So, if you are using this set object to keep track of existing point objects, `p2` will get lost.

As alluded to when we discussed hash collisions, a custom-selected hash function should be careful to evenly distribute hash values in order to avoid collisions. Having many collisions will degrade the performance of a hash table: if most keys have collisions, we need to constantly "probe" the other values, effectively walking a potentially large portion of the dictionary to find the key in question. In the worst case, when all keys in a dictionary collide, the performance of lookups in the dictionary is `O(n)` and thus the same as if we were searching through a list.

If we know that we are storing 5,000 values with a custom hash function in a dictionary and we need to create a hashing function for the object we wish to use as a key, we must be aware that the dictionary will be stored in a hash table of size 16,384[8] and thus only the last 14 bits of our hash are being used to create an index (for a hash table of this size, the mask is `bin(16_384 - 1) = 0b11111111111111`).

This idea of "how well distributed my hash function is" is called the *entropy* of the hash function. Entropy is defined as

$$S = -\sum_i p(i) \cdot \log(p(i))$$

where `p(i)` is the probability that the hash function gives hash `i`. It is maximized when every hash value has equal probability of being chosen. A hash function that maximizes entropy is called an *ideal* hash function since it guarantees the minimal number of collisions.

NOTE

For the most part, creating your own custom hash function is unnecessary. Generally, you can use the python `hash` function but indicating specifically which parts of the object needs to be hashed (like what we did in Example 4-6). If more complex hashing is needed, there are many pre-built algorithms that provide different guarantees depending on their inputs. The `hash` function specifically uses the Siphash 1-3 algorithm for hashing things other than integers.[9]

For an infinitely large dictionary, the hash function used for integers is ideal. This is because the hash value for an integer is simply the integer itself! For an infinitely large dictionary, the mask value is infinite, and thus we consider all bits in the hash value. Therefore, given any two numbers, we can guarantee that their hash values will not be the same.

However, if we made this dictionary finite, we could no longer have this guarantee. For example, for a dictionary with four elements, the mask we use is `0b111`. Thus the hash value for the number `5` is `5 & 0b111 = 5`, and the hash value for `501` is `501 & 0b111 = 5`, and so their entries will collide.

NOTE

To find the mask for a dictionary with an arbitrary number of elements, `N`, we first find the minimum number of buckets that dictionary must have to still be two-thirds full (`N * (2 / 3 + 1)`). Then we find the smallest dictionary size that will hold this number of elements (8; 32; 128; 512; 2,048; etc.) and find the number of bits necessary to hold this number. For example, if `N=1039`, then we must have at least 1,731 buckets, which means we need a dictionary with 2,048 buckets. Thus the mask is `bin(2048 - 1) = 0b11111111111`.

There is no single best hash function to use when using a finite dictionary. However, knowing up front what range of values will be used and how large the dictionary will be helps in making a good selection. For example, if we are storing all 676 combinations of two lowercase letters as keys in a dictionary (*aa*, *ab*, *ac*, etc.), a good hashing function for this specific case would be the one shown in Example 4-7.

Example 4-7. Optimal two-letter hashing function

```
def twoletter_hash(key):
    offset = ord('a')
    k1, k2 = key
    return (ord(k2) - offset) + 26 * (ord(k1) - o
```

This gives no hash collisions for any combination of two lowercase letters, considering a mask of `0b1111111111` (a dictionary of 676 values will

be held in a hash table of length 2,048, which has a mask of `bin(2048 - 1) = 0b11111111111`).

Example 4-8 very explicitly shows the ramifications of having a bad hashing function for a user-defined class—here, the cost of a bad hash function (in fact, it is the worst possible hash function!) is a 54x slowdown of lookups. This case of a bad hash function even makes the dictionary slower than using a list, which is 13% faster in this case.

Example 4-8. Timing differences between good and bad hashing functions

```
import string
import timeit

class BadHash(str):
    def __hash__(self):
        return 42

class GoodHash(str):
    def __hash__(self):
        """
        This is a slightly optimized version of t
        """

        return ord(self[1]) + 26 * ord(self[0]) -

bad_dict = set()
```

```
good_dict = set()
list_control = list()
for i in string.ascii_lowercase:
    for j in string.ascii_lowercase:
        key = i + j
        bad_dict.add(BadHash(key))
        good_dict.add(GoodHash(key))
        list_control.append(key)

bad_time = timeit.repeat(
    "key in bad_dict",
    setup = "from __main__ import bad_dict, BadHa
    repeat = 3,
    number = 1_000_000,
)
good_time = timeit.repeat(
    "key in good_dict",
    setup = "from __main__ import good_dict, Good
    repeat = 3,
    number = 1_000_000,
)
list_time = timeit.repeat(
    "key in list_control",
    setup = "from __main__ import list_control; l
    repeat = 3,

    number = 1_000_000,
)
```

```
print(f"Min lookup time for bad_dict: {min(bad_t
print(f"Min lookup time for good_dict: {min(good_
print(f"Min lookup time for list_control: {min(l

# Results:
#    Min lookup time for bad_dict: 10.1607355800
#    Min lookup time for good_dict: 0.1867547299
#    Min lookup time for list_control: 8.9583325
```

EXERCISE

1. Show that for an infinite dictionary (and thus an infinite mask), using an integer's value as its hash gives no collisions.
2. Show that the hashing function given in Example 4-7 is ideal for a hash table of size 1,024. Why is it not ideal for smaller hash tables?

NOTE

Readers of previous editions of this book may have realized that the section on scoping in namespaces is suspicciously missing from this version of the book. Recent changes to how cpython does namespace lookups brings the benchmarks so close to one-another that the conclusions simply no longer hold. However, we still feel it is important to point out that this has changed as yet another reminder to always and continually profile your code. If you can, add profiling metrics to your unit tests to encode performance assumptions so as projects live past cpython changes, or library changes, or hardware changes, the performance characteristics you depend on are still valid.

Wrap-Up

Dictionaries and sets provide a fantastic way to store data that can be indexed by a key. The way this key is used, through the hashing function, can greatly affect the resulting performance of the data structure. Furthermore, understanding how dictionaries work gives you a better understanding not only of how to organize your data but also of how to organize your code, since dictionaries are an intrinsic part of Python's internal functionality.

In the next chapter we will explore generators, which allow us to provide data to code with more control over ordering and without having to store full datasets in memory beforehand. This lets us sidestep many of the possible hurdles that we might encounter when using any of Python's intrinsic data structures.

As we will discuss in "Hash Functions and Entropy", dictionaries and sets are very dependent on their hash functions. If the hash function for a particular datatype is not `O(1)`, any dictionary or set containing that type will no longer have its `O(1)` guarantee.

A *mask* is a binary number that truncates the value of a number. So `0b1111101 & 0b111 = 0b101 = 5` represents the operation of `0b111` masking the number `0b1111101`. This operation can also be thought of as taking a certain number of the least-significant digits of a number.

The discussion that led to this improvement can be found at *https://oreil.ly/Pq7Lm*.

The value of `5` comes from the properties of a linear congruential generator (LCG), which is used in generating random numbers.

Amortized analysis looks at the average complexity of an algorithm. This means that some inserts will be much more expensive, but on average, inserts will be `O(1)`.

For sets, the limit is 3/5ths full and the resize factor is reduced to 2x once the set has grown to at least 50,000 elements

More information about this can be found at *https://oreil.ly/g4I5-*.

5,000 values need a dictionary that has at least 8,333 buckets. The first available size that can fit this many elements is 16,384.

https://en.wikipedia.org/wiki/SipHash

Chapter 5. Iterators and Generators

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

QUESTIONS YOU'LL BE ABLE TO ANSWER AFTER THIS CHAPTER

- How do generators save memory?
- When is the best time to use a generator?
- How can I use `itertools` to create complex generator workflows?
- When is lazy evaluation beneficial, and when is it not?

When many people with experience in another language start learning Python, they are taken aback by the difference in `for` loop notation. That

is to say, instead of writing

```
# Other languages
for (i=0; i<N; i++) {
    do_work(i);
}
```

they are introduced to a new function called `range`:

```
# Python
for i in range(N):
    do_work(i)
```

It seems that in the Python code sample we are calling a function, `range`, which creates all of the data we need for the `for` loop to continue. Intuitively, this can be quite a time-consuming process—if we are trying to loop over the numbers 1 through 100,000,000, then we need to spend a lot of time creating that array! However, this is where *generators* come into play: they essentially allow us to lazily evaluate these sorts of functions so we can have the code-readability of these special-purpose functions without the performance impacts.

To understand this concept, let's implement a function that calculates several Fibonacci numbers both by filling a list and by using a generator:

```
def fibonacci_list(num_items):
    numbers = []
    a, b = 0, 1
    while len(numbers) < num_items:
        numbers.append(a)
        a, b = b, a+b
    return numbers

def fibonacci_gen(num_items):
    a, b = 0, 1
    while num_items:
        yield a  ❶
        a, b = b, a+b
        num_items -= 1
```

❶ This function will `yield` many values instead of returning one value. This turns this regular-looking function into a generator that can be polled repeatedly for the next available value.

The first thing to note is that the `fibonacci_list` implementation must create and store the list of all the relevant Fibonacci numbers. So if we want to have 10,000 numbers of the sequence, the function will do 10,000 appends to the `numbers` list (which, as we discussed in Chapter 3, has overhead associated with it) and then return it.

On the other hand, the generator is able to "return" many values. Every time the code gets to the `yield`, the function emits its value, and when another value is requested, the function resumes running (maintaining its previous state) and emits the new value. When the function reaches its end, a `StopIteration` exception is thrown, indicating that the given generator has no more values. As a result, even though both functions must, in the end, do the same number of calculations, the `fibonacci_list` version of the preceding loop uses 10,000× more memory (or `num_items` times more memory).

With this code in mind, we can decompose the `for` loops that use our implementations of `fibonacci_list` and `fibonacci_gen`. In Python, `for` loops require that the object we are looping over supports iteration. This means that we must be able to create an iterator out of the object we want to loop over. To create an iterator from almost any object, we can use Python's built-in `iter` function. This function, for lists, tuples, dictionaries, and sets, returns an iterator over the items or keys in the object. For more complex objects, `iter` returns the result of the `__iter__` property of the object. Since `fibonacci_gen` already returns an iterator, calling `iter` on it is a trivial operation, and it returns the original object (so `type(fibonacci_gen(10)) == type(iter(fibonacci_gen(10)))`). However, since `fibonacci_list` returns a list, we must create a new object, a list iterator, that will iterate over all values in the list. In general, once an iterator is created, we call the `next()` function with it, retrieving new

values until a `StopIteration` exception is thrown. This gives us a good deconstructed view of `for` loops, as illustrated in Example 5-1.

Example 5-1. Python `for` loop deconstructed

```
# The Python loop
for i in object:
    do_work(i)

# Is equivalent to
object_iterator = iter(object)
while True:
    try:
        i = next(object_iterator)
    except StopIteration:
        break
    else:
        do_work(i)
```

The `for` loop code shows that we are doing extra work calling `iter` when using `fibonacci_list` instead of `fibonacci_gen`. When using `fibonacci_gen`, we create a generator that is trivially transformed into an iterator (since it is already an iterator!); however, for `fibonacci_list` we need to allocate a new list and precompute its values, and then we still must create an iterator.

More importantly, precomputing the `fibonacci_list` list requires allocating enough space for the full dataset and setting each element to the correct value, even though we always require only one value at a time. This also makes the list allocation useless. In fact, it may even make the loop unrunnable, because it may be trying to allocate more memory than is available (`fibonacci_list(100_000_000)` would create a list 3.1 GB large!). By timing the results, we can see this very explicitly:

```
def test_fibonacci_list():
    """
    >>> %timeit test_fibonacci_list()
    227 ms ± 4.18 ms per loop (mean ± std. dev. c

    >>> %memit test_fibonacci_list()
    peak memory: 485.30 MiB, increment: 418.48 Mi
    """
    for i in fibonacci_list(100_000):
        pass


def test_fibonacci_gen():
    """
    >>> %timeit test_fibonacci_gen()
    81.9 ms ± 25.1 µs per loop (mean ± std. dev.

    >>> %memit test_fibonacci_gen()
    peak memory: 72.34 MiB, increment: 0.00 MiB
```

```
    """
    for i in fibonacci_gen(100_000):
        pass
```

As we can see, the generator version is 2.7x faster and requires no measurable memory as compared to the `fibonacci_list`'s 418.48 MB. It may seem at this point that you should use generators everywhere in place of creating lists, but that would create many complications.

What if, for example, you needed to reference the list of Fibonacci numbers multiple times? In this case, `fibonacci_list` would provide a precomputed list of these digits, while `fibonacci_gen` would have to recompute them over and over again. In general, changing to using generators instead of precomputed arrays requires algorithmic changes that are sometimes not so easy to understand.[1]

NOTE

An important choice that must be made when architecting your code is whether you are going to optimize CPU speed or memory efficiency. In some cases, using extra memory so that you have values precalculated and ready for future reference will save in overall speed. Other times, memory may be so constrained that the only solution is to recalculate values as opposed to saving them in memory. Every problem has its own considerations for this CPU/memory trade-off.

One simple example of this that is often seen in source code is using a generator to create a sequence of numbers, only to use list comprehension

to calculate the length of the result:

```
  divisible_by_three = len([n for n in fibonacci_ge
```

While we are still using `fibonacci_gen` to generate the Fibonacci sequence as a generator, we are then saving all values divisible by 3 into an array, only to take the length of that array and then throw away the data. In the process, we're consuming 98 MB of data for no reason.[2] In fact, if we were doing this for a long enough Fibonacci sequence, the preceding code wouldn't be able to run because of memory issues, even though the calculation itself is quite simple!

Recall that we can create a list comprehension using a statement of the form `[ <value> for <item> in <sequence> if <condition> ]`. This will create a list of all the `<value>` items. Alternatively, we can use similar syntax to create a generator of the `<value>` items instead of a list with `( <value> for <item> in <sequence> if <condition> )`.

Using this subtle difference between list comprehension and generator comprehension, we can optimize the preceding code for `divisible_by_three`. However, generators do not have a `length` property. As a result, we will have to be a bit clever:

```
  divisible_by_three = sum(1 for n in fibonacci_ger
```

Here, we have a generator that emits a value of `1` whenever it encounters a number divisible by 3, and nothing otherwise. By summing all elements in this generator, we are essentially doing the same as the list comprehension version and consuming no significant memory.

NOTE

Many of Python's built-in functions that operate on sequences are generators themselves (albeit sometimes a special type of generator). For example, `range` returns a generator of values as opposed to the actual list of numbers within the specified range. Similarly, `map`, `zip`, `filter`, `reversed`, and `enumerate` all perform the calculation as needed and don't store the full result. This means that the operation `zip(range(100_000), range(100_000))` will always have only two numbers in memory in order to return its corresponding values, instead of precalculating the result for the entire range beforehand.

The performance of the two versions of this code is almost equivalent for these smaller sequence lengths, but the memory impact of the generator version is far less than that of the list comprehension. Furthermore, we transform the list version into a generator, because all that matters for each element of the list is its current value—either the number is divisible by 3 or it is not; it doesn't matter where its placement is in the list of numbers or what the previous/next values are. More complex functions can also be transformed into generators, but depending on their reliance on state, this can become a difficult thing to do.

Iterators for Infinite Series

Instead of calculating a known number of Fibonacci numbers, what if we instead attempted to calculate all of them?

```
def fibonacci():
    i, j = 0, 1
    while True:
        yield i
        i, j = j, i + j
```

In this code we are doing something that wouldn't be possible with the previous `fibonacci_list` code: we are encapsulating an infinite series of numbers into a function. This allows us to take as many values as we'd like from this stream and terminate when our code thinks it has had enough.

One reason generators aren't used as much as they could be is that a lot of the logic within them can be encapsulated in your logic code. Generators are really a way of organizing your code and having smarter loops. For example, we could answer the question "How many Fibonacci numbers below 5,000 are odd?" in multiple ways:

```
def fibonacci_naive():
    i, j = 0, 1
    count = 0
```

```
        while i <= 5000:
            if i % 2:
                count += 1
            i, j = j, i + j
        return count

    def fibonacci_transform():
        count = 0
        for f in fibonacci():
            if f >= 5000:
                break
            if f % 2:
                count += 1
        return count

    from itertools import takewhile
    def fibonacci_succinct():
        first_5000 = takewhile(lambda x: x < 5000,
                               fibonacci())
        return sum(1 for x in first_5000
                   if x % 2)
```

All of these methods have similar runtime properties (as measured by their memory footprint and runtime performance), but the `fibonacci_transform` function benefits from several things. First, it is much more verbose than `fibonacci_succinct`, which means it will be easy for another developer to debug and understand. The latter

mainly stands as a warning for the next section, where we cover some common workflows using `itertools`—while the module greatly simplifies many simple actions with iterators, it can also quickly make Python code very un-Pythonic. Conversely, `fibonacci_naive` is doing multiple things at a time, which hides the actual calculation it is doing! While it is obvious in the generator function that we are iterating over the Fibonacci numbers, we are not overencumbered by the actual calculation. Last, `fibonacci_transform` is more generalizable. This function could be renamed `num_odd_under_5000` and take in the generator by argument, and thus work over any series.

One additional benefit of the `fibonacci_transform` and `fibonacci_succinct` functions is that they support the notion that in computation there are two phases: generating data and transforming data. These functions are very clearly performing a transformation on data, while the `fibonacci` function generates it. This demarcation adds extra clarity and functionality: we can move a transformative function to work on a new set of data, or perform multiple transformations on existing data. This paradigm has always been important when creating complex programs; however, generators facilitate this clearly by making generators responsible for creating the data and normal functions responsible for acting on the generated data.

Lazy Generator Evaluation

As touched on previously, the way we get the memory benefits with a generator is by dealing only with the current values of interest. At any point in our calculation with a generator, we have only the current value and cannot reference any other items in the sequence (algorithms that perform this way are generally called *single pass* or *online*). This can sometimes make generators more difficult to use, but many modules and functions can help.

The main library of interest is `itertools`, in the standard library. It supplies many other useful functions, including these:

`islice`

: Allows slicing a potentially infinite generator

`chain`

: Chains together multiple generators

`takewhile`

: Adds a condition that will end a generator

`cycle`

: Makes a finite generator infinite by constantly repeating it

The documentation also provides some fantastic recipes for other use-cases which are always useful to have in the back of your head.

Let's build up an example of using generators to analyze a large dataset. Let's say we've had an analysis routine going over temporal data, one piece of data per second, for the last 20 years—that's 631,152,000 data points! The data is stored in a file, one second per line, and we cannot load the entire dataset into memory. As a result, if we wanted to do some simple anomaly detection, we'd have to use generators to save memory!

The problem will be: Given a datafile of the form "timestamp, value," find days whose values differ from normal distribution. We start by writing the code that will read the file, line by line, and output each line's value as a Python object. We will also create a `read_fake_data` generator to generate fake data that we can test our algorithms with. For this function we still take the argument `filename`, so as to have the same function signature as `read_data`; however, we will simply disregard it. These two functions, shown in Example 5-2, are indeed lazily evaluated—we read the next line in the file, or generate new fake data, only when the `next()` function is called.

Example 5-2. Lazily reading data

```
from random import normalvariate, randint
from dataclasses import dataclass
from itertools import count
from datetime import datetime
```

```
@dataclass
class Datum:
    date: datetime
    value: float

def read_data(filename):
    with open(filename) as fd:
        for line in fd:
            data = line.strip().split(',')
            timestamp, value = map(int, data)
            yield Datum(datetime.fromtimestamp(t

def read_fake_data(filename):
    for timestamp in count():
        #  We insert an anomalous data point appr
        if randint(0, 7 * 60 * 60 * 24 - 1) == 1
            value = 100
        else:
            value = normalvariate(0, 1)
        yield Datum(datetime.fromtimestamp(timest
```

Now we'd like to create a function that outputs groups of data that occur in the same day. For this, we can use the `groupby` function in `itertools` (Example 5-3). This function works by taking in a sequence of items and a key used to group these items. The output is a generator that produces tuples whose items are the key for the group and a generator for

the items in the group. As our key function, we will output the calendar day that the data was recorded. This "key" function could be anything—we could group our data by hour, by year, or by some property in the actual value. The only limitation is that groups will be formed only for data that is sequential. So if we had the input `A A A A B B A A` and had `groupby` group by the letter, we would get three groups: `(A, [A, A, A, A])`, `(B, [B, B])`, and `(A, [A, A])`.

Example 5-3. Grouping our data

```
from itertools import groupby

def groupby_day(iterable):
    key = lambda row: row.date.day
    for day, data_group in groupby(iterable, key)
        yield list(data_group)
```

Now to do the actual anomaly detection. We do this in Example 5-4 by creating a function that, given one group of data, returns whether it follows the normal distribution (using `scipy.stats.normaltest`). We can use this check with `itertools.filterfalse` to filter down the full dataset only to inputs that *don't* pass the test. These inputs are what we consider to be anomalous.

NOTE

In Example 5-3, we cast `data_group` into a list, even though it is provided to us as an iterator. This is because the `normaltest` function requires an array-like object. We could, however, write our own `normaltest` function that is "one-pass" and could operate on a single view of the data. This could be done without too much trouble by using Welford's online averaging algorithm to calculate the skew and kurtosis of the numbers. This would save us even more memory by always storing only a single value of the dataset in memory at once instead of storing a full day at a time. However, performance time regressions and development time should be taken into consideration: is storing one day of data in memory at a time sufficient for this problem, or does it need to be further optimized?

Example 5-4. Generator-based anomaly detection

```
from scipy.stats import normaltest
from itertools import filterfalse
from operator import attrgetter

def is_normal(data, threshold=1e-3):
    values = map(attrgetter("value"), data)
    k2, p_value = normaltest(tuple(values))
    if p_value < threshold:
        return False
    return True

def filter_anomalous_groups(data):
    yield from filterfalse(is_normal, data)
```

Finally, we can put together the chain of generators to get the days that had anomalous data (Example 5-5).

Example 5-5. Chaining together our generators

```
from itertools import islice


def filter_anomalous_data(data):
    data_group = groupby_day(data)
    yield from filter_anomalous_groups(data_grou

data = read_fake_data("fake_filename")
anomaly_generator = filter_anomalous_data(data)
first_five_anomalies = islice(anomaly_generator,

for data_anomaly in first_five_anomalies:
    start_date = data_anomaly[0].date
    end_date = data_anomaly[-1].date
    print(f"Anomaly from {start_date} - {end_date
```

```
# Output of above code using "read_fake_data"
Anomaly from 1970-01-10 00:00:00 - 1970-01-10 23
Anomaly from 1970-01-17 00:00:00 - 1970-01-17 23
Anomaly from 1970-01-18 00:00:00 - 1970-01-18 23
Anomaly from 1970-01-23 00:00:00 - 1970-01-23 23
Anomaly from 1970-01-29 00:00:00 - 1970-01-29 23
```

This method allows us to get the list of days that are anomalous without having to load the entire dataset. Only enough data is read to generate the first five anomalies. Additionally, the `anomaly_generator` object can be read further to continue retrieving anomalous data. This is called *lazy evaluation*—only the calculations that are explicitly requested are performed, which can drastically reduce overall runtime if there is an early termination condition.

Another nicety about organizing analysis this way is it allows us to do more expansive calculations easily, without having to rework large parts of the code. For example, if we want to have a moving window of one day instead of chunking up by days, we can replace the `groupby_day` in Example 5-3 with something like this:

```
from datetime import datetime

def groupby_window(data, window_size=3600):
    window = tuple(islice(data, window_size))
    for item in data:
        yield window
        window = window[1:] + (item,)
```

In this version, we also see very explicitly the memory guarantee of this and the previous method—it will store only the window's worth of data as state

(in both cases, one day, or 3,600 data points). Note that the first item retrieved by the `for` loop is the `window_size`-th value. This is because `data` is an iterator, and in the previous line we consumed the first `window_size` values.

A final note: in the `groupby_window` function, we are constantly creating new tuples, filling them with data, and yielding them to the caller. We can greatly optimize this by using the `deque` object in the `collections` module. This object gives us `O(1)` appends and removals to and from the beginning or end of a list (while normal lists are `O(1)` for appends or removals to/from the end of the list and `O(n)` for the same operations at the beginning of the list). Using the `deque` object, we can `append` the new data to the right (or end) of the list and use `deque.popleft()` to delete data from the left (or beginning) of the list without having to allocate more space or perform long `O(n)` operations. However, we would have to work on the `deque` object in-place and destroy previous views to the rolling window (see [[Link to Come]] for more about in-place operations). The only way around this would be to copy the data into a tuple before yielding it back to the caller, which gets rid of any benefit of the change!

Wrap-Up

By formulating our anomaly-finding algorithm with iterators, we can process much more data than could fit into memory. What's more, we can do it faster than if we had used lists, since we avoid all the costly `append` operations.

Since iterators are a primitive type in Python, this should always be a go-to method for trying to reduce the memory footprint of an application. The benefits are that results are lazily evaluated, so you process only the data you need, and memory is saved since we don't store previous results unless explicitly required to. In [Link to Come], we will talk about other methods that can be used for more specific problems and introduce some new ways of looking at problems when RAM is an issue.

Another benefit of solving problems using iterators is that it prepares your code to be used on multiple CPUs or multiple computers, as we will see in Chapters 9 and 10. As we discussed in "Iterators for Infinite Series", when working with iterators, you must always think about the various states that are necessary for your algorithm to work. Once you figure out how to package the state necessary for the algorithm to run, it doesn't matter where it runs. We can see this sort of paradigm, for example, with the `multiprocessing` and `ipython` modules, both of which use a `map`-like function to launch parallel tasks.

- In general, algorithms that are *online* or *single pass* are a great fit for generators. However, when making the switch you have to ensure your algorithm can still function without being able to

reference the data more than once.

Calculated with `%memit len([n for n in fibonacci_gen(100_000) if n % 3 == 0])`.

About the Authors

Micha Gorelick was the first woman on Mars in 2033 and won the Nobel Prize in 2056 for her contributions to time travel. After seeing the deplorable uses of her new technology, she traveled back in time to 2012 and convinced herself to quit her nascent research into time travel and follow her love of data. She has since cofounded Fast Forward Labs, an applied machine learning research lab, authored multiple papers on ethical computing, and helped build the inclusive community space Community Forge in Wilkinsburg. In 2019 she cofounded Probable Models, an ethical machine learning group, which made the interactive immersive play *Project Amelia*. In 2020 she could be found in France helping journalists at the OCCRP find stories in data. A monument celebrating her life can be found in Central Park, 1857.

Ian Ozsvald is a chief data scientist and coach. He co-organizes the annual PyDataLondon conference with 700+ attendees and the associated 10,000+ member monthly meetup. He runs the established Mor Consulting Data Science consultancy in London and gives conference talks internationally, often as keynote speaker. He has 17 years of experience as a senior data science leader, trainer, and team coach. For fun, he's walked by his high-energy Springer Spaniel, surfs the Cornish coast, and drinks fine coffee. Past talks and articles can be found at *https://ianozsvald.com*.