

O'REILLY®

Second
Edition

High Performance Python

Practical Performant
Programming for Humans



Micha Gorelick & Ian Ozsváld

High Performance Python

Your Python code may run correctly, but you need it to run faster. Updated for Python 3, this expanded edition shows you how to locate performance bottlenecks and significantly speed up your code in high-data-volume programs. By exploring the fundamental theory behind design choices, *High Performance Python* helps you gain a deeper understanding of Python's implementation.

How do you take advantage of multicore architectures or clusters? Or build a system that scales up and down without losing reliability? Experienced Python programmers will learn concrete solutions to many issues, along with war stories from companies that use high-performance Python for social media analytics, productionized machine learning, and more.

- Get a better grasp of NumPy, Cython, and profilers
- Learn how Python abstracts the underlying computer architecture
- Use profiling to find bottlenecks in CPU time and memory usage
- Write efficient programs by choosing appropriate data structures
- Speed up matrix and vector computations
- Use tools to compile Python down to machine code
- Manage multiple I/O and computational operations concurrently
- Convert multiprocessing code to run on local or remote clusters
- Deploy code faster using tools like Docker

"This is one of a rare class of programming books that will change the way you think about the practice of programming."

—Hilary Mason
Data Scientist in Residence at Accel

Micha Gorelick cofounded Fast Forward Labs, where he served as resident mad scientist. He currently works on ethical and practical questions around researching and deploying machine learning.

Ian Ozsvárd is a chief data scientist and team coach. He's co-organizer of the annual PyData London conference, runs the established Mor Consulting data science consultancy in London, and gives conference talks internationally.

PYTHON / PERFORMANCE

US \$59.99 CAN \$79.99

ISBN: 978-1-492-05502-0



Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

High Performance Python

Practical Performant Programming for Humans

Micha Gorelick and Ian Ozsvárd

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

High Performance Python

by Micha Gorelick and Ian Ozsvald

Copyright © 2020 Micha Gorelick and Ian Ozsvald. All rights reserved.

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.

Acquisitions Editor: Tyler Ortman

Indexer: Potomac Indexing, LLC

Development Editor: Sarah Grey

Interior Designer: David Futato

Production Editor: Christopher Faucher

Cover Designer: Karen Montgomery

Copyeditor: Arthur Johnson

Illustrator: Rebecca Demarest

Proofreader: Sharon Wilkey

September 2014: First Edition

May 2020: Second Edition

Revision History for the Second Edition

2020-04-30: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492055020> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *High Performance Python*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

High Performance Python is available under the [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 International License](#).

978-1-492-05502-0

[LSI]

Table of Contents

Foreword.....	xi
Preface.....	xiii
1. Understanding Performant Python.....	1
The Fundamental Computer System	2
Computing Units	2
Memory Units	5
Communications Layers	8
Putting the Fundamental Elements Together	10
Idealized Computing Versus the Python Virtual Machine	10
So Why Use Python?	14
How to Be a Highly Performant Programmer	16
Good Working Practices	17
Some Thoughts on Good Notebook Practice	19
Getting the Joy Back into Your Work	20
2. Profiling to Find Bottlenecks.....	21
Profiling Efficiently	22
Introducing the Julia Set	23
Calculating the Full Julia Set	26
Simple Approaches to Timing—print and a Decorator	30
Simple Timing Using the Unix time Command	33
Using the cProfile Module	35
Visualizing cProfile Output with SnakeViz	39
Using line_profiler for Line-by-Line Measurements	40
Using memory_profiler to Diagnose Memory Usage	46
Introspecting an Existing Process with PySpy	54

Bytecode: Under the Hood	55
Using the <code>dis</code> Module to Examine CPython Bytecode	55
Different Approaches, Different Complexity	57
Unit Testing During Optimization to Maintain Correctness	59
No-op <code>@profile</code> Decorator	60
Strategies to Profile Your Code Successfully	62
Wrap-Up	64
3. Lists and Tuples.....	65
A More Efficient Search	68
Lists Versus Tuples	71
Lists as Dynamic Arrays	72
Tuples as Static Arrays	76
Wrap-Up	77
4. Dictionaries and Sets.....	79
How Do Dictionaries and Sets Work?	83
Inserting and Retrieving	83
Deletion	87
Resizing	87
Hash Functions and Entropy	88
Dictionaries and Namespaces	92
Wrap-Up	95
5. Iterators and Generators.....	97
Iterators for Infinite Series	101
Lazy Generator Evaluation	103
Wrap-Up	107
6. Matrix and Vector Computation.....	109
Introduction to the Problem	110
Aren't Python Lists Good Enough?	115
Problems with Allocating Too Much	117
Memory Fragmentation	120
Understanding <code>perf</code>	122
Making Decisions with <code>perf</code> 's Output	125
Enter <code>numpy</code>	126
Applying <code>numpy</code> to the Diffusion Problem	129
Memory Allocations and In-Place Operations	133
Selective Optimizations: Finding What Needs to Be Fixed	137
<code>numexpr</code> : Making In-Place Operations Faster and Easier	140
A Cautionary Tale: Verify "Optimizations" (<code>scipy</code>)	142

Lessons from Matrix Optimizations	143
Pandas	146
Pandas's Internal Model	146
Applying a Function to Many Rows of Data	148
Building DataFrames and Series from Partial Results Rather than Concatenating	156
There's More Than One (and Possibly a Faster) Way to Do a Job	157
Advice for Effective Pandas Development	159
Wrap-Up	160
7. Compiling to C.....	161
What Sort of Speed Gains Are Possible?	162
JIT Versus AOT Compilers	164
Why Does Type Information Help the Code Run Faster?	164
Using a C Compiler	165
Reviewing the Julia Set Example	166
Cython	167
Compiling a Pure Python Version Using Cython	167
pyximport	169
Cython Annotations to Analyze a Block of Code	170
Adding Some Type Annotations	172
Cython and numpy	176
Parallelizing the Solution with OpenMP on One Machine	178
Numba	180
Numba to Compile NumPy for Pandas	182
PyPy	183
Garbage Collection Differences	184
Running PyPy and Installing Modules	185
A Summary of Speed Improvements	186
When to Use Each Technology	187
Other Upcoming Projects	188
Graphics Processing Units (GPUs)	189
Dynamic Graphs: PyTorch	190
Basic GPU Profiling	193
Performance Considerations of GPUs	194
When to Use GPUs	196
Foreign Function Interfaces	197
ctypes	199
cffi	201
f2py	204
CPython Module	207
Wrap-Up	211

8. Asynchronous I/O.....	213
Introduction to Asynchronous Programming	215
How Does <code>async/await</code> Work?	218
Serial Crawler	219
Gevent	221
tornado	226
aiohttp	229
Shared CPU-I/O Workload	233
Serial	233
Batched Results	235
Full Async	238
Wrap-Up	243
9. The multiprocessing Module.....	245
An Overview of the multiprocessing Module	248
Estimating Pi Using the Monte Carlo Method	250
Estimating Pi Using Processes and Threads	251
Using Python Objects	252
Replacing multiprocessing with Joblib	260
Random Numbers in Parallel Systems	263
Using numpy	264
Finding Prime Numbers	267
Queues of Work	273
Verifying Primes Using Interprocess Communication	278
Serial Solution	283
Naive Pool Solution	284
A Less Naive Pool Solution	285
Using Manager.Value as a Flag	286
Using Redis as a Flag	288
Using RawValue as a Flag	290
Using mmap as a Flag	291
Using mmap as a Flag Redux	293
Sharing numpy Data with multiprocessing	295
Synchronizing File and Variable Access	301
File Locking	302
Locking a Value	305
Wrap-Up	308
10. Clusters and Job Queues.....	311
Benefits of Clustering	312
Drawbacks of Clustering	313
\$462 Million Wall Street Loss Through Poor Cluster Upgrade Strategy	315

Skype’s 24-Hour Global Outage	315
Common Cluster Designs	316
How to Start a Clustered Solution	317
Ways to Avoid Pain When Using Clusters	318
Two Clustering Solutions	319
Using IPython Parallel to Support Research	319
Parallel Pandas with Dask	322
NSQ for Robust Production Clustering	326
Queues	327
Pub/sub	328
Distributed Prime Calculation	330
Other Clustering Tools to Look At	334
Docker	335
Docker’s Performance	335
Advantages of Docker	339
Wrap-Up	340
11. Using Less RAM.....	341
Objects for Primitives Are Expensive	342
The array Module Stores Many Primitive Objects Cheaply	344
Using Less RAM in NumPy with NumExpr	346
Understanding the RAM Used in a Collection	350
Bytes Versus Unicode	352
Efficiently Storing Lots of Text in RAM	353
Trying These Approaches on 11 Million Tokens	354
Modeling More Text with Scikit-Learn’s FeatureHasher	362
Introducing DictVectorizer and FeatureHasher	362
Comparing DictVectorizer and FeatureHasher on a Real Problem	365
SciPy’s Sparse Matrices	366
Tips for Using Less RAM	370
Probabilistic Data Structures	371
Very Approximate Counting with a 1-Byte Morris Counter	372
K-Minimum Values	375
Bloom Filters	379
LogLog Counter	385
Real-World Example	389
12. Lessons from the Field.....	393
Streamlining Feature Engineering Pipelines with Feature-engine	394
Feature Engineering for Machine Learning	394
The Hard Task of Deploying Feature Engineering Pipelines	395
Leveraging the Power of Open Source Python Libraries	395

Feature-engine Smooths Building and Deployment of Feature Engineering Pipelines	396
Helping with the Adoption of a New Open Source Package	397
Developing, Maintaining, and Encouraging Contribution to Open Source Libraries	398
Highly Performant Data Science Teams	400
How Long Will It Take?	400
Discovery and Planning	401
Managing Expectations and Delivery	402
Numba	403
A Simple Example	404
Best Practices and Recommendations	405
Getting Help	409
Optimizing Versus Thinking	409
Adaptive Lab's Social Media Analytics (2014)	412
Python at Adaptive Lab	413
SoMA's Design	413
Our Development Methodology	414
Maintaining SoMA	414
Advice for Fellow Engineers	415
Making Deep Learning Fly with RadimRehurek.com (2014)	415
The Sweet Spot	416
Lessons in Optimizing	417
Conclusion	420
Large-Scale Productionized Machine Learning at Lyst.com (2014)	420
Cluster Design	420
Code Evolution in a Fast-Moving Start-Up	421
Building the Recommendation Engine	421
Reporting and Monitoring	422
Some Advice	422
Large-Scale Social Media Analysis at Smesh (2014)	422
Python's Role at Smesh	423
The Platform	423
High Performance Real-Time String Matching	424
Reporting, Monitoring, Debugging, and Deployment	425
PyPy for Successful Web and Data Processing Systems (2014)	426
Prerequisites	427
The Database	428
The Web Application	428
OCR and Translation	429
Task Distribution and Workers	429
Conclusion	429

Task Queues at Lanyrd.com (2014)	430
Python's Role at Lanyrd	430
Making the Task Queue Performant	431
Reporting, Monitoring, Debugging, and Deployment	431
Advice to a Fellow Developer	431
Index.....	433

Foreword

When you think about high performance computing, you might imagine giant clusters of machines modeling complex weather phenomena or trying to understand signals in data collected about far-off stars. It's easy to assume that only people building specialized systems should worry about the performance characteristics of their code. By picking up this book, you've taken a step toward learning the theory and practices you'll need to write highly performant code. Every programmer can benefit from understanding how to build performant systems.

There are an obvious set of applications that are just on the edge of possible, and you won't be able to approach them without writing optimally performant code. If that's your practice, you're in the right place. But there is a much broader set of applications that can benefit from performant code.

We often think that new technical capabilities are what drives innovation, but I'm equally fond of capabilities that increase the accessibility of technology by orders of magnitude. When something becomes ten times cheaper in time or compute costs, suddenly the set of applications you can address is wider than you imagined.

The first time this principle manifested in my own work was over a decade ago, when I was working at a social media company, and we ran an analysis over multiple terabytes of data to determine whether people clicked on more photos of cats or dogs on social media.

It was dogs, of course. Cats just have better branding.

This was an outstandingly frivolous use of compute time and infrastructure at the time! Gaining the ability to apply techniques that had previously been restricted to sufficiently high-value applications, such as fraud detection, to a seemingly trivial question opened up a new world of now-accessible possibilities. We were able to take what we learned from these experiments and build a whole new set of products in search and content discovery.

For an example that you might encounter today, consider a machine-learning system that recognizes unexpected animals or people in security video footage. A sufficiently performant system could allow you to embed that model into the camera itself, improving privacy or, even if running in the cloud, using significantly less compute and power—benefiting the environment and reducing your operating costs. This can free up resources for you to look at adjacent problems, potentially building a more valuable system.

We all desire to create systems that are effective, easy to understand, and performant. Unfortunately, it often feels like we have to pick two (or one) out of the three! *High Performance Python* is a handbook for people who want to make things that are capable of all three.

This book stands apart from other texts on the subject in three ways. First, it's written for us—humans who write code. You'll find all of the context you need to understand why you might make certain choices. Second, Gorelick and Ozsváld do a wonderful job of curating and explaining the necessary theory to support that context. Finally, in this updated edition, you'll learn the specific quirks of the most useful libraries for implementing these approaches today.

This is one of a rare class of programming books that will change the way you think about the practice of programming. I've given this book to many people who could benefit from the additional tools it provides. The ideas that you'll explore in its pages will make you a better programmer, no matter what language or environment you choose to work in.

Enjoy the adventure.

— Hilary Mason,
Data Scientist in Residence at Accel

Preface

Python is easy to learn. You’re probably here because now that your code runs correctly, you need it to run faster. You like the fact that your code is easy to modify and you can iterate with ideas quickly. The trade-off between *easy to develop* and *runs as quickly as I need* is a well-understood and often-bemoaned phenomenon. There are solutions.

Some people have serial processes that have to run faster. Others have problems that could take advantage of multicore architectures, clusters, or graphics processing units. Some need scalable systems that can process more or less as expediency and funds allow, without losing reliability. Others will realize that their coding techniques, often borrowed from other languages, perhaps aren’t as natural as examples they see from others.

In this book we will cover all of these topics, giving practical guidance for understanding bottlenecks and producing faster and more scalable solutions. We also include some war stories from those who went ahead of you, who took the knocks so you don’t have to.

Python is well suited for rapid development, production deployments, and scalable systems. The ecosystem is full of people who are working to make it scale on your behalf, leaving you more time to focus on the more challenging tasks around you.

Who This Book Is For

You’ve used Python for long enough to have an idea about why certain things are slow and to have seen technologies like Cython, numpy, and PyPy being discussed as possible solutions. You might also have programmed with other languages and so know that there’s more than one way to solve a performance problem.

While this book is primarily aimed at people with CPU-bound problems, we also look at data transfer and memory-bound solutions. Typically, these problems are faced by scientists, engineers, quants, and academics.

We also look at problems that a web developer might face, including the movement of data and the use of just-in-time (JIT) compilers like PyPy and asynchronous I/O for easy-win performance gains.

It might help if you have a background in C (or C++, or maybe Java), but it isn't a prerequisite. Python's most common interpreter (CPython—the standard you normally get if you type `python` at the command line) is written in C, and so the hooks and libraries all expose the gory inner C machinery. There are lots of other techniques that we cover that don't assume any knowledge of C.

You might also have a lower-level knowledge of the CPU, memory architecture, and data buses, but again, that's not strictly necessary.

Who This Book Is Not For

This book is meant for intermediate to advanced Python programmers. Motivated novice Python programmers may be able to follow along as well, but we recommend having a solid Python foundation.

We don't cover storage-system optimization. If you have a SQL or NoSQL bottleneck, then this book probably won't help you.

What You'll Learn

Your authors have been working with large volumes of data, a requirement for *I want the answers faster!* and a need for scalable architectures, for many years in both industry and academia. We'll try to impart our hard-won experience to save you from making the mistakes that we've made.

At the start of each chapter, we'll list questions that the following text should answer. (If it doesn't, tell us and we'll fix it in the next revision!)

We cover the following topics:

- Background on the machinery of a computer so you know what's happening behind the scenes
- Lists and tuples—the subtle semantic and speed differences in these fundamental data structures
- Dictionaries and sets—memory allocation strategies and access algorithms in these important data structures
- Iterators—how to write in a more Pythonic way and open the door to infinite data streams using iteration
- Pure Python approaches—how to use Python and its modules effectively

- Matrices with `numpy`—how to use the beloved `numpy` library like a beast
- Compilation and just-in-time computing—processing faster by compiling down to machine code, making sure you’re guided by the results of profiling
- Concurrency—ways to move data efficiently
- `multiprocessing`—various ways to use the built-in `multiprocessing` library for parallel computing and to efficiently share `numpy` matrices, and some costs and benefits of interprocess communication (IPC)
- Cluster computing—convert your `multiprocessing` code to run on a local or remote cluster for both research and production systems
- Using less RAM—approaches to solving large problems without buying a humungous computer
- Lessons from the field—lessons encoded in war stories from those who took the blows so you don’t have to

Python 3

Python 3 is the standard version of Python as of 2020, with Python 2.7 deprecated after a 10-year migration process. If you’re still on Python 2.7, you’re doing it wrong—many libraries are no longer supported for your line of Python, and support will become more expensive over time. Please do the community a favor and migrate to Python 3, and make sure that all new projects use Python 3.

In this book, we use 64-bit Python. Whilst 32-bit Python is supported, it is far less common for scientific work. We’d expect all the libraries to work as usual, but numeric precision, which depends on the number of bits available for counting, is likely to change. 64-bit is dominant in this field, along with *nix environments (often Linux or Mac). 64-bit lets you address larger amounts of RAM. *nix lets you build applications that can be deployed and configured in well-understood ways with well-understood behaviors.

If you’re a Windows user, you’ll have to buckle up. Most of what we show will work just fine, but some things are OS-specific, and you’ll have to research a Windows solution. The biggest difficulty a Windows user might face is the installation of modules: research in sites like Stack Overflow should give you the solutions you need. If you’re on Windows, having a virtual machine (e.g., using VirtualBox) with a running Linux installation might help you to experiment more freely.

Windows users should definitely look at a packaged solution like those available through Anaconda, Canopy, Python(x,y), or Sage. These same distributions will make the lives of Linux and Mac users far simpler too.

Changes from Python 2.7

If you've upgraded from Python 2.7, you might not be aware of a few relevant changes:

- `/` meant *integer* division in Python 2.7, whereas it performs *float* division in Python 3.
- `str` and `unicode` were used to represent text data in Python 2.7; in Python 3, everything is a `str`, and these are always Unicode. For clarity, a `bytes` type is used if we're using unencoded byte sequences.

If you're in the process of upgrading your code, two good guides are “[Porting Python 2 Code to Python 3](#)” and “[Supporting Python 3: An in-depth guide](#)”. With a distribution like Anaconda or Canopy, you can run both Python 2 and Python 3 simultaneously—this will simplify your porting.

License

This book is licensed under [Creative Commons Attribution-NonCommercial-NoDerivs 3.0](#).

You're welcome to use this book for noncommercial purposes, including for noncommercial teaching. The license allows only for complete reproductions; for partial reproductions, please contact O'Reilly (see “[How to Contact Us](#)” on page xviii). Please attribute the book as noted in the following section.

We negotiated that the book should have a Creative Commons license so the contents could spread further around the world. We'd be quite happy to receive a beer if this decision has helped you. We suspect that the O'Reilly staff would feel similarly about the beer.

How to Make an Attribution

The Creative Commons license requires that you attribute your use of a part of this book. Attribution just means that you should write something that someone else can follow to find this book. The following would be sensible: “*High Performance Python*, 2nd ed., by Micha Gorelick and Ian Ozsváld (O'Reilly). Copyright 2020 Micha Gorelick and Ian Ozsváld, 978-1-492-05502-0.”

Errata and Feedback

We encourage you to review this book on public sites like Amazon—please help others understand if they would benefit from this book! You can also email us at feedback@highperformancepython.com.

We're particularly keen to hear about errors in the book, successful use cases where the book has helped you, and high performance techniques that we should cover in the next edition. You can access the web page for this book at <https://oreil.ly/high-performance-python-2e>.

Complaints are welcomed through the instant-complaint-transmission-service > /dev/null.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, datatypes, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip, suggestion, or critical thinking question.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/mynameisfiber/high_performance_python_2e.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and more information about our books and courses, see our website at <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://youtube.com/oreillymedia>

Acknowledgments

Hilary Mason wrote our foreword—thanks for composing such a wonderful opening narrative for our book. Giles Weaver and Dimitri Denisonok provided invaluable technical feedback on this edition; great work, chaps.

Thanks to Patrick Cooper, Kyran Dale, Dan Foreman-Mackey, Calvin Giles, Brian Granger, Jamie Matthews, John Montgomery, Christian Schou Oxvig, Matt “snakes” Reiferson, Balthazar Rouberol, Michael Skirpan, Luke Underwood, Jake Vanderplas, and William Winter for invaluable feedback and contributions.

Ian thanks his wife, Emily, for letting him disappear for another eight months to write this second edition (thankfully, she's terribly understanding). Ian apologizes to his dog for sitting and writing rather than walking in the woods quite as much as she'd have liked.

Micha thanks Marion and the rest of his friends and family for being so patient while he learned to write.

O'Reilly editors are rather lovely to work with; do strongly consider talking to them if you want to write your own book.

Our contributors to the “Lessons from the Field” chapter very kindly shared their time and hard-won lessons. We give thanks to Soledad Galli, Linda Uruchurtu, Vanentin Haenel, and Vincent D. Warmerdam for this edition and to Ben Jackson, Radim Řehůřek, Sebastjan Trepca, Alex Kelly, Marko Tasic, and Andrew Godwin for their time and effort during the previous edition.

Understanding Performant Python

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](#)” on [page 8](#).

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.

¹ Not to be confused with interprocess communication, which shares the same acronym—we’ll look at that topic in [Chapter 9](#).

Finally, and most important for the higher-level programmer, there is the prevalence of multicore architectures. These architectures include multiple CPUs within the same unit, 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!

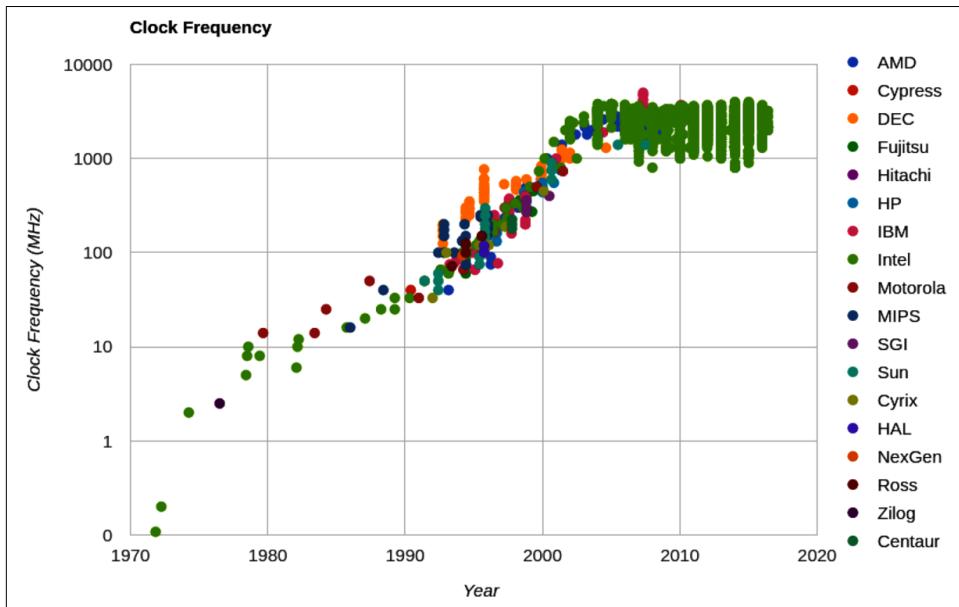


Figure 1-1. Clock speed of CPUs over time (from [CPU DB](#))

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.

Furthermore, 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` ([Chapter 9](#)), technologies like `numpy` or `numexpr` ([Chapter 6](#)), Cython ([Chapter 7](#)), or distributed models of computing ([Chapter 10](#)).



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. 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 (10 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 (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.

² Speeds in this section are from <https://oreil.ly/pToi7>.

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—most of these processes can happen independently, while other calculations are being performed!

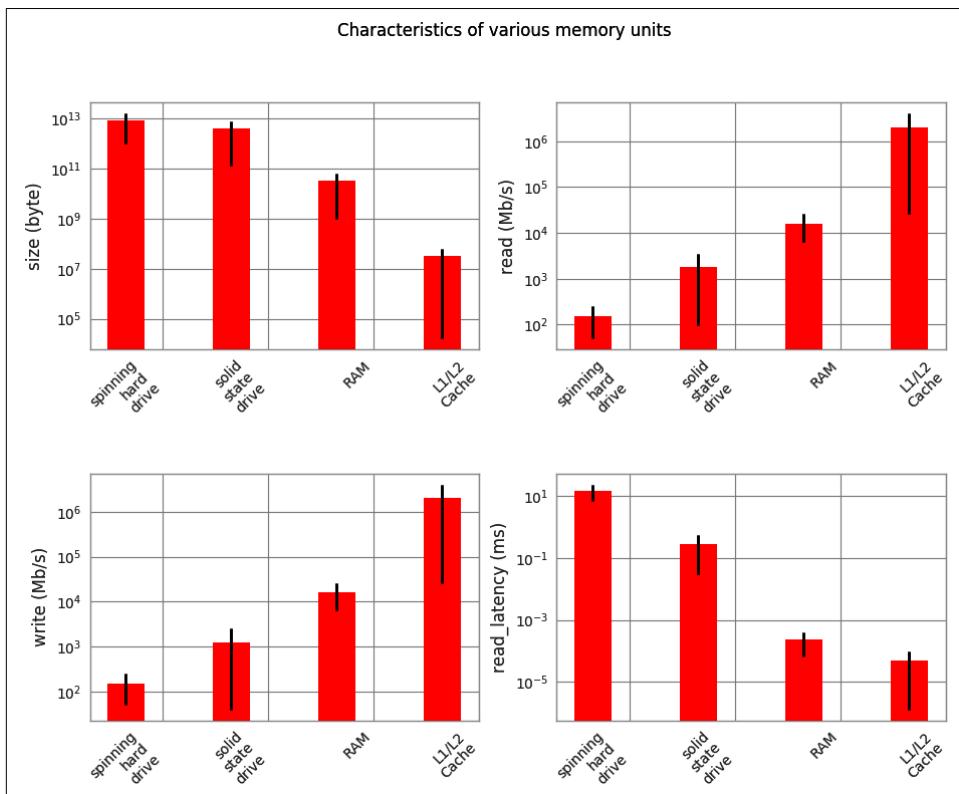


Figure 1-2. Characteristic values for different types of memory units (values from February 2014)

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).

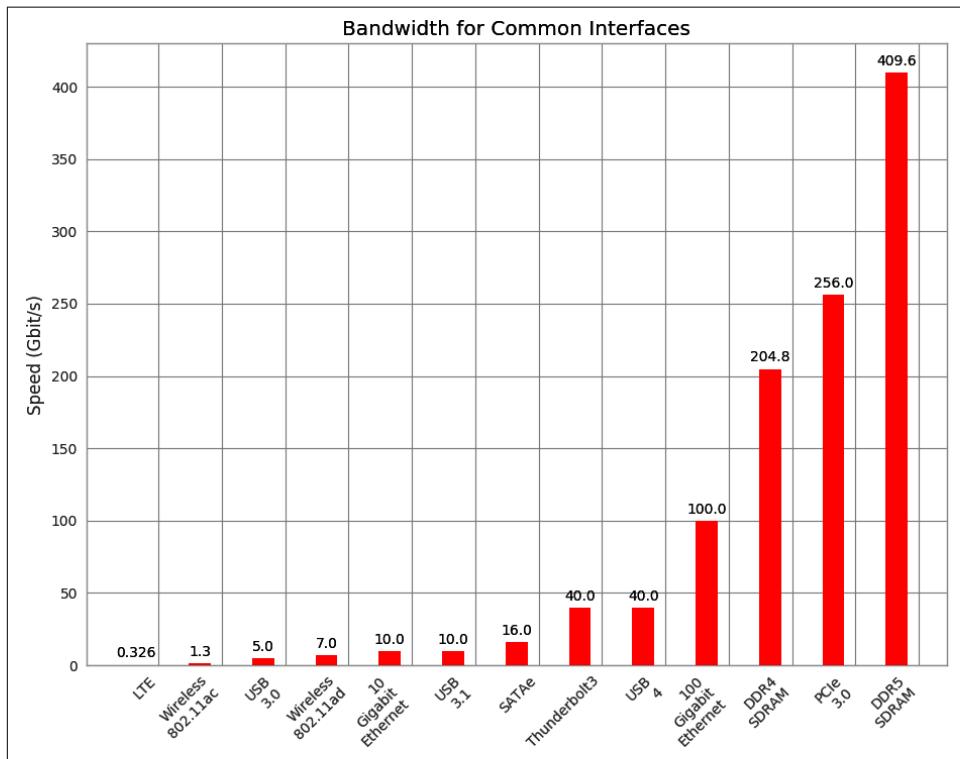


Figure 1-3. Connection speeds of various common interfaces³

³ Data is from <https://oreil.ly/7SC8d>.

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(10_000_000)}")
# check_prime(10,000,000) = False
print(f"check_prime(10,000,019) = {check_prime(10_000_019)}")
# 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.



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):
    sqrt_number = math.sqrt(number)
    numbers = range(2, int(sqrt_number)+1)
    for i in range(0, len(numbers), 5):
        # the following line is not valid Python code
        result = (number / numbers[i:(i + 5)]).is_integer()
        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 five 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 [Chapter 6](#).

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 haystack))

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

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.

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 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.

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 \sqrt{N} . 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 \sqrt{N} / 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.

⁴ In [Chapter 6](#), we'll see how we can regain this control and tune our code all the way down to the memory utilization patterns.

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:

`unicode and bytes`

Baked into the core language

`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`

`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, MongoDB, HDF5, and SQL

Web development frameworks

Performant systems for creating websites, such as `aiohttp`, `django`, `pyramid`, `flask`, and `tornado`

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
- Sage, a Matlab-like environment that includes an integrated development environment (IDE)
- 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 “[Cython](#)” on page 167), 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.

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.

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.

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.



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 `pipenv` coupled with Docker. Both are sensible solutions and are significantly better 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.

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.

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 [Bulwark library](#) is an example of a testing framework focused on Pandas 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 five 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 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 his personal life and spend days away from the computer to work on nontechnical projects. 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.

Profiling to Find Bottlenecks

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 \times to 100 \times 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” on page 35](#)), 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” on page 40](#)), 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 ([Chapter 7](#)).

In [Chapter 6](#), 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 [Example 6-8](#) 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” on page 46](#)). 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.



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” on page 55](#)), 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.

Before the end of the chapter, we'll review how to integrate unit tests while profiling ([“Unit Testing During Optimization to Maintain Correctness” on page 59](#)) 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” on page 62](#)) 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 [Chapter 7](#) 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.

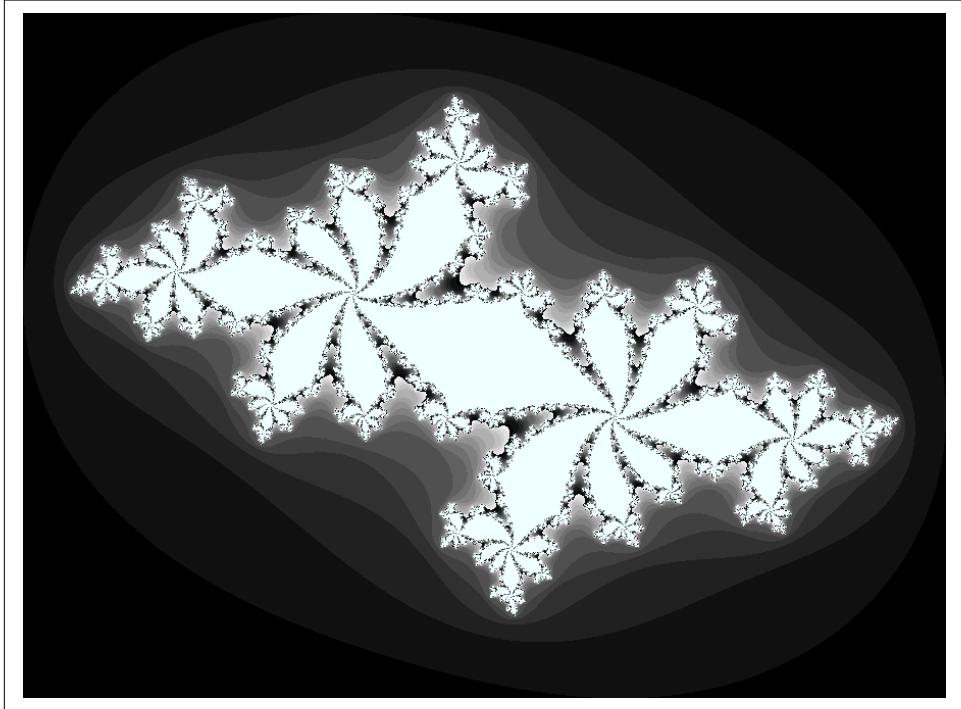


Figure 2-1. Julia set plot with a false gray scale to highlight detail

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 iterations per point  
        if abs(z) < 2.0: # has the escape condition been broken?  
            z = z*z + c  
        else:  
            break  
    # store the iteration count for each z and draw later
```

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 $\text{abs}(z) < 2$ 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 $2.54 \geq 2.0$, 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}, c={c: .5f}")  
  
0: z=-0.62772-0.42193j, abs(z)=0.756, c=-0.62772-0.42193j  
1: z=-0.41171+0.10778j, abs(z)=0.426, c=-0.62772-0.42193j  
2: z=-0.46983-0.51068j, abs(z)=0.694, c=-0.62772-0.42193j  
3: z=-0.66777+0.05793j, abs(z)=0.670, c=-0.62772-0.42193j  
4: z=-0.18516-0.49930j, abs(z)=0.533, c=-0.62772-0.42193j  
5: z=-0.84274-0.23703j, abs(z)=0.875, c=-0.62772-0.42193j  
6: z= 0.02630-0.02242j, abs(z)=0.035, c=-0.62772-0.42193j  
7: z=-0.62753-0.42311j, abs(z)=0.757, c=-0.62772-0.42193j  
8: z=-0.41295+0.10910j, abs(z)=0.427, c=-0.62772-0.42193j
```

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 (`max_iter`) 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 $+2$.

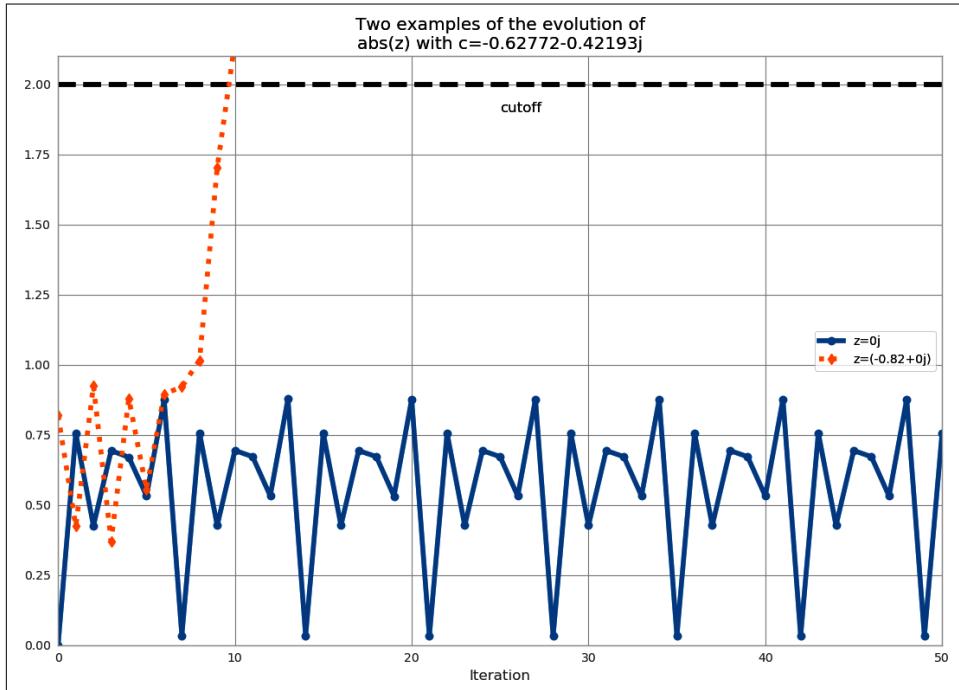


Figure 2-2. Two coordinate examples evolving for the Julia set

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 image drawing"""
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, -0.42193
```

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_iterations):
    """Create a list of complex coordinates (zs) and complex parameters (cs),
    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 condition for each cell.
    # Note that our initial condition is a constant and could easily be removed,
    # we use it to simulate a real-world scenario with several inputs to our
    # 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_iterations, zs, cs)
end_time = time.time()
secs = end_time - start_time
print(calculate_z_serial_purepython.__name__ + " took", secs, "seconds")

# This sum is expected for a 1000^2 grid with 300 iterations
# It ensures that our code evolves exactly as we'd intended
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 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

```

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 solution with
    # reasonable defaults for a laptop
    calc_pure_python(desired_width=1000, max_iterations=300)
```

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

```
# running the above produces:
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 8.087012767791748 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.

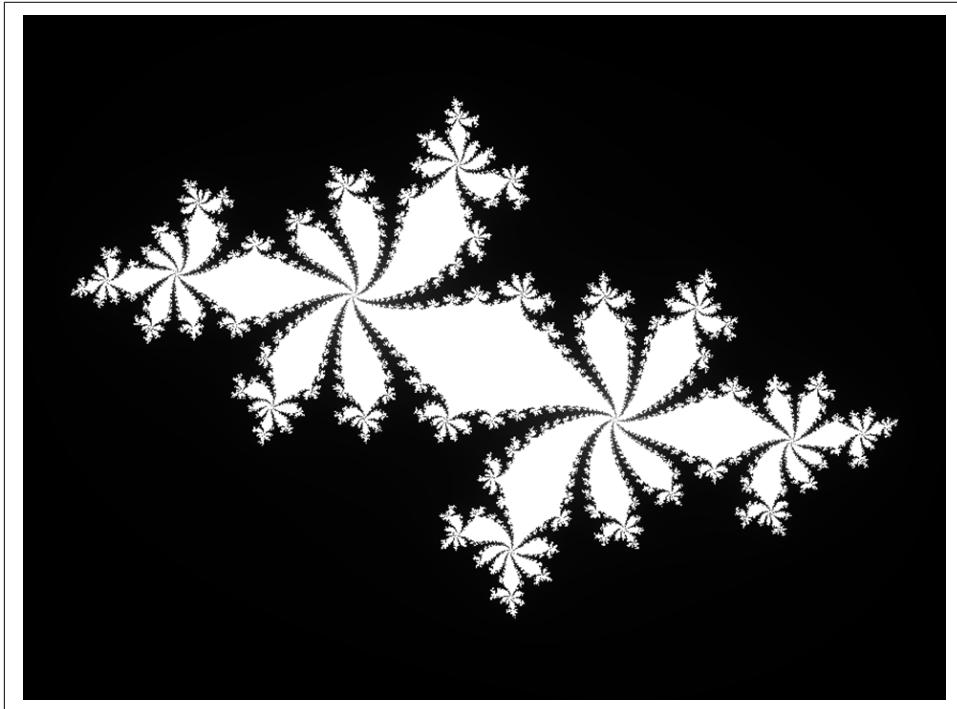


Figure 2-3. Julia plot example using a pure gray scale

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 8 seconds to run using CPython 3.7. 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 9550 with an Intel Core I7 6700HQ (2.6 GHz, 6 MB cache, Quad Core with Hyperthreading) and 32 GB of RAM running Linux Mint 19.1 (Ubuntu 18.04).

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):
        pass
```

```

t1 = time.time()
result = fn(*args, **kwargs)
t2 = time.time()
print(f"@timefn: {fn.__name__} took {t2 - t1} seconds")
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: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 8.00485110282898 seconds
calculate_z_serial_purepython took 8.004898071289062 seconds

```



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.



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" \
    "julia1.calc_pure_python(desired_width=1000, max_iterations=300)"

```

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:

```
5 loops, best of 1: 8.45 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 8.45 seconds (as the best case), while single calls to `calculate_z_serial_purepython` take 8.0 seconds as measured by the `@timedfn` decorator. The difference is mainly the time taken to create the `zs` and `cs` lists.

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  
In [2]: %timeit julia1.calc_pure_python(desired_width=1000, max_iterations=300)
```



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.

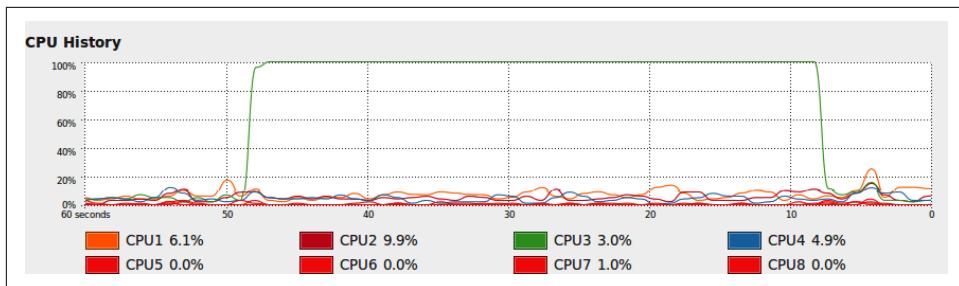


Figure 2-4. System Monitor on Ubuntu showing variation in background CPU usage while we time our function

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: 1000
Total elements: 1000000
calculate_z_serial_purepython took 8.279886722564697 seconds
real 8.84
user 8.73
sys 0.10
```

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` quick-and-dirty 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:

```
$ /usr/bin/time --verbose python julia1_nopil.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 8.477287530899048 seconds
  Command being timed: "python julia1_nopil.py"
  User time (seconds): 8.97
  System time (seconds): 0.05
  Percent of CPU this job got: 99%
  Elapsed (wall clock) time (h:mm:ss or m:ss): 0:09.03
  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): 98620
  Average resident set size (kbytes): 0
  Major (requiring I/O) page faults: 0
  Minor (reclaiming a frame) page faults: 26645
  Voluntary context switches: 1
  Involuntary context switches: 27
  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
```

Probably the most useful indicator here is `Major (requiring I/O) page faults`, as 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.

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.



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.py
...
Length of x: 1000
Total elements: 1000000
```

```
calculate_z_serial_purepython took 11.498265266418457 seconds
36221995 function calls in 12.234 seconds
```

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	12.234	12.234	{built-in method builtins.exec}
1	0.038	0.038	12.234	12.234	julia1_nopil.py:1(<module>)
1	0.571	0.571	12.197	12.197	julia1_nopil.py:23 (calc_pure_python)
1	8.369	8.369	11.498	11.498	julia1_nopil.py:9 (calculate_z_serial_purepython)
34219980	3.129	0.000	3.129	0.000	{built-in method builtins.abs}
2002000	0.121	0.000	0.121	0.000	{method 'append' of 'list' objects}
1	0.006	0.006	0.006	0.006	{built-in method builtins.sum}
3	0.000	0.000	0.000	0.000	{built-in method builtins.print}
2	0.000	0.000	0.000	0.000	{built-in method time.time}
4	0.000	0.000	0.000	0.000	{built-in method builtins.len}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

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 12 seconds (this time includes the overhead of using `cProfile`). Previously, our code took around 8 seconds to execute—we've just added a 4-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 12 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 11 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 3 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 3 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.



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 3.1 seconds. We know that `calculate_z_serial_pure python` costs 11.4 seconds in total.

The final line of the profiling output refers to `lspref`; 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.py
```

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 0x7f77088edf28>
```

```
In [4]: p.print_stats()
Fri Jun 14 17:59:28 2019      profile.stats

    36221995 function calls in 12.169 seconds

Ordered by: cumulative time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
          1    0.000    0.000   12.169   12.169 {built-in method builtins.exec}
          1    0.033    0.033   12.169   12.169 julia1_nopil.py:1(<module>)
          1    0.576    0.576   12.135   12.135 julia1_nopil.py:23
                                         (calc_pure_python)
          1    8.266    8.266   11.429   11.429 julia1_nopil.py:9
                                         (calculate_z_serial_purepython)
34219980    3.163    0.000   3.163    0.000 {built-in method builtins.abs}
2002000    0.123    0.000   0.123    0.000 {method 'append' of 'list' objects}
          1    0.006    0.006   0.006    0.006 {built-in method builtins.sum}
          3    0.000    0.000   0.000    0.000 {built-in method builtins.print}
          4    0.000    0.000   0.000    0.000 {built-in method builtins.len}
          2    0.000    0.000   0.000    0.000 {built-in method time.time}
          1    0.000    0.000   0.000    0.000 {method 'disable' of
                                         '_lsprof.Profiler' objects}
```

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 by...
                                              ncalls  tottime  cumtime
{built-in method builtins.exec}      <-      1    0.033   12.169
julia1_nopil.py:1(<module>)        <-      1    0.576   12.135
julia1_nopil.py:23(calc_pure_python) <-      1    8.266   11.429
julia1_nopil.py:9(...)            <-      1    3.163   3.163
{built-in method builtins.abs}       <- 34219980    0.123   0.123
{method 'append' of 'list' objects}  <- 2002000    0.006   0.006
{built-in method builtins.sum}       <-      3    0.000   0.000
{built-in method builtins.print}    <-      2    0.000   0.000
{built-in method builtins.len}      <-      2    0.000   0.000
```

```
{built-in method time.time}           <-      2    0.000    0.000  
:23(calc_pure_python)
```

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		called...
{built-in method builtins.exec}	->	ncalls tottime cumtime
1 0.033 12.169		julia1_nopil.py:1(<module>)
1 0.576 12.135		julia1_nopil.py:23
		(calc_pure_python)
julia1_nopil.py:23(calc_pure_python)	->	1 8.266 11.429
		julia1_nopil.py:9
		(calculate_z_serial_purepython)
		2 0.000 0.000
		{built-in method builtins.len}
		3 0.000 0.000
		{built-in method builtins.print}
		1 0.006 0.006
		{built-in method builtins.sum}
		2 0.000 0.000
		{built-in method time.time}
		2002000 0.123 0.123
		{method 'append' of 'list' objects}
julia1_nopil.py:9(...)	->	34219980 3.163 3.163
		{built-in method builtins.abs}
		{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 25% 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.

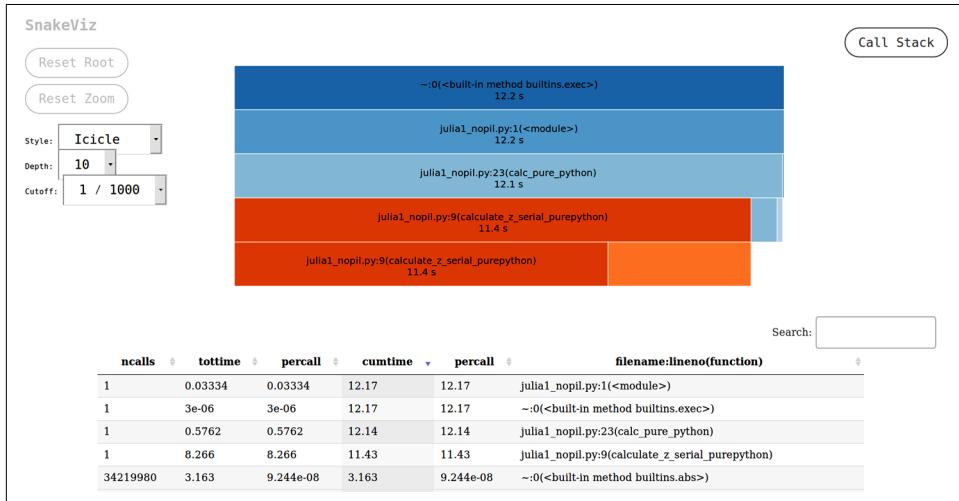


Figure 2-5. snakeviz visualizing profile.stats

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.lprof
Timer unit: 1e-06 s

Total time: 49.2011 s
File: julia1_lineprofiler.py
Function: calculate_z_serial_purepython at line 9

Line #      Hits  Per Hit   % Time  Line Contents
=====
9                      @profile
10                     def calculate_z_serial_purepython(maxiter,
11                                         zs, cs):
12             """Calculate output list using Julia update rule"""
13             1    3298.0    0.0      output = [0] * len(zs)
14             1000001    0.4    0.8      for i in range(len(zs)):
15             1000000    0.4    0.7      n = 0
16             1000000    0.4    0.9      z = zs[i]
17             34219980   0.5    38.0     while abs(z) < 2 and n < maxiter:
18             33219980   0.5    30.8     z = z * z + c
19             33219980   0.4    27.1     n += 1
20             1000000    0.4    0.8      output[i] = n
21                 1    1.0    0.0      return output
```

Introducing `kernprof.py` adds a substantial amount to the runtime. In this example, `calculate_z_serial_purepython` takes 49 seconds; this is up from 8 seconds using simple `print` statements and 12 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 38% 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 ([Chapter 7](#)) give us a massive win. The creation of the `output` list and the updates on line 20 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.



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.lprof
```

```

Timer unit: 1e-06 s

Total time: 82.88 s
File: julia1_lineprofiler2.py
Function: calculate_z_serial_purepython at line 9

Line #      Hits  Per Hit   % Time  Line Contents
=====
9           @profile
10          def calculate_z_serial_purepython(maxiter,
11                                         zs, cs):
12              """Calculate output list using Julia update rule"""
13              1    3309.0    0.0      output = [0] * len(zs)
14      1000001    0.4    0.5      for i in range(len(zs)):
15      1000000    0.4    0.5      n = 0
16      1000000    0.5    0.5      z = zs[i]
17      1000000    0.4    0.5      c = cs[i]
18      34219980   0.6   23.1      while True:
19      34219980   0.4   18.3          not_yet_escaped = abs(z) < 2
20      34219980   0.4   17.3          iterations_left = n < maxiter
21      33219980   0.5   20.5          if not_yet_escaped and iterations_left:
22      33219980   0.4   17.3              z = z * z + c
23                               n += 1
24      1000000    0.4    0.5          else:
25      1000000    0.4    0.5              break
26              1    0.0    0.0      return output

```

This version takes 82 seconds to execute, while the previous version took 49 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 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits', or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.

```

```

In [1]: z = 0+0j
In [2]: %timeit abs(z) < 2
97.6 ns ± 0.138 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
In [3]: n = 1
In [4]: maxiter = 300
In [5]: %timeit n < maxiter
42.1 ns ± 0.0355 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

```

From this simple analysis, it looks as though the logic test on `n` is more than two 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. The overheads of `line_profiler` also confuse the result, and the results on line 17 for both versions are similar. We should reject the hypothesis that in Python 3.7 changing the order of the logic results in a consistent speedup—there’s no clear evidence for this. Ian notes that with Python 2.7 we *could* accept this hypothesis, but with Python 3.7 that’s no longer the case.

Using a more suitable approach to solve this problem (e.g., swapping to using Cython or PyPy, as described in [Chapter 7](#)) 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.lprof
Timer unit: 1e-06 s

Total time: 48.9154 s
File: julia1_lineprofiler3.py
Function: calculate_z_serial_purepython at line 9

Line #    Hits  Per Hit % Time  Line Contents
=====
9          @profile
10         def calculate_z_serial_purepython(maxiter,
11                                         zs, cs):
12             """Calculate output list using Julia update rule"""
13             1    3312.0   0.0      output = [0] * len(zs)
14             1000001   0.4   0.8      for i in range(len(zs)):
15             1000000   0.4   0.7          n = 0
16             1000000   0.4   0.8          z = zs[i]
17             1000000   0.4   0.8          c = cs[i]
18             34219980   0.5   38.2      while n < maxiter and abs(z) < 2:
19             33219980   0.5   30.7          z = z * z + c
20             33219980   0.4   27.1          n += 1
21             1000000   0.4   0.8      output[i] = n
22             1    1.0   0.0      return output
```

As expected, we can see from the output in [Example 2-9](#) that `calculate_z_serial_purepython` takes most (97%) 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: 88.334 s
File: julia1_lineprofiler3.py
Function: calc_pure_python at line 24

Line #    Hits  Per Hit % Time  Line Contents
=====
24          @profile
25         def calc_pure_python(draw_output,
26                               desired_width,
27                               max_iterations):
28             """Create a list of complex...
...
44             1        1.0     0.0      zs = []
45             1        0.0     0.0      cs = []
46             1001    0.7     0.0      for ycoord in y:
47             1001000  0.6     0.7      for xcoord in x:
```

```

48 1000000      0.9    1.0      zs.append(complex(xcoord, ycoord))
49 1000000      0.9    1.0      cs.append(complex(c_real, c_imag))
50
51     1      40.0    0.0      print("Length of x:", len(x))
52     1       7.0    0.0      print("Total elements:", len(zs))
53     1       4.0    0.0      start_time = time.time()
54     1 85969310.0   97.3      output = calculate_z_serial_purepython \
                                (max_iterations, zs, cs)
55     1       4.0    0.0      end_time = time.time()
56     1       1.0    0.0      secs = end_time - start_time
57     1      36.0    0.0      print(calculate_z_serial...
58
59     1      6345.0   0.0      assert sum(output) == 33219980

```

`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.



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](#)” on [page 60](#).

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 12, 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 46, we see that the allocation of the `zs` and `cs` lists changes the `Mem usage` column from 48 MB to 125 MB (a change of +77 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.

¹ `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.

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_memoryprofiler.py
...
Line #    Mem usage    Increment   Line Contents
=====
9  126.363 MiB  126.363 MiB  @profile
10                      def calculate_z_serial_purepython(maxiter,
11                                         zs, cs):
12                                         """Calculate output list using...
13                                         output = [0] * len(zs)
14                                         for i in range(len(zs)):
15                                             n = 0
16                                             z = zs[i]
17                                             c = cs[i]
18                                             while n < maxiter and abs(z) < 2:
19                                                 z = z * z + c
20                                                 n += 1
21                                             output[i] = n
22                                         return output
...
Line #    Mem usage    Increment   Line Contents
=====
24  48.113 MiB  48.113 MiB  @profile
25                      def calc_pure_python(draw_output,
26                                         desired_width,
27                                         max_iterations):
28                                         """Create a list of complex...
29                                         x_step = (x2 - x1) / desired_width
30                                         y_step = (y1 - y2) / desired_width
31                                         x = []
32                                         y = []
33                                         ycoord = y2
34                                         while ycoord > y1:
35                                             y.append(ycoord)
36                                         ycoord += y_step
37                                         xcoord = x1
38                                         while xcoord < x2:
39                                             x.append(xcoord)
40                                         xcoord += x_step
41                                         zs = []
42                                         cs = []
43                                         for ycoord in y:
44                                             for xcoord in x:
45                                                 zs.append(complex(xcoord, ycoord))
46                                                 cs.append(complex(c_real, c_imag))
47                                         print("Length of x:", len(x))
```

```
52 125.961 MiB    0.000 MiB      print("Total elements:", len(zs))
53 125.961 MiB    0.000 MiB      start_time = time.time()
54 136.609 MiB    10.648 MiB     output = calculate_z_serial...
55 136.609 MiB    0.000 MiB      end_time = time.time()
56 136.609 MiB    0.000 MiB      secs = end_time - start_time
57 136.609 MiB    0.000 MiB      print(calculate_z_serial_purepython...)
58
59 136.609 MiB    0.000 MiB      assert sum(output) == 33219980
```

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.

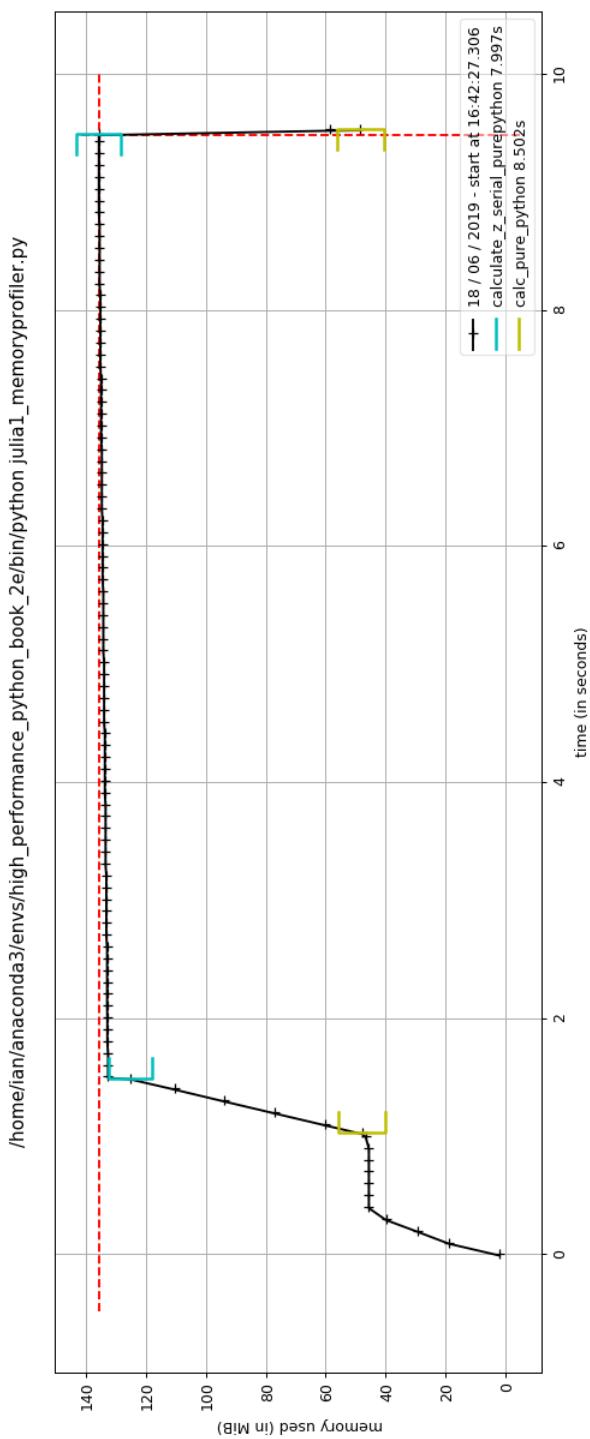


Figure 2-6. *memory_profiler* report using *mprof*

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 rule"""
    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!

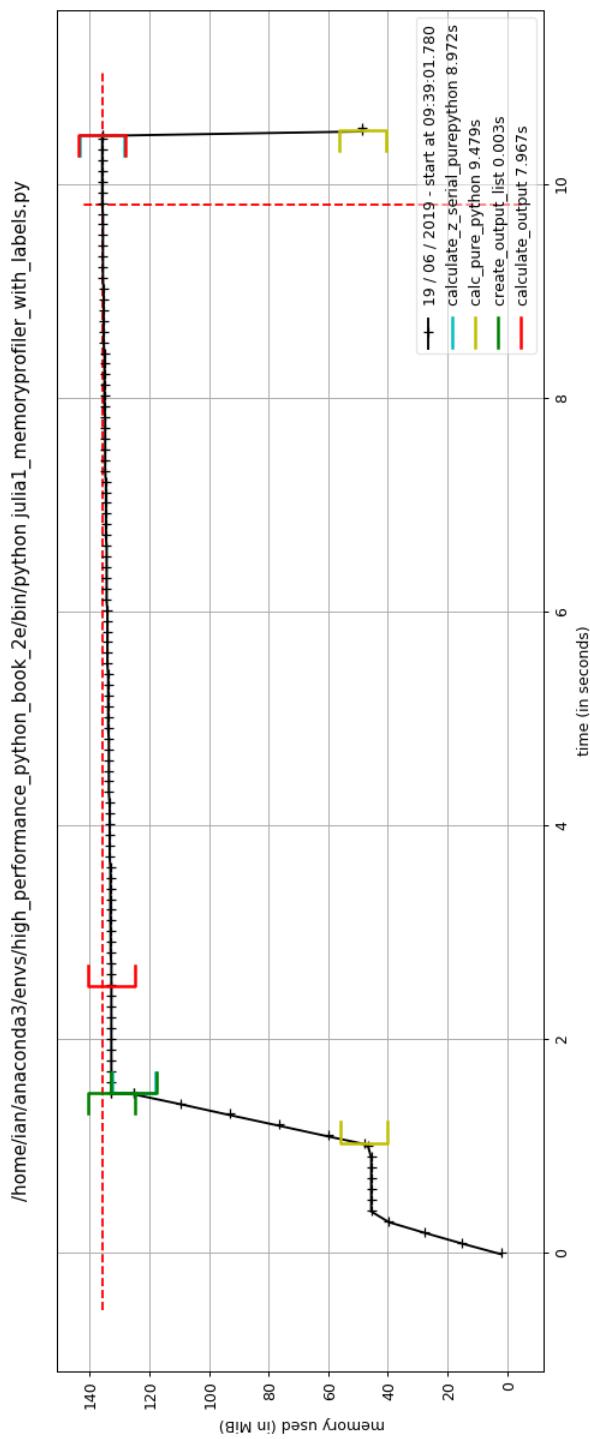


Figure 2-7. *memory_profiler* report using *mprof* with labels

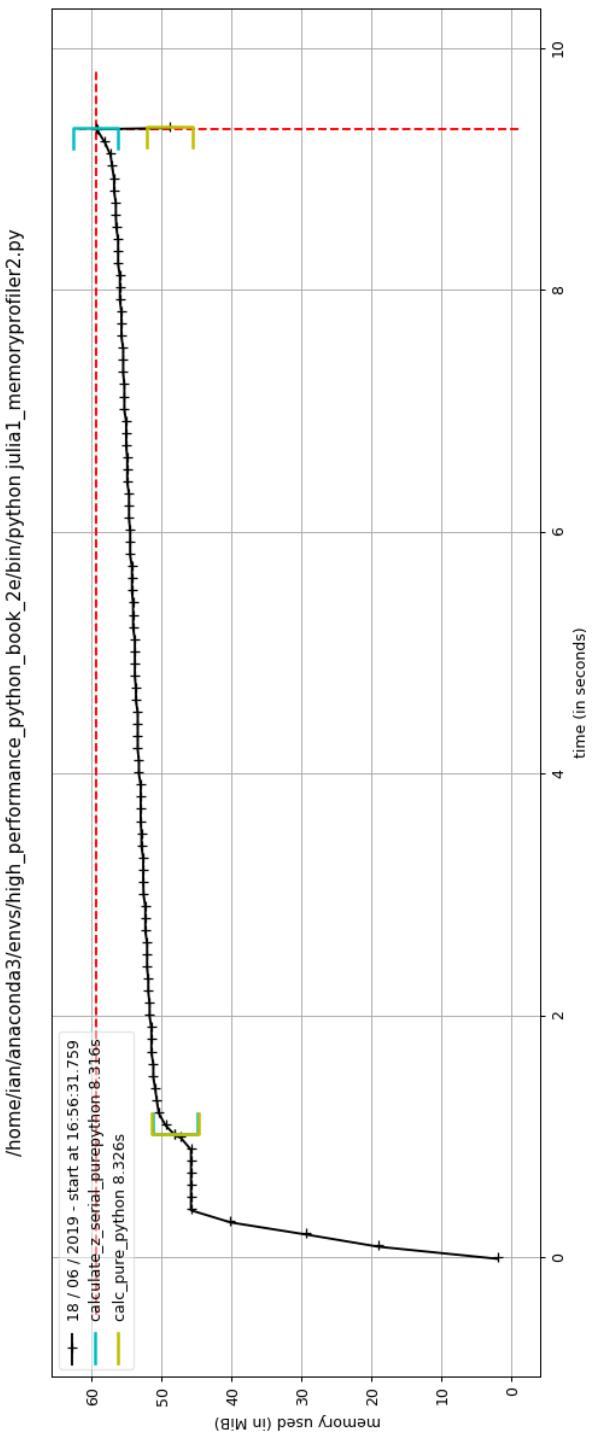


Figure 2-8. `memory_profiler` after removing two large lists

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 rule"""
    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 [Chapter 11](#), 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-mem=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.

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.

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 2-13](#).

Example 2-13. Running PySpy at the command line

```
$ ps -A -o pid,rss,cmd | ack python
...
15953 96156 python julia1_nopil.py
```

```
...
$ sudo env "PATH=$PATH" py-spy --pid 15953
```

In [Figure 2-9](#), 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.

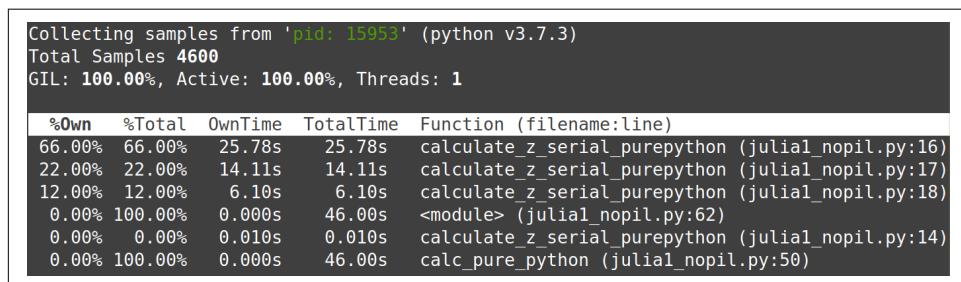


Figure 2-9. Introspecting a Python process using PySpy

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 --flame profile.svg -- python julia1_nopil.py`. You'll see in [Figure 2-10](#) that the width of the display represents the entire program's runtime, and each layer moving down the image represents functions called from above.

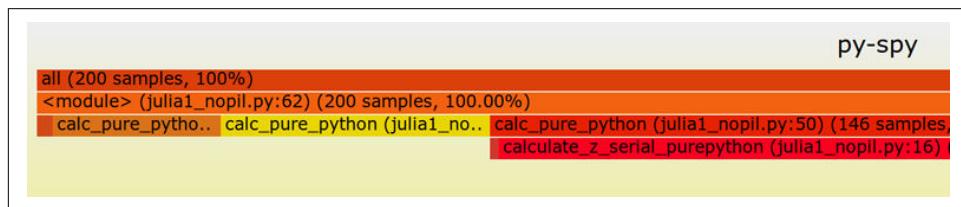


Figure 2-10. Part of a flame chart for PySpy

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 2-14](#), we disassemble the outer loop of our CPU-bound function.



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 2-14. 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_purepython)
  1           0 LOAD_CONST              1 (0)
  2           2 BUILD_LIST               1
  4           4 LOAD_GLOBAL              0 (len)
  6           6 LOAD_FAST                 1 (zs)
  8           8 CALL_FUNCTION            1
 10          10 BINARY_MULTIPLY
 12          12 STORE_FAST               3 (output)

 14          14 SETUP_LOOP               94 (to 110)
 16          16 LOAD_GLOBAL              1 (range)
 18          18 LOAD_GLOBAL              0 (len)
 20          20 LOAD_FAST                 1 (zs)
 22          22 CALL_FUNCTION            1
 24          24 CALL_FUNCTION            1
 26          26 GET_ITER
  >>    28 FOR_ITER                 78 (to 108)
 30          30 STORE_FAST               4 (i)

 13          32 LOAD_CONST              1 (0)
 34          34 STORE_FAST               5 (n)
...
 19          >>  98 LOAD_FAST               5 (n)
 100         100 LOAD_FAST              3 (output)
 102         102 LOAD_FAST               4 (i)
 104         104 STORE_SUBSCR
 106         106 JUMP_ABSOLUTE          28
  >>  108 POP_BLOCK

 20          >> 110 LOAD_FAST               3 (output)
 112         112 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. Inside Python line 12, it calls the `len` function from the stack, which consumes the `zs` reference in the stack; then it applies a binary multiply to the last two arguments (the length of `zs` and the single-element list) and stores the result 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).



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

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.²

—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.

² 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!

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 2-15](#). Both do the same job, but the first generates a lot of additional Python bytecode, which will cause more overhead.

Example 2-15. 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 identical results from both functions"
```

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 2-16](#), 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 2-16. Using %timeit to test our hypothesis that using built-in functions should be faster than writing our own functions

```
In [2]: %timeit fn_expressive()
52.4 ms ± 86.4 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [3]: %timeit fn_terse()
18.1 ms ± 1.38 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

If we use the `dis` module to investigate the code for each function, as shown in [Example 2-17](#), we can see that the virtual machine has 17 lines to execute with the more expressive function and only 6 to execute with the very readable but terser second function.

Example 2-17. Using dis to view the number of bytecode instructions involved in our two functions

```
In [4]: import dis
In [5]: dis.dis(fn_expressive)
2           0 LOAD_CONST               1 (0)
2           2 STORE_FAST                1 (total)
```

```

3           4 SETUP_LOOP           24 (to 30)
            6 LOAD_GLOBAL          0 (range)
            8 LOAD_FAST             0 (upper)
           10 CALL_FUNCTION         1
           12 GET_ITER
>>  14 FOR_ITER              12 (to 28)
   16 STORE_FAST              2 (n)

4           18 LOAD_FAST             1 (total)
   20 LOAD_FAST              2 (n)
   22 INPLACE_ADD
   24 STORE_FAST             1 (total)
   26 JUMP_ABSOLUTE          14
>>  28 POP_BLOCK

5     >> 30 LOAD_FAST             1 (total)
   32 RETURN_VALUE

```

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

```

8           0 LOAD_GLOBAL          0 (sum)
   2 LOAD_GLOBAL          1 (range)
   4 LOAD_FAST             0 (upper)
   6 CALL_FUNCTION          1
   8 CALL_FUNCTION          1
  10 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.



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 2-18](#). It has a test (for `pytest`) and a function that we've been profiling with either `line_profiler` or `memory_profiler`.

Example 2-18. 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 test and profile""""
```

```

# artificial "we're doing something clever and expensive" delay
time.sleep(1)
return useful_input ** 2

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

```

If we run `pytest` on our code, we'll get a `NameError`, as shown in [Example 2-19](#).

Example 2-19. A missing decorator during testing breaks out tests in an unhelpful way!

```

$ pytest utility.py
=====
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.8.0, pluggy-0.12.0
rootdir: noop_profile_decorator
plugins: cov-2.7.1
collected 0 items / 1 errors

=====
ERRORS =====
____ ERROR collecting utility.py _____
utility.py:20: in <module>
    @profile
E  NameError: name 'profile' is not defined

```

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 2-20](#).

Example 2-20. Add a no-op `@profile` decorator to the namespace while unit testing

```

# check for line_profiler or memory_profiler in the local scope, both
# are injected by their respective tools or they're absent
# if these tools aren't being used (in which case we need to substitute
# a dummy @profile decorator)
if 'line_profiler' not in dir() and 'profile' not in dir():
    def profile(func):
        return func

```

Having added the no-op decorator, we can now run our `pytest` successfully, as shown in [Example 2-21](#), along with our profilers—with no additional code changes.

Example 2-21. With the no-op decorator, we have working tests, and both of our profilers work correctly

```
$ pytest utility.py
=====
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.8.0, pluggy-0.12.0
rootdir: /home/ian/workspace/personal_projects/high_performance_python_book_2e/
           high-performance-python-2e/examples_ian/ian/ch02/noop_profile_decorator
plugins: cov-2.7.1
collected 1 item

utility.py .

=====
1 passed in 3.04 seconds =====

$ kernprof -l -v utility.py
Example call `some_fn(2)` == 4
...
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
20                      @profile
21                      def some_fn(useful_input):
22                          """An expensive function that...
23                          # artificial 'we're doing...
24                      1    1001345.0 1001345.0    100.0    time.sleep(1)
25                      1        7.0       7.0     0.0    return useful_input ** 2

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

Line #      Mem usage      Increment  Line Contents
=====
20  48.277 MiB  48.277 MiB  @profile
21                      def some_fn(useful_input):
22                          """An expensive function that we wish to...
23                          # artificial 'we're doing something clever...
24  48.277 MiB    0.000 MiB    time.sleep(1)
25  48.277 MiB    0.000 MiB    return useful_input ** 2
```

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. This means that a cool CPU may run the same block of code faster than a hot CPU. 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. 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.

Lists and Tuples

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

¹ $O(1)$ 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).

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.



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.

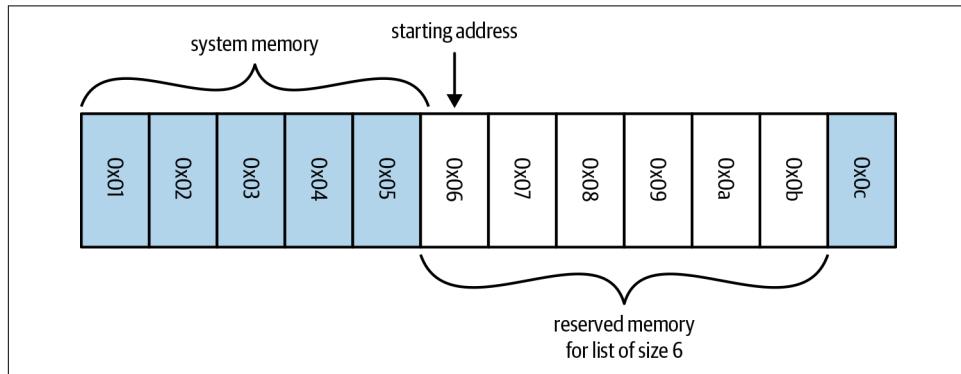


Figure 3-1. Example of system memory layout for an array of size 6

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

² 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!

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.



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]
...:
30.1 ns ± 0.996 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

>>> %%timeit l = list(range(10_000_000))
...:     l[100_000]
...:
28.9 ns ± 0.894 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

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?” on page 83](#)), which are a fundamental data structure powering [Chapter 4](#), 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 an impossible step to take from the constant-time lookups we saw before, but sometimes it is the best option (especially since search algorithms are more 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.



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 converting your data to a dictionary and then doing a 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 value in the haystack
    # 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 needle (and vice versa for the
        # value at j), we don't need to use absolute values here
        if haystack[i] - needle > needle - haystack[j]:
            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 because we inserted new elements
# with bisect.insort
print(important_numbers)
# > [14, 265, 496, 661, 683, 734, 881, 892, 973, 992]

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

closest_index = find_closest(important_numbers, 500)
print(f"Closest value to 500: {important_numbers[closest_index]}")
# > Closest value to 500: 496

closest_index = find_closest(important_numbers, 1100)
print(f"Closest value to 1100: {important_numbers[closest_index]}")
# > 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 within them cannot be changed after they have been created.
- 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 [Chapter 6](#), 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 $N + 1$ 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 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](#).³

³ The code responsible for this overallocation can be seen in the Python source code in [Objects/listobject.c:list_resize](#).

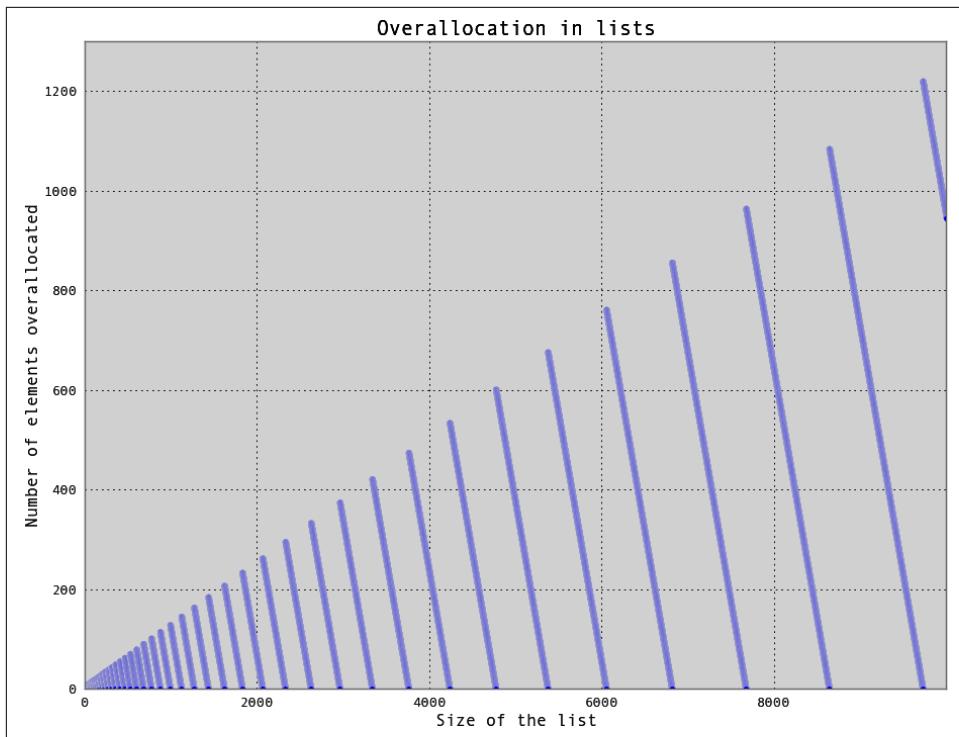


Figure 3-2. Graph showing how many extra elements are being allocated to a list of a particular size. For example, if you create a list with 8,000 elements using `append`, Python will allocate space for about 8,600 elements, overallocating 600 elements!

Example 3-5. List allocation equation in Python 3.7

```
M = (N >> 3) + (3 if N < 9 else 6)
```

N	0	1-4	5-8	9-16	17-25	26-35	36-46	...	991-1120
M	0	4	8	16	25	35	46	...	1120

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.

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](#).

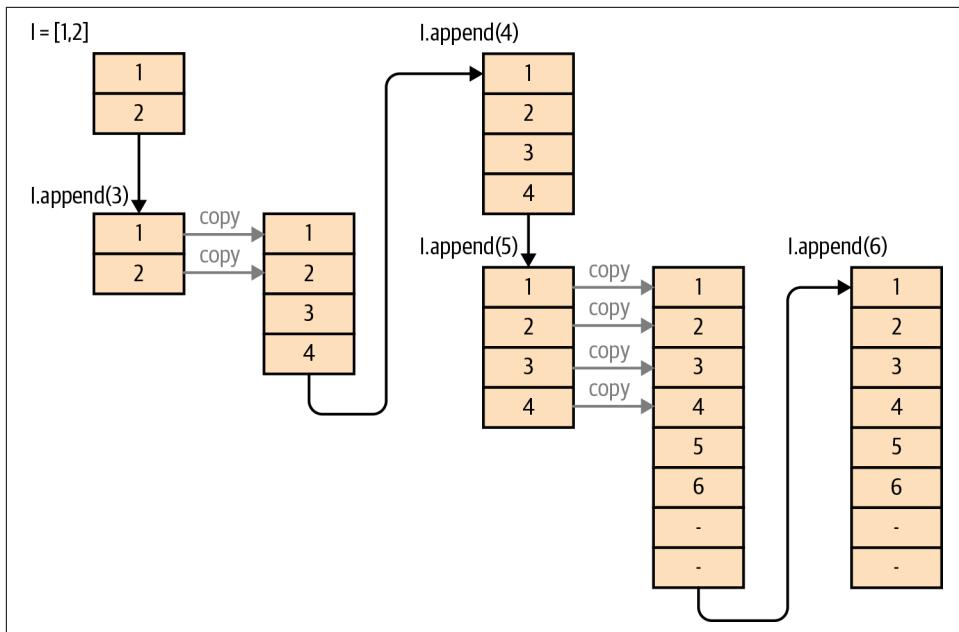


Figure 3-3. Example of how a list is mutated on multiple appends

Example 3-6. Resizing a list

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



This extra allocation happens on the first `append`. 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 even for 100,000 elements, we use 2.7× the memory by building the list with `appends` versus a list comprehension:

Example 3-7. Memory and time consequences of appends versus list comprehensions

```
>>> %memit [i*i for i in range(100_000)]
peak memory: 70.50 MiB, increment: 3.02 MiB

>>> %%memit l = []
... for i in range(100_000):
...     l.append(i * 2)
```

```

...
peak memory: 67.47 MiB, increment: 8.17 MiB

>>> %timeit [i*i for i in range(100_000)]
7.99 ms ± 219 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

>>> %%timeit l = []
... for i in range(100_000):
...     l.append(i * 2)
...
12.2 ms ± 184 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```

The overall runtime is also slower, because of the extra Python statements that must be run as well as the cost of reallocating memory. This effect becomes especially pronounced when you are maintaining many small lists or when keeping a particularly large list. If we are storing 1,000,000 lists, each containing 10 elements, we would suppose that 10,000,000 elements' worth of memory is being used. In actuality, however, up to 16,000,000 elements could have been allocated if the `append` operator was used to construct the list. Similarly, for a large list of 100,000,000 elements, we actually have 112,500,007 elements allocated!

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 assignment

```

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 allocate any extra 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 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 112,500,007 elements' worth of memory, while a tuple holding the same data will only ever use exactly 100,000,000 elements' worth of 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.

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 1–20, however, when they are no longer in use, the space isn't immediately given back to the system: up to 20,000 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.

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 3-8](#) shows that instantiating a list can be 5.1× slower than instantiating a tuple—which can add up quickly if this is done in a fast loop!

Example 3-8. Instantiation timings for lists versus tuples

```
>>> %timeit l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
95 ns ± 1.87 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

>>> %timeit t = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
12.5 ns ± 0.199 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
```

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 $O(1)$, avoiding 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 “[Aren’t Python Lists Good Enough?](#)” on page 115, 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.

Dictionaries and Sets

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.



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](#)” on page 88 for more details.

While we saw in the previous chapter that we are restricted to, at best, $O(\log n)$ lookup time on lists/tuples with no intrinsic order (through a search operation), 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?](#)” on page 83, 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 John Doe’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 = [
    ("John Doe", "555-555-5555"),
    ("Albert Einstein", "212-555-5555"),
]
print(f"John Doe's phone number is {find_phonenumber(phonebook, 'John Doe')}")
```



We could also do this by sorting the list and using the `bisect` module (from [Example 3-4](#)) in order to get $O(\log n)$ performance.

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.

¹ As we will discuss in “[Hash Functions and Entropy](#)” on page 88, 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.

Example 4-2. Phone book lookup with a dictionary

```
phonebook = {
    "John Doe": "555-555-5555",
    "Albert Einstein" : "212-555-5555",
}
print(f"John Doe's phone number is {phonebook['John Doe']}")
```

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.



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

def set_unique_names(phonebook):
    unique_names = set()
    for name, phonenumber in phonebook: ❸
        first_name, last_name = name.split(" ", 1)
        unique_names.add(first_name) ❹
    return len(unique_names)

phonebook = [
    ("John Doe", "555-555-5555"),
    ("Albert Einstein", "212-555-5555"),
```

```

        ("John Murphey", "202-555-5555"),
        ("Albert Rutherford", "647-555-5555"),
        ("Guido van Rossum", "301-555-5555"),
    ]
}

print("Number of unique names from set method:", set_unique_names(phonebook))
print("Number of unique names from list method:", list_unique_names(phonebook))

```

- ➊, ➋ We must go over all the items in our phone book, and thus this loop costs $O(n)$.
- ➌ 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.
- ➍ 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_names`, which starts out as empty and then grows, in the worst case, when all names are unique, to be the size of phone book. 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 $O(n^2)$.

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 7,412 unique first names, we see how drastic the difference between $O(n)$ and $O(n^2)$ can be:

```

>>> %timeit list_unique_names(large_phonebook)
1.13 s ± 26.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

>>> %timeit set_unique_names(large_phonebook)
4.48 ms ± 177 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```

In other words, the set algorithm gave us a $252\times$ speedup! In addition, as the size of the phonebook grows, the speed gains increase (we get a $557\times$ speedup with a phone book with 100,000 entries and 15,574 unique first names).

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.

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 smallest 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 $0b11111111$ (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.

² 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.

As an extra optimization, Python first 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 shows 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_SHIFT=5):
    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.,

³ The discussion that led to this improvement can be found at <https://oreil.ly/Pq7Lm>.

⁴ The value of 5 comes from the properties of a linear congruential generator (LCG), which is used in generating random numbers.

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](#)” on page 88, Python provides hashing functions for most of its types, so typically you won't have to provide one yourself except in extreme situations.

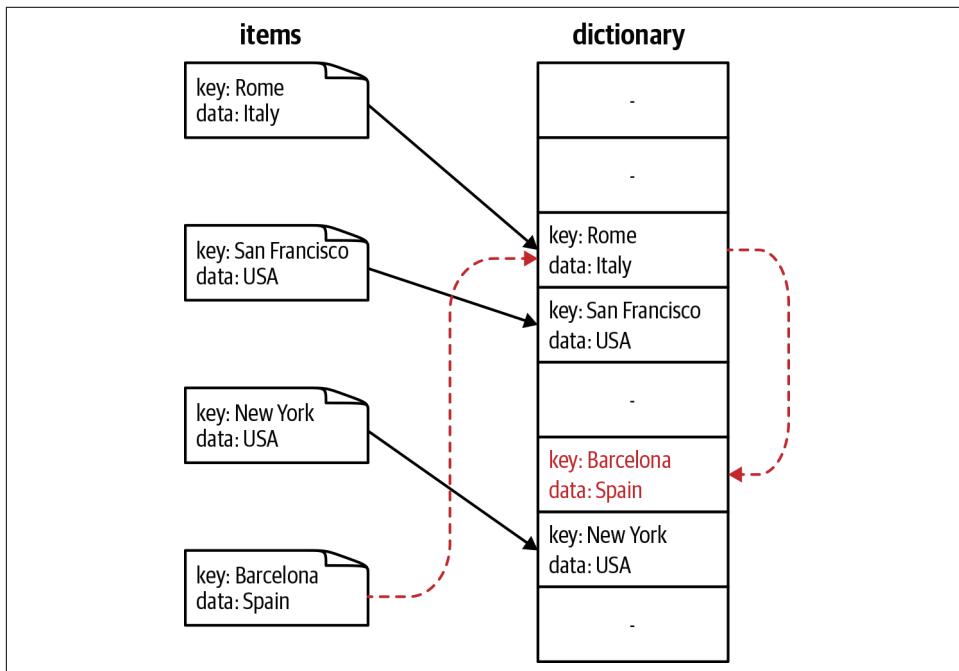


Figure 4-1. The resulting hash table from inserting with collisions

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 arbitrary values to cities
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](#)” on page 88 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 NULLs 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

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 elements), and it will resize by 3× if the

⁵ 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 $O(1)$.

dictionary is more than two-thirds full. So once the sixth item is being inserted into the originally empty dictionary, it will be resized to hold 18 elements. At this point, once the 13th element is inserted into the object, it will be resized to 39, then 81, and so on, always increasing the size by 3× (we will explain how to calculate a dictionary’s size in “[Hash Functions and Entropy](#)” on page 88). This gives the following possible sizes:

```
8; 18; 39; 81; 165; 333; 669; 1,341; 2,685; 5,373; 10,749; 21,501; 43,005; ...
```

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.**

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.⁶

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])
```

⁶ More information about this can be found at <https://oreil.ly/g4I5->.

```
set([<__main__.Point at 0x1099bfc90>, <__main__.Point at 0x1099bfbd0>])
>>> 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 == other.y
```

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:

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

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 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 16,384⁷ 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) = 0b1111111111111111`).

⁷ 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.

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.

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.



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 $\text{bin}(2048 - 1) = 0b111111111111$.

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 would be the one shown in [Example 4-6](#).

Example 4-6. Optimal two-letter hashing function

```
def twoletter_hash(key):
    offset = ord('a')
    k1, k2 = key
    return (ord(k2) - offset) + 26 * (ord(k1) - offset)
```

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) = 0b111111111111`).

Example 4-7 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 41.8× slowdown of lookups.

Example 4-7. 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 twoletter_hash
        """
        return ord(self[1]) + 26 * ord(self[0]) - 2619

baddict = set()
goodecict = set()
for i in string.ascii_lowercase:
    for j in string.ascii_lowercase:
        key = i + j
        baddict.add(BadHash(key))
        goodecict.add(GoodHash(key))

badtime = timeit.repeat(
    "key in baddict",
    setup = "from __main__ import baddict, BadHash; key = BadHash('zz')",
    repeat = 3,
    number = 1_000_000,
)
goodtime = timeit.repeat(
    "key in goodecict",
    setup = "from __main__ import goodecict, GoodHash; key = GoodHash('zz')",
    repeat = 3,
    number = 1_000_000,
)

print(f"Min lookup time for baddict: {min(badtime)}")
print(f"Min lookup time for goodecict: {min(goodtime)}")

# Results:
# Min lookup time for baddict: 17.719061855008476
# Min lookup time for goodecict: 0.42408075400453527
```

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 4-6](#) is ideal for a hash table of size 1,024. Why is it not ideal for smaller hash tables?

Dictionaries and Namespaces

Doing a lookup on a dictionary is fast; however, doing it unnecessarily will slow down your code, just as any extraneous lines will. One area where this surfaces is in Python's namespace management, which heavily uses dictionaries to do its lookups.

Whenever a variable, function, or module is invoked in Python, there is a hierarchy that determines where it looks for these objects. First, Python looks inside the `locals()` array, which has entries for all local variables. Python works hard to make local variable lookups fast, and this is the only part of the chain that doesn't require a dictionary lookup. If it doesn't exist there, the `globals()` dictionary is searched. Finally, if the object isn't found there, the `__builtin__` object is searched. It is important to note that while `locals()` and `globals()` are explicitly dictionaries and `__builtin__` is technically a module object, when searching `__builtin__` for a given property, we are just doing a dictionary lookup inside *its* `locals()` map (this is the case for all module objects and class objects!).

To make this clearer, let's look at a simple example of calling functions that are defined in different scopes ([Example 4-8](#)). We can disassemble the functions with the `dis` module ([Example 4-9](#)) to get a better understanding of how these namespace lookups are happening (see “[Using the dis Module to Examine CPython Bytecode](#)” on page 55).

Example 4-8. Namespace lookups

```
import math
from math import sin

def test1(x):
    """
    >>> %timeit test1(123_456)
    162 µs ± 3.82 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
    """
    res = 1
    for _ in range(1000):
        res += math.sin(x)
```

```

    return res

def test2(x):
    """
>>> %timeit test2(123_456)
124 µs ± 6.77 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
"""
    res = 1
    for _ in range(1000):
        res += sin(x)
    return res

def test3(x, sin=math.sin):
    """
>>> %timeit test3(123_456)
105 µs ± 3.35 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
"""
    res = 1
    for _ in range(1000):
        res += sin(x)
    return res

```

Example 4-9. Namespace lookups disassembled

```

>>> dis.dis(test1)
...cut...
      20 LOAD_GLOBAL              1 (math)
      22 LOAD_METHOD               2 (sin)
      24 LOAD_FAST                  0 (x)
      26 CALL_METHOD                 1
...cut..

>>> dis.dis(test2)
...cut...
      20 LOAD_GLOBAL              1 (sin)
      22 LOAD_FAST                  0 (x)
      24 CALL_FUNCTION                 1
...cut...

>>> dis.dis(test3)
...cut...
      20 LOAD_FAST                  1 (sin)
      22 LOAD_FAST                  0 (x)
      24 CALL_FUNCTION                 1
...cut...

```

The first function, `test1`, makes the call to `sin` by explicitly looking at the `math` library. This is also evident in the bytecode that is produced: first a reference to the `math` module must be loaded, and then we do an attribute lookup on this module until we finally have a reference to the `sin` function. **This is done through two dictionary lookups: one to find the `math` module and one to find the `sin` function within the module.**

On the other hand, `test2` explicitly imports the `sin` function from the `math` module, and the function is then directly accessible within the global namespace. This means we can avoid the lookup of the `math` module and the subsequent attribute lookup. However, we still must find the `sin` function within the global namespace. This is yet another reason to be explicit about what functions you are importing from a module. This practice not only makes code more readable, because the reader knows exactly what functionality is required from external sources, but it also simplifies changing the implementation of specific functions and generally speeds up code!

Finally, `test3` defines the `sin` function as a keyword argument, with its default value being a reference to the `sin` function within the `math` module. While we still do need to find a reference to this function within the module, this is necessary only when the `test3` function is first defined. After this, the reference to the `sin` function is stored within the function definition as a local variable in the form of a default keyword argument. As mentioned previously, local variables do not need a dictionary lookup to be found; they are stored in a very slim array that has very fast lookup times. Because of this, finding the function is quite fast!

While these effects are an interesting result of the way namespaces in Python are managed, `test3` is definitely not “Pythonic.” Luckily, these extra dictionary lookups start to degrade performance only when they are called a lot (i.e., in the innermost block of a very fast loop, such as in the Julia set example). With this in mind, a more readable solution would be to set a local variable with the global reference before the loop is started. We’ll still have to do the global lookup once whenever the function is called, but all the calls to that function in the loop will be made faster. This speaks to the fact that even minute slowdowns in code can be amplified if that code is being run millions of times. Even though a dictionary lookup itself may take only several hundred nanoseconds, if we are looping millions of times over this lookup, those nanoseconds can quickly add up.



A message about microbenchmarks: it may seem confusing that in [Example 4-8](#) we add in extra work with the `for` loop and the modification to the `res` variable. Originally, each of these functions simply had the relevant `return sin(x)` line and nothing else. As a result, we were also getting nanosecond runtimes and results that did not make any sense!

When we added a bigger workload within each function, as done through the loop and the modification of the `res` variable, we started seeing the results we expected. With a bigger workload inside the function, we can be more sure that we are not measuring overhead from the benchmarking/timing process. In general, when you are running benchmarks and have a difference in timing in the nanoseconds, it's important to sit back for a second and think through whether the experiment you are running is valid or whether you are measuring noise or unrelated timings as a result of instrumentation.

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.

Iterators and Generators

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 ①
        a, b = b, a+b
        num_items -= 1
```

- ➊ 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()
    332 ns ± 13.1 ns per loop (mean ± std. dev. of 7 runs, 1 loop each)

    >>> %memit test_fibonacci_list()
    peak memory: 492.82 MiB, increment: 441.75 MiB
    """
    for i in fibonacci_list(100_000):
        pass
```

```

def test_fibonacci_gen():
    """
    >>> %timeit test_fibonacci_gen()
    126 ms ± 905 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

    >>> %memit test_fibonacci_gen()
    peak memory: 51.13 MiB, increment: 0.00 MiB
    """
    for i in fibonacci_gen(100_000):
        pass

```

As we can see, the generator version is over twice as fast and requires no measurable memory as compared to the `fibonacci_list`'s 441 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



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_gen(100_000) if n % 3 == 0])
```

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 86 MB of data for no reason.² In fact, if we were doing this for a long enough

¹ 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.

² Calculated with `%memit len([n for n in fibonacci_gen(100_000) if n % 3 == 0])`.

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_gen(100_000) if n % 3 == 0)
```

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.



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 j  
    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 j <= 5000:  
        if j % 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:

<code>islice</code>	Allows slicing a potentially infinite generator
<code>chain</code>	Chains together multiple generators
<code>takewhile</code>	Adds a condition that will end a generator
<code>cycle</code>	Makes a finite generator infinite by constantly repeating it

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 itertools import count
from datetime import datetime

def read_data(filename):
    with open(filename) as fd:
        for line in fd:
            data = line.strip().split(',')
            timestamp, value = map(int, data)
            yield datetime.fromtimestamp(timestamp), value

def read_fake_data(filename):
    for timestamp in count():
        # We insert an anomalous data point approximately once a week
        if randint(0, 7 * 60 * 60 * 24 - 1) == 1:
            value = normalvariate(0, 1)
        else:
            value = 100
        yield datetime.fromtimestamp(timestamp), value
```

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[0].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.



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

def is_normal(data, threshold=1e-3):
    _, values = zip(*data)
    k2, p_value = normaltest(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_group)

data = read_data(filename)
anomaly_generator = filter_anomalous_data(data)
first_five_anomalies = islice(anomaly_generator, 5)

for data_anomaly in first_five_anomalies:
    start_date = data_anomaly[0][0]
    end_date = data_anomaly[-1][0]
    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 23:59:59
Anomaly from 1970-01-17 00:00:00 - 1970-01-17 23:59:59
Anomaly from 1970-01-18 00:00:00 - 1970-01-18 23:59:59
Anomaly from 1970-01-23 00:00:00 - 1970-01-23 23:59:59
Anomaly from 1970-01-29 00:00:00 - 1970-01-29 23:59:59
```

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 “[Memory Allocations and In-Place Operations](#)” on page 133 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 [Chapter 11](#), 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](#)” on page 101, 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.

Matrix and Vector Computation

Questions You'll Be Able to Answer After This Chapter

- What are the bottlenecks in vector calculations?
- What tools can I use to see how efficiently the CPU is doing my calculations?
- Why is `numpy` better at numerical calculations than pure Python?
- What are `cache-misses` and `page-faults`?
- How can I track the memory allocations in my code?

Regardless of what problem you are trying to solve on a computer, you will encounter vector computation at some point. Vector calculations are integral to how a computer works and how it tries to speed up runtimes of programs down at the silicon level—the only thing the computer knows how to do is operate on numbers, and knowing how to do several of those calculations at once will speed up your program.

In this chapter we try to unwrap some of the complexities of this problem by focusing on a relatively simple mathematical problem, solving the diffusion equation, and understanding what is happening at the CPU level. By understanding how different Python code affects the CPU and how to effectively probe these things, we can learn how to understand other problems as well.

We will start by introducing the problem and coming up with a quick solution using pure Python. After identifying some memory issues and trying to fix them using pure Python, we will introduce `numpy` and identify how and why it speeds up our code. Then we will start doing some algorithmic changes and specialize our code to solve the problem at hand. By removing some of the generality of the libraries we are using, we will yet again be able to gain more speed. Next, we introduce some extra modules

that will help facilitate this sort of process out in the field, and also explore a cautionary tale about optimizing before profiling.

Finally, we'll take a look at the Pandas library, which builds upon numpy by taking columns of homogeneous data and storing them in a table of heterogeneous types. Pandas has grown beyond using pure numpy types and now can mix its own missing-data-aware types with numpy datatypes. While Pandas is incredibly popular with scientific developers and data scientists, there's a lot of misinformation about ways to make it run quickly; we address some of these issues and give you tips for writing performant and supportable analysis code.

Introduction to the Problem

To explore matrix and vector computation in this chapter, we will repeatedly use the example of diffusion in fluids. Diffusion is one of the mechanisms that moves fluids and tries to make them uniformly mixed.



This section is meant to give a deeper understanding of the equations we will be solving throughout the chapter. It is not strictly necessary that you understand this section in order to approach the rest of the chapter. If you wish to skip this section, make sure to at least look at the algorithm in Examples 6-1 and 6-2 to understand the code we will be optimizing.

On the other hand, if you read this section and want even more explanation, read Chapter 17 of *Numerical Recipes*, 3rd Edition, by William Press et al. (Cambridge University Press).

In this section we will explore the mathematics behind the diffusion equation. This may seem complicated, but don't worry! We will quickly simplify this to make it more understandable. Also, it is important to note that while having a basic understanding of the final equation we are solving will be useful while reading this chapter, it is not completely necessary; the subsequent chapters will focus mainly on various formulations of the code, not the equation. However, having an understanding of the equations will help you gain intuition about ways of optimizing your code. This is true in general—understanding the motivation behind your code and the intricacies of the algorithm will give you deeper insight about possible methods of optimization.

One simple example of diffusion is dye in water: if you put several drops of dye into water at room temperature, the dye will slowly move out until it fully mixes with the water. Since we are not stirring the water, nor is it warm enough to create convection currents, diffusion will be the main process mixing the two liquids. When solving these equations numerically, we pick what we want the initial condition to look like

and are able to evolve the initial condition forward in time to see what it will look like at a later time (see [Figure 6-2](#)).

All this being said, the most important thing to know about diffusion for our purposes is its formulation. Stated as a partial differential equation in one dimension (1D), the diffusion equation is written as follows:

$$\frac{\partial}{\partial t} u(x, t) = D \cdot \frac{\partial^2}{\partial x^2} u(x, t)$$

In this formulation, u is the vector representing the quantities we are diffusing. For example, we could have a vector with values of 0 where there is only water, and of 1 where there is only dye (and values in between where there is mixing). In general, this will be a 2D or 3D matrix representing an actual area or volume of fluid. In this way, we could have u be a 3D matrix representing the fluid in a glass, and instead of simply doing the second derivative along the x direction, we'd have to take it over all axes. In addition, D is a physical value that represents properties of the fluid we are simulating. A large value of D represents a fluid that can diffuse very easily. For simplicity, we will set $D = 1$ for our code but still include it in the calculations.



The diffusion equation is also called the *heat equation*. In this case, u represents the temperature of a region, and D describes how well the material conducts heat. Solving the equation tells us how the heat is being transferred. So instead of solving for how a couple of drops of dye diffuse through water, we might be solving for how the heat generated by a CPU diffuses into a heat sink.

What we will do is take the diffusion equation, which is continuous in space and time, and approximate it using discrete volumes and discrete times. We will do so using Euler's method. *Euler's method* simply takes the derivative and writes it as a difference, such that

$$\frac{\partial}{\partial t} u(x, t) \approx \frac{u(x, t + dt) - u(x, t)}{dt}$$

where dt is now a fixed number. This fixed number represents the time step, or the resolution in time for which we wish to solve this equation. It can be thought of as the frame rate of the movie we are trying to make. As the frame rate goes up (or dt goes down), we get a clearer picture of what happens. In fact, as dt approaches zero, Euler's approximation becomes exact (note, however, that this exactness can be achieved only theoretically, since there is only finite precision on a computer and numerical errors will quickly dominate any results). We can thus rewrite this

equation to figure out what $u(x, t + dt)$ is, given $u(x, t)$. What this means for us is that we can start with some initial state ($u(x, 0)$), representing the glass of water just as we add a drop of dye into it) and churn through the mechanisms we've outlined to “evolve” that initial state and see what it will look like at future times ($u(x, dt)$). This type of problem is called an *initial value problem* or *Cauchy problem*. Doing a similar trick for the derivative in x using the finite differences approximation, we arrive at the final equation:

$$u(x, t + dt) = u(x, t) + dt * D * \frac{u(x + dx, t) + u(x - dx, t) - 2 \cdot u(x, t)}{dx^2}$$

Here, similar to how dt represents the frame rate, dx represents the resolution of the images—the smaller dx is, the smaller a region every cell in our matrix represents. For simplicity, we will set $D = 1$ and $dx = 1$. These two values become very important when doing proper physical simulations; however, since we are solving the diffusion equation for illustrative purposes, they are not important to us.

Using this equation, we can solve almost any diffusion problem. However, there are some considerations regarding this equation. First, we said before that the spatial index in u (i.e., the x parameter) will be represented as the indices into a matrix. What happens when we try to find the value at $x - dx$ when x is at the beginning of the matrix? This problem is called the *boundary condition*. You can have fixed boundary conditions that say “any value out of the bounds of my matrix will be set to 0” (or any other value). Alternatively, you can have periodic boundary conditions that say that the values will wrap around. (That is, if one of the dimensions of your matrix has length N , the value in that dimension at index -1 is the same as at $N - 1$, and the value at N is the same as at index 0 . In other words, if you are trying to access the value at index i , you will get the value at index $(i \% N)$.)

Another consideration is how we are going to store the multiple time components of u . We could have one matrix for every time value we do our calculation for. At minimum, it seems that we will need two matrices: one for the current state of the fluid and one for the next state of the fluid. As we'll see, there are very drastic performance considerations for this particular question.

So what does it look like to solve this problem in practice? [Example 6-1](#) contains some pseudocode to illustrate the way we can use our equation to solve the problem.

Example 6-1. Pseudocode for 1D diffusion

```
# Create the initial conditions
u = vector of length N
for i in range(N):
    u = 0 if there is water, 1 if there is dye
```

```

# Evolve the initial conditions
D = 1
t = 0
dt = 0.0001
while True:
    print(f"Current time is: {t}")
    unew = vector of size N

    # Update step for every cell
    for i in range(N):
        unew[i] = u[i] + D * dt * (u[(i+1)%N] + u[(i-1)%N] - 2 * u[i])
    # Move the updated solution into u
    u = unew

    visualize(u)

```

This code will take an initial condition of the dye in water and tell us what the system looks like at every 0.0001-second interval in the future. The results can be seen in **Figure 6-1**, where we evolve our very concentrated drop of dye (represented by the top-hat function) into the future. We can see how, far into the future, the dye becomes well mixed, to the point where everywhere has a similar concentration of the dye.

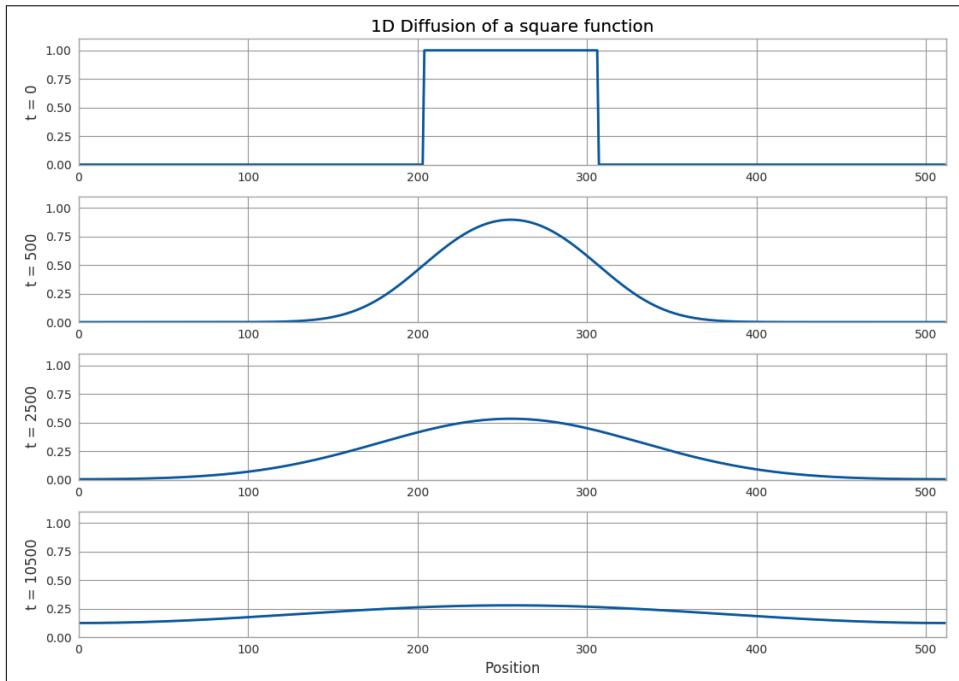


Figure 6-1. Example of 1D diffusion

For the purposes of this chapter, we will be solving the 2D version of the preceding equation. All this means is that instead of operating over a vector (or in other words, a matrix with one index), we will be operating over a 2D matrix. The only change to the equation (and thus to the subsequent code) is that we must now also take the second derivative in the y direction. This simply means that the original equation we were working with becomes the following:

$$\frac{\partial}{\partial t} u(x, y, t) = D \cdot \left(\frac{\partial^2}{\partial x^2} u(x, y, t) + \frac{\partial^2}{\partial y^2} u(x, y, t) \right)$$

This numerical diffusion equation in 2D translates to the pseudocode in [Example 6-2](#), using the same methods we used before.

Example 6-2. Algorithm for calculating 2D diffusion

```
for i in range(N):
    for j in range(M):
        unew[i][j] = u[i][j] + dt * (
            (u[(i + 1) % N][j] + u[(i - 1) % N][j] - 2 * u[i][j]) + # d^2 u / dx^2
            (u[i][(j + 1) % M] + u[i][(j - 1) % M] - 2 * u[i][j])    # d^2 u / dy^2
        )
```

We can now put all of this together and write the full Python 2D diffusion code that we will use as the basis for our benchmarks for the rest of this chapter. While the code looks more complicated, the results are similar to that of the 1D diffusion (as can be seen in [Figure 6-2](#)).

If you'd like to do some additional reading on the topics in this section, check out the [Wikipedia page on the diffusion equation](#) and [Chapter 7 of Numerical Methods for Complex Systems](#) by S. V. Gurevich.

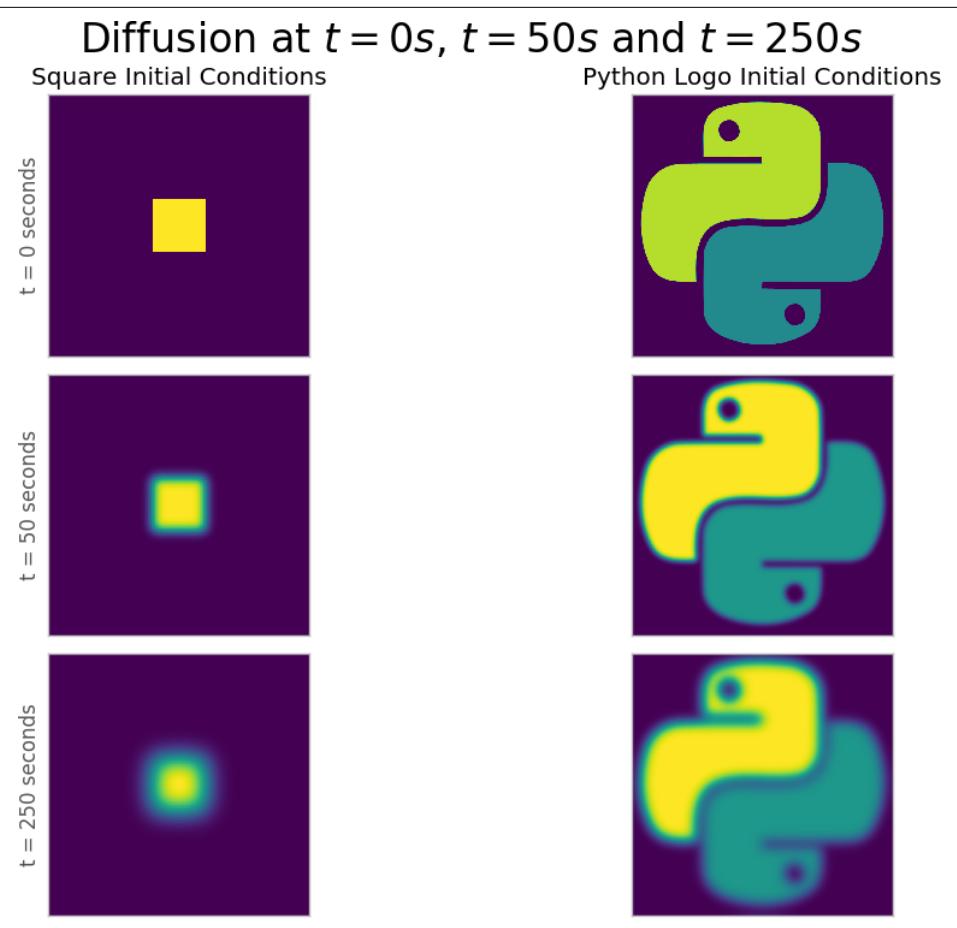


Figure 6-2. Example of 2D diffusion for two sets of initial conditions

Aren't Python Lists Good Enough?

Let's take our pseudocode from [Example 6-1](#) and formalize it so we can better analyze its runtime performance. The first step is to write out the evolution function that takes in the matrix and returns its evolved state. This is shown in [Example 6-3](#).

Example 6-3. Pure Python 2D diffusion

```
grid_shape = (640, 640)

def evolve(grid, dt, D=1.0):
    xmax, ymax = grid_shape
    new_grid = [[0.0] * ymax for x in range(xmax)]
```

```

for i in range(xmax):
    for j in range(ymax):
        grid_xx = (
            grid[(i + 1) % xmax][j] + grid[(i - 1) % xmax][j] - 2.0 * grid[i][j]
        )
        grid_yy = (
            grid[i][(j + 1) % ymax] + grid[i][(j - 1) % ymax] - 2.0 * grid[i][j]
        )
        new_grid[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt
return new_grid

```



Instead of preallocating the `new_grid` list, we could have built it up in the `for` loop by using `appends`. While this would have been noticeably faster than what we have written, the conclusions we draw are still applicable. We chose this method because it is more illustrative.

The global variable `grid_shape` designates how big a region we will simulate; and, as explained in “[Introduction to the Problem](#)” on page 110, we are using periodic boundary conditions (which is why we use modulo for the indices). To actually use this code, we must initialize a grid and call `evolve` on it. The code in [Example 6-4](#) is a very generic initialization procedure that will be reused throughout the chapter (its performance characteristics will not be analyzed since it must run only once, as opposed to the `evolve` function, which is called repeatedly).

Example 6-4. Pure Python 2D diffusion initialization

```

def run_experiment(num_iterations):
    # Setting up initial conditions ①
    xmax, ymax = grid_shape
    grid = [[0.0] * ymax for x in range(xmax)]

    # These initial conditions are simulating a drop of dye in the middle of our
    # simulated region
    block_low = int(grid_shape[0] * 0.4)
    block_high = int(grid_shape[0] * 0.5)
    for i in range(block_low, block_high):
        for j in range(block_low, block_high):
            grid[i][j] = 0.005

    # Evolve the initial conditions
    start = time.time()
    for i in range(num_iterations):
        grid = evolve(grid, 0.1)
    return time.time() - start

```

- ① The initial conditions used here are the same as in the square example in [Figure 6-2](#).

The values for `dt` and `grid` elements have been chosen to be sufficiently small that the algorithm is stable. See *Numerical Recipes* for a more in-depth treatment of this algorithm's convergence characteristics.

Problems with Allocating Too Much

By using `line_profiler` on the pure Python evolution function, we can start to unravel what is contributing to a possibly slow runtime. Looking at the profiler output in [Example 6-5](#), we see that most of the time in the function is spent doing the derivative calculation and updating the grid.¹ This is what we want, since this is a purely CPU-bound problem—any time not spent on solving the CPU-bound problem is an obvious place for optimization.

Example 6-5. Pure Python 2D diffusion profiling

```
$ kernprof -lv diffusion_python.py
Wrote profile results to diffusion_python.py.lprof
Timer unit: 1e-06 s

Total time: 787.161 s
File: diffusion_python.py
Function: evolve at line 12

Line #      Hits            Time  Per Hit   % Time  Line Contents
=====
12          500        843.0     1.7    0.0      @profile
13                      xmax, ymax = grid_shape  ①
14      500    24764794.0   49529.6   3.1      new_grid = [[0.0 for x in ...
15      500    208683.0      0.7    0.0      for i in range(xmax):  ②
16          320500           0.6    16.4
17 205120000  128928913.0      0.6    16.4      for j in range(ymax):
18 204800000  222422192.0      1.1    28.3      grid_xx = ...
19 204800000  228660607.0      1.1    29.0      grid_yy = ...
20 204800000  182174957.0      0.9    23.1      new_grid[i][j] = ...
21          500        331.0     0.7    0.0      return new_grid  ③
```

- ① This statement takes such a long time per hit because `grid_shape` must be retrieved from the local namespace (see “[Dictionaries and Namespaces](#)” on page [92](#) for more information).

¹ This is the code from [Example 6-3](#), truncated to fit within the page margins. Recall that `kernprof` requires functions to be decorated with `@profile` in order to be profiled (see “[Using `line_profiler` for Line-by-Line Measurements](#)” on page [40](#)).

- ② This line has 320,500 hits associated with it, because the grid we operated over had `xmax = 640` and we ran the function 500 times. The calculation is $(640 + 1) * 500$, where the extra one evaluation is from the termination of the loop.
- ③ This line has 500 hits associated with it, which informs us that the function was profiled over 500 runs.

It's interesting to see the big difference in the `Per Hit` and `% Time` fields for line 15, where we allocate the new grid. This difference occurs because, while the line itself is quite slow (the `Per Hit` field shows that each run takes 0.0495 seconds, much slower than all other lines), it isn't called as often as other lines inside the loop. If we were to reduce the size of the grid and do more iterations (i.e., reduce the number of iterations of the loop but increase the number of times we call the function), we would see the `% Time` of this line increase and quickly dominate the runtime.

This is a waste, because the properties of `new_grid` do not change—no matter what values we send to `evolve`, the `new_grid` list will always be the same shape and size and contain the same values. A simple optimization would be to allocate this list once and simply reuse it. This way, we need to run this code only once, no matter the size of the grid or the number of iterations. This sort of optimization is similar to moving repetitive code outside a fast loop:

```
from math import sin

def loop_slow(num_iterations):
    """
    >>> %timeit loop_slow(int(1e4))
    1.68 ms ± 61.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
    """
    result = 0
    for i in range(num_iterations):
        result += i * sin(num_iterations) ①
    return result

def loop_fast(num_iterations):
    """
    >>> %timeit loop_fast(int(1e4))
    551 µs ± 23.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
    """
    result = 0
    factor = sin(num_iterations)
    for i in range(num_iterations):
        result += i
    return result * factor
```

- ① The value of `sin(num_iterations)` doesn't change throughout the loop, so there is no use recalculating it every time.

We can do a similar transformation to our diffusion code, as illustrated in [Example 6-6](#). In this case, we would want to instantiate `new_grid` in [Example 6-4](#) and send it in to our `evolve` function. That function will do the same as it did before: read the `grid` list and write to the `new_grid` list. Then we can simply swap `new_grid` with `grid` and continue again.

Example 6-6. Pure Python 2D diffusion after reducing memory allocations

```
def evolve(grid, dt, out, D=1.0):
    xmax, ymax = grid_shape
    for i in range(xmax):
        for j in range(ymax):
            grid_xx = (
                grid[(i + 1) % xmax][j] + grid[(i - 1) % xmax][j] - 2.0 * grid[i][j]
            )
            grid_yy = (
                grid[i][(j + 1) % ymax] + grid[i][(j - 1) % ymax] - 2.0 * grid[i][j]
            )
            out[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt

def run_experiment(num_iterations):
    # Setting up initial conditions
    xmax, ymax = grid_shape
    next_grid = [[0.0] * ymax for x in range(xmax)]
    grid = [[0.0] * ymax for x in range(xmax)]

    block_low = int(grid_shape[0] * 0.4)
    block_high = int(grid_shape[0] * 0.5)
    for i in range(block_low, block_high):
        for j in range(block_low, block_high):
            grid[i][j] = 0.005

    start = time.time()
    for i in range(num_iterations):
        # evolve modifies grid and next_grid in-place
        evolve(grid, 0.1, next_grid)
        grid, next_grid = next_grid, grid
    return time.time() - start
```

We can see from the line profile of the modified version of the code in [Example 6-7](#) that this small change has given us a 31.25% speedup.² This leads us to a conclusion similar to the one made during our discussion of `append` operations on lists (see “[Lists as Dynamic Arrays](#)” on page 72): memory allocations are not cheap. Every time

² The code profiled in [Example 6-7](#) is the code from [Example 6-6](#); it has been truncated to fit within the page margins.

we request memory to store a variable or a list, Python must take its time to talk to the operating system in order to allocate the new space, and then we must iterate over the newly allocated space to initialize it to some value.

Whenever possible, reusing space that has already been allocated will give performance speedups. However, be careful when implementing these changes. While the speedups can be substantial, as always you should profile to make sure you are achieving the results you want and are not simply polluting your code base.

Example 6-7. Line profiling Python diffusion after reducing allocations

```
$ kernprof -lv diffusion_python_memory.py
Wrote profile results to diffusion_python_memory.py.lprof
Timer unit: 1e-06 s

Total time: 541.138 s
File: diffusion_python_memory.py
Function: evolve at line 12

Line #      Hits            Time  Per Hit   % Time  Line Contents
=====
12          500        503.0     1.0      0.0  @profile
13          320500    131498.0    0.4      0.0  def evolve(grid, dt, out, D=1.0):
14      205120000    81105090.0    0.4     15.0      xmax, ymax = grid_shape
15      204800000    166271837.0    0.8     30.7      for i in range(xmax):
16      204800000    169216352.0    0.8     31.3          for j in range(ymax):
17      204800000    124412452.0    0.6     23.0              grid_xx = ...
18          320500    124412452.0    0.6     23.0              grid_yy = ...
19          320500    124412452.0    0.6     23.0              out[i][j] = ...
```

Memory Fragmentation

The Python code we wrote in [Example 6-6](#) still has a problem that is at the heart of using Python for these sorts of vectorized operations: Python doesn't natively support vectorization. There are two reasons for this: Python lists store pointers to the actual data, and Python bytecode is not optimized for vectorization, so `for` loops cannot predict when using vectorization would be beneficial.

The fact that Python lists store *pointers* means that, instead of actually holding the data we care about, lists store locations where that data can be found. For most uses, this is good because it allows us to store whatever type of data we like inside a list. However, when it comes to vector and matrix operations, this is a source of a lot of performance degradation.

The degradation occurs because every time we want to fetch an element from the `grid` matrix, we must do multiple lookups. For example, doing `grid[5][2]` requires us to first do a list lookup for index 5 on the list `grid`. This will return a pointer to

where the data at that location is stored. Then we need to do another list lookup on this returned object, for the element at index 2. Once we have this reference, we have the location where the actual data is stored.



Rather than creating a grid as a list-of-lists (`grid[x][y]`), how would you create a grid indexed by a tuple (`grid[(x, y)]`)? How would this affect the performance of the code?

The overhead for one such lookup is not big and can be, in most cases, disregarded. However, if the data we wanted was located in one contiguous block in memory, we could move *all* of the data in one operation instead of needing two operations for each element. This is one of the major points with data fragmentation: when your data is fragmented, you must move each piece over individually instead of moving the entire block over. This means you are invoking more memory transfer overhead, and you are forcing the CPU to wait while data is being transferred. We will see with `perf` just how important this is when looking at the `cache-misses`.

This problem of getting the right data to the CPU when it is needed is related to the *von Neumann bottleneck*. This refers to the limited bandwidth that exists between the memory and the CPU as a result of the tiered memory architecture that modern computers use. If we could move data infinitely fast, we would not need any cache, since the CPU could retrieve any data it needed instantly. This would be a state in which the bottleneck is nonexistent.

Since we can't move data infinitely fast, we must prefetch data from RAM and store it in smaller but faster CPU caches so that, hopefully, when the CPU needs a piece of data, it will be in a location that can be read from quickly. While this is a severely idealized way of looking at the architecture, we can still see some of the problems with it—how do we know what data will be needed in the future? The CPU does a good job with mechanisms called *branch prediction* and *pipelining*, which try to predict the next instruction and load the relevant portions of memory into the cache while still working on the current instruction. However, the best way to minimize the effects of the bottleneck is to be smart about how we allocate our memory and how we do our calculations over our data.

Probing how well memory is being moved to the CPU can be quite hard; however, in Linux the `perf` tool can be used to get amazing amounts of insight into how the CPU is dealing with the program being run.³ For example, we can run `perf` on the pure

³ On macOS you can get similar metrics by using Google's `gperftools` and the provided Instruments app. For Windows, we are told Visual Studio Profiler works well; however, we don't have experience with it.

Python code from [Example 6-6](#) and see just how efficiently the CPU is running our code.

The results are shown in [Example 6-8](#). Note that the output in this example and the following `perf` examples has been truncated to fit within the margins of the page. The removed data included variances for each measurement, indicating how much the values changed throughout multiple benchmarks. This is useful for seeing how much a measured value is dependent on the actual performance characteristics of the program versus other system properties, such as other running programs using system resources.

Example 6-8. Performance counters for pure Python 2D diffusion with reduced memory allocations (grid size: 640 × 640, 500 iterations)

```
$ perf stat -e cycles,instructions,\  
cache-references,cache-misses,branches,branch-misses,task-clock,faults,\  
minor-faults,cs,migrations python diffusion_python_memory.py
```

Performance counter stats for 'python diffusion_python_memory.py':

415,864,974,126	cycles	#	2.889 GHz
1,210,522,769,388	instructions	#	2.91 insn per cycle
656,345,027	cache-references	#	4.560 M/sec
349,562,390	cache-misses	#	53.259 % of all cache refs
251,537,944,600	branches	#	1747.583 M/sec
1,970,031,461	branch-misses	#	0.78% of all branches
143934.730837	task-clock (msec)	#	1.000 CPUs utilized
12,791	faults	#	0.089 K/sec
12,791	minor-faults	#	0.089 K/sec
117	cs	#	0.001 K/sec
6	migrations	#	0.000 K/sec

143.935522122 seconds time elapsed

Understanding `perf`

Let's take a second to understand the various performance metrics that `perf` is giving us and their connection to our code. The `task-clock` metric tells us how many clock cycles our task took. This is different from the total runtime, because if our program took one second to run but used two CPUs, then the `task-clock` would be 2000 (`task-clock` is generally in milliseconds). Conveniently, `perf` does the calculation for us and tells us, next to this metric, how many CPUs were utilized (where it says "XXXX CPUs utilized"). This number wouldn't be exactly 2 even when two CPUs are being used, though, because the process sometimes relied on other subsystems to do instructions for it (for example, when memory was allocated).

On the other hand, `instructions` tells us how many CPU instructions our code issued, and `cycles` tells us how many CPU cycles it took to run all of these instructions. The difference between these two numbers gives us an indication of how well our code is vectorizing and pipelining. With pipelining, the CPU is able to run the current operation while fetching and preparing the next one.

`cs` (representing “context switches”) and `CPU-migrations` tell us about how the program is halted in order to wait for a kernel operation to finish (such as I/O), to let other applications run, or to move execution to another CPU core. When a `context-switch` happens, the program’s execution is halted and another program is allowed to run instead. This is a *very* time-intensive task and is something we would like to minimize as much as possible, but we don’t have too much control over when this happens. The kernel delegates when programs are allowed to be switched out; however, we can do things to disincentivize the kernel from moving *our* program. In general, the kernel suspends a program when it is doing I/O (such as reading from memory, disk, or the network). As you’ll see in later chapters, we can use asynchronous routines to make sure that our program still uses the CPU even when waiting for I/O, which will let us keep running without being context-switched. In addition, we could set the `nice` value of our program to give our program priority and stop the kernel from context-switching it.⁴ Similarly, `CPU-migrations` happen when the program is halted and resumed on a different CPU than the one it was on before, in order to have all CPUs have the same level of utilization. This can be seen as an especially bad context switch, as not only is our program being temporarily halted, but we also lose whatever data we had in the L1 cache (recall that each CPU has its own L1 cache).

A `page-fault` (or just `fault`) is part of the modern Unix memory allocation scheme. When memory is allocated, the kernel doesn’t do much except give the program a reference to memory. Later, however, when the memory is first used, the operating system throws a minor page fault interrupt, which pauses the program that is being run and properly allocates the memory. This is called a *lazy allocation system*. While this method is an optimization over previous memory allocation systems, minor page faults are quite an expensive operation since most of the operations are done outside the scope of the program you are running. There is also a major page fault, which happens when the program requests data from a device (disk, network, etc.) that hasn’t been read yet. These are even more expensive operations: not only do they interrupt your program, but they also involve reading from whichever device the data lives on. This sort of page fault does not generally affect CPU-bound work; however, it will be a source of pain for any program that does disk or network reads/writes.⁵

⁴ This can be done by running the Python process through the `nice` utility (`nice -n -20 python program.py`). A nice value of -20 will make sure it yields execution as little as possible.

⁵ A good survey of the various faults can be found at <https://oreil.ly/12Beg>.

Once we have data in memory and we reference it, the data makes its way through the various tiers of memory (L1/L2/L3 memory—see “[Communications Layers](#)” on [page 8](#) for a discussion of this). Whenever we reference data that is in our cache, the `cache-references` metric increases. If we do not already have this data in the cache and need to fetch it from RAM, this counts as a `cache-miss`. We won’t get a cache miss if we are reading data we have read recently (that data will still be in the cache) or data that is located *near* data we have recently read (data is sent from RAM into the cache in chunks). Cache misses can be a source of slowdowns when it comes to CPU-bound work, since we need to wait to fetch the data from RAM *and* we interrupt the flow of our execution pipeline (more on this in a second). As a result, reading through an array in order will give many `cache-references` but not many `cache-misses` since if we read element i , element $i + 1$ will already be in cache. If, however, we read randomly through an array or otherwise don’t lay out our data in memory well, every read will require access to data that couldn’t possibly already be in cache. Later in this chapter we will discuss how to reduce this effect by optimizing the layout of data in memory.

A `branch` is a time in the code where the execution flow changes. Think of an `if...then` statement—depending on the result of the conditional, we will be executing either one section of code or another. This is essentially a branch in the execution of the code—the next instruction in the program could be one of two things. To optimize this, especially with regard to the pipeline, the CPU tries to guess which direction the branch will take and preload the relevant instructions. When this prediction is incorrect, we will get a `branch-miss`. Branch misses can be quite confusing and can result in many strange effects (for example, some loops will run substantially faster on sorted lists than on unsorted lists simply because there will be fewer branch misses).⁶

There are a lot more metrics that `perf` can keep track of, many of which are very specific to the CPU you are running the code on. You can run `perf list` to get the list of currently supported metrics on your system. For example, in the previous edition of this book, we ran on a machine that also supported `stalled-cycles-frontend` and `stalled-cycles-backend`, which tell us how many cycles our program was waiting for the frontend or backend of the pipeline to be filled. This can happen because of a cache miss, a mispredicted branch, or a resource conflict. The frontend of the pipeline is responsible for fetching the next instruction from memory and decoding it into a valid operation, while the backend is responsible for actually running the operation. These sorts of metrics can help tune the performance of a code to the optimizations and architecture choices of a particular CPU; however, unless you are always running on the same chip-set, it may be excessive to worry too much about them.

⁶ This effect is beautifully explained in this [Stack Overflow response](#).



If you would like a more thorough explanation of what is going on at the CPU level with the various performance metrics, check out Gurpur M. Prabhu's fantastic "[Computer Architecture Tutorial](#)." It deals with the problems at a very low level, which will give you a good understanding of what is going on under the hood when you run your code.

Making Decisions with perf's Output

With all this in mind, the performance metrics in [Example 6-8](#) are telling us that while running our code, the CPU had to reference the L1/L2 cache 656,345,027 times. Of those references, 349,562,390 (or 53.3%) were requests for data that wasn't in memory at the time and had to be retrieved. In addition, we can see that in each CPU cycle we are able to perform an average of 2.91 instructions, which tells us the total speed boost from pipelining, out-of-order execution, and hyperthreading (or any other CPU feature that lets you run more than one instruction per clock cycle).

Fragmentation increases the number of memory transfers to the CPU. Additionally, since you don't have multiple pieces of data ready in the CPU cache when a calculation is requested, you cannot vectorize the calculations. As explained in "[Communications Layers](#)" on page 8, vectorization of computations (or having the CPU do multiple computations at a time) can occur only if we can fill the CPU cache with all the relevant data. Since the bus can only move contiguous chunks of memory, this is possible only if the grid data is stored sequentially in RAM. Since a list stores pointers to data instead of the actual data, the actual values in the grid are scattered throughout memory and cannot be copied all at once.

We can alleviate this problem by using the `array` module instead of lists. These objects store data sequentially in memory, so that a slice of the `array` actually represents a continuous range in memory. However, this doesn't completely fix the problem—now we have data that is stored sequentially in memory, but Python still does not know how to vectorize our loops. What we would like is for any loop that does arithmetic on our array one element at a time to work on chunks of data, but as mentioned previously, there is no such bytecode optimization in Python (partly because of the extremely dynamic nature of the language).



Why doesn't having the data we want stored sequentially in memory automatically give us vectorization? If we look at the raw machine code that the CPU is running, vectorized operations (such as multiplying two arrays) use a different part of the CPU and different instructions than nonvectorized operations. For Python to use these special instructions, we must have a module that was created to use them. We will soon see how `numpy` gives us access to these specialized instructions.

Furthermore, because of implementation details, using the `array` type when creating lists of data that must be iterated on is actually *slower* than simply creating a `list`. This is because the `array` object stores a very low-level representation of the numbers it stores, and this must be converted into a Python-compatible version before being returned to the user. This extra overhead happens every time you index an `array` type. That implementation decision has made the `array` object less suitable for math and more suitable for storing fixed-type data more efficiently in memory.

Enter numpy

To deal with the fragmentation we found using `perf`, we must find a package that can efficiently vectorize operations. Luckily, `numpy` has all of the features we need—it stores data in contiguous chunks of memory and supports vectorized operations on its data. As a result, any arithmetic we do on `numpy` arrays happens in chunks without us having to explicitly loop over each element.⁷ Not only is it much easier to do matrix arithmetic this way, but it is also faster. Let's look at an example:

```
from array import array
import numpy

def norm_square_list(vector):
    """
    >>> vector = list(range(1_000_000))
    >>> %timeit norm_square_list(vector)
    85.5 ms ± 1.65 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
    """
    norm = 0
    for v in vector:
        norm += v * v
    return norm

def norm_square_list_comprehension(vector):
    """
    >>> vector = list(range(1_000_000))
    >>> %timeit norm_square_list_comprehension(vector)
    80.3 ms ± 1.37 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
    """
    return sum([v * v for v in vector])

def norm_square_array(vector):
    """
```

⁷ For an in-depth look at `numpy` over a variety of problems, check out *From Python to Numpy* by Nicolas P. Rougier.

```

>>> vector_array = array('l', range(1_000_000))
>>> %timeit norm_square_array(vector_array)
101 ms ± 4.69 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
"""

norm = 0
for v in vector:
    norm += v * v
return norm

def norm_square_numpy(vector):
"""
>>> vector_np = numpy.arange(1_000_000)
>>> %timeit norm_square_numpy(vector_np)
3.22 ms ± 136 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
"""
return numpy.sum(vector * vector) ①

def norm_square_numpy_dot(vector):
"""
>>> vector_np = numpy.arange(1_000_000)
>>> %timeit norm_square_numpy_dot(vector_np)
960 µs ± 41.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
"""
return numpy.dot(vector, vector) ②

```

- ① This creates two implied loops over `vector`, one to do the multiplication and one to do the sum. These loops are similar to the loops from `norm_square_list_comprehension`, but they are executed using `numpy`'s optimized numerical code.
- ② This is the preferred way of doing vector norms in `numpy` by using the vectorized `numpy.dot` operation. The less efficient `norm_square_numpy` code is provided for illustration.

The simpler `numpy` code runs 89× faster than `norm_square_list` and 83.65× faster than the “optimized” Python list comprehension. The difference in speed between the pure Python looping method and the list comprehension method shows the benefit of doing more calculation behind the scenes rather than explicitly in your Python code. By performing calculations using Python’s already-built machinery, we get the speed of the native C code that Python is built on. This is also partly the same reasoning behind why we have such a drastic speedup in the `numpy` code: instead of using the generalized list structure, we have a finely tuned and specially built object for dealing with arrays of numbers.

In addition to more lightweight and specialized machinery, the `numpy` object gives us memory locality and vectorized operations, which are incredibly important when dealing with numerical computations. The CPU is exceedingly fast, and most of the time simply getting it the data it needs faster is the best way to optimize code quickly. Running each function using the `perf` tool we looked at earlier shows that the `array` and pure Python functions take about 10^{12} instructions, while the `numpy` version takes about 10^9 instructions. In addition, the `array` and pure Python versions have around 53% cache misses, while `numpy` has around 20%.

In our `norm_square_numpy` code, when doing `vector * vector`, there is an *implied* loop that `numpy` will take care of. The implied loop is the same loop we have explicitly written out in the other examples: loop over all items in `vector`, multiplying each item by itself. However, since we tell `numpy` to do this instead of explicitly writing it out in Python code, `numpy` can take advantage of all the optimizations it wants. In the background, `numpy` has very optimized C code that has been made specifically to take advantage of any vectorization the CPU has enabled. In addition, `numpy` arrays are represented sequentially in memory as low-level numerical types, which gives them the same space requirements as `array` objects (from the `array` module).

As an added bonus, we can reformulate the problem as a dot product, which `numpy` supports. This gives us a single operation to calculate the value we want, as opposed to first taking the product of the two vectors and then summing them. As you can see in [Figure 6-3](#), this operation, `norm_numpy_dot`, outperforms all the others by quite a substantial margin—this is thanks to the specialization of the function, and because we don't need to store the intermediate value of `vector * vector` as we did in `norm_numpy`.

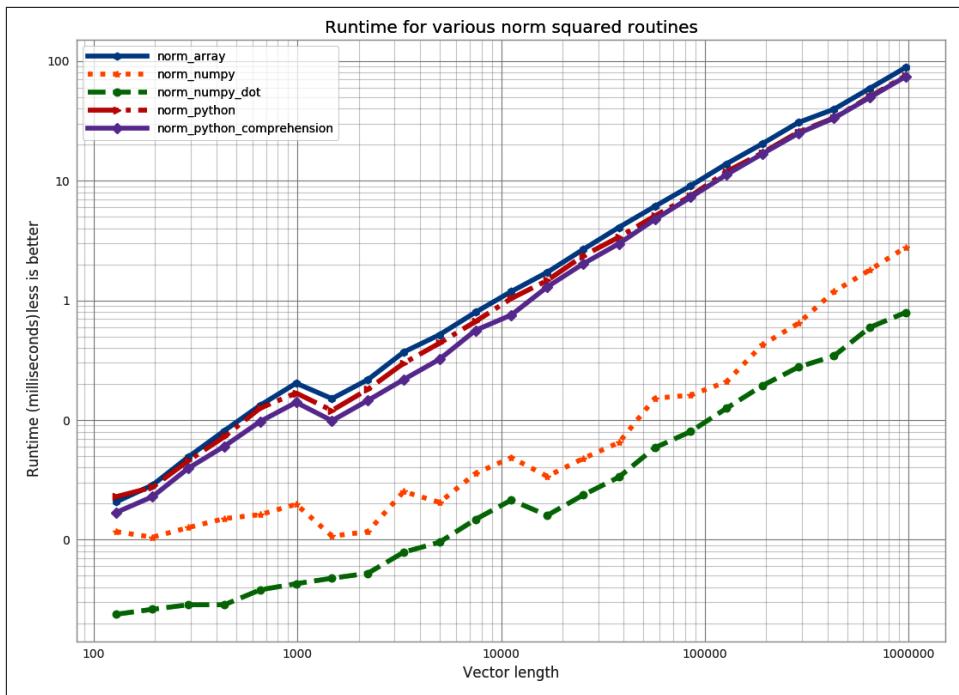


Figure 6-3. Runtimes for the various norm squared routines with vectors of different length

Applying numpy to the Diffusion Problem

Using what we've learned about numpy, we can easily adapt our pure Python code to be vectorized. The only new functionality we must introduce is numpy's `roll` function. This function does the same thing as our modulo-index trick, but it does so for an entire numpy array. In essence, it vectorizes this reindexing:

```
>>> import numpy as np
>>> np.roll([1,2,3,4], 1)
array([4, 1, 2, 3])

>>> np.roll([[1,2,3],[4,5,6]], 1, axis=1)
array([[3, 1, 2],
       [6, 4, 5]])
```

The `roll` function creates a new numpy array, which can be thought of as both good and bad. The downside is that we are taking time to allocate new space, which then needs to be filled with the appropriate data. On the other hand, once we have created this new rolled array, we will be able to vectorize operations on it quite quickly without suffering from cache misses from the CPU cache. This can substantially affect the speed of the actual calculation we must do on the grid. Later in this chapter

we will rewrite this so that we get the same benefit without having to constantly allocate more memory.

With this additional function we can rewrite the Python diffusion code from [Example 6-6](#) using simpler, and vectorized, `numpy` arrays. [Example 6-9](#) shows our initial `numpy` diffusion code.

Example 6-9. Initial numpy diffusion

```
from numpy import (zeros, roll)

grid_shape = (640, 640)

def laplacian(grid):
    return (
        roll(grid, +1, 0) +
        roll(grid, -1, 0) +
        roll(grid, +1, 1) +
        roll(grid, -1, 1) -
        4 * grid
    )

def evolve(grid, dt, D=1):
    return grid + dt * D * laplacian(grid)

def run_experiment(num_iterations):
    grid = zeros(grid_shape)

    block_low = int(grid_shape[0] * 0.4)
    block_high = int(grid_shape[0] * 0.5)
    grid[block_low:block_high, block_low:block_high] = 0.005

    start = time.time()
    for i in range(num_iterations):
        grid = evolve(grid, 0.1)
    return time.time() - start
```

Immediately we see that this code is much shorter. This is sometimes a good indication of performance: we are doing a lot of the heavy lifting outside the Python interpreter, and hopefully inside a module specially built for performance and for solving a particular problem (however, this should always be tested!). One of the assumptions here is that `numpy` is using better memory management to give the CPU the data it needs more quickly. However, since whether this happens or not relies on the actual implementation of `numpy`, let's profile our code to see whether our hypothesis is correct. [Example 6-10](#) shows the results.

Example 6-10. Performance counters for numpy 2D diffusion (grid size: 640 × 640, 500 iterations)

```
$ perf stat -e cycles,instructions,\  
cache-references,cache-misses,branches,branch-misses,task-clock,faults,\  
minor-faults,cs,migrations python diffusion_numpy.py
```

Performance counter stats for 'python diffusion_numpy.py':

8,432,416,866	cycles	#	2.886 GHz
7,114,758,602	instructions	#	0.84 insn per cycle
1,040,831,469	cache-references	#	356.176 M/sec
216,490,683	cache-misses	#	20.800 % of all cache refs
1,252,928,847	branches	#	428.756 M/sec
8,174,531	branch-misses	#	0.65% of all branches
2922.239426	task-clock (msec)	#	1.285 CPUs utilized
403,282	faults	#	0.138 M/sec
403,282	minor-faults	#	0.138 M/sec
96	cs	#	0.033 K/sec
5	migrations	#	0.002 K/sec

2.274377105 seconds time elapsed

This shows that the simple change to numpy has given us a 63.3× speedup over the pure Python implementation with reduced memory allocations ([Example 6-8](#)). How was this achieved?

First of all, we can thank the vectorization that numpy gives. Although the numpy version seems to be running fewer instructions per cycle, each instruction does much more work. That is to say, one vectorized instruction can multiply four (or more) numbers in an array together instead of requiring four independent multiplication instructions. Overall, this results in fewer total instructions being necessary to solve the same problem.

Several other factors contribute to the numpy version requiring a lower absolute number of instructions to solve the diffusion problem. One of them has to do with the full Python API being available when running the pure Python version, but not necessarily for the numpy version—for example, the pure Python grids can be appended to in pure Python but not in numpy. Even though we aren’t explicitly using this (or other) functionality, there is overhead in providing a system where it *could* be available. Since numpy can make the assumption that the data being stored is always going to be numbers, everything regarding the arrays can be optimized for operations over numbers. We will continue on the track of removing necessary functionality in favor of performance when we talk about Cython (see “[Cython](#)” on page 167), where it is even possible to remove list bounds checking to speed up list lookups.

Normally, the number of instructions doesn't necessarily correlate with performance—the program with fewer instructions may not issue them efficiently, or they may be slow instructions. However, we see that in addition to reducing the number of instructions, the `numpy` version has also reduced a large inefficiency: cache misses (20.8% cache misses instead of 53.3%). As explained in “[Memory Fragmentation](#)” on [page 120](#), cache misses slow down computations because the CPU must wait for data to be retrieved from slower memory instead of having the data immediately available in its cache. In fact, memory fragmentation is such a dominant factor in performance that if we disable vectorization in `numpy` but keep everything else the same,⁸ we still see a sizable speed increase compared to the pure Python version ([Example 6-11](#)).

Example 6-11. Performance counters for numpy 2D diffusion without vectorization (grid size: 640 × 640, 500 iterations)

```
$ perf stat -e cycles,instructions,\  
cache-references,cache-misses,branches,branch-misses,task-clock,faults,\  
minor-faults,cs,migrations python diffusion_numpy.py
```

Performance counter stats for 'python diffusion_numpy.py':

50,086,999,350	cycles	#	2.888 GHz
53,611,608,977	instructions	#	1.07 insn per cycle
1,131,742,674	cache-references	#	65.266 M/sec
322,483,897	cache-misses	#	28.494 % of all cache refs
4,001,923,035	branches	#	230.785 M/sec
6,211,101	branch-misses	#	0.16% of all branches
17340.464580	task-clock (msec)	#	1.000 CPUs utilized
403,193	faults	#	0.023 M/sec
403,193	minor-faults	#	0.023 M/sec
74	cs	#	0.004 K/sec
6	migrations	#	0.000 K/sec

17.339656586 seconds time elapsed

This shows us that the dominant factor in our 63.3× speedup when introducing `numpy` is not the vectorized instruction set but rather the memory locality and reduced memory fragmentation. In fact, we can see from the preceding experiment that vectorization accounts for only about 13% of that 63.3× speedup.⁹

⁸ We do this by compiling `numpy` with the `-fno-tree-vectorize` flag. For this experiment, we built `numpy` 1.17.3 with the following command: `$ OPT=' -fno-tree-vectorize' FOPT=' -fno-tree-vectorize' BLAS=None LAPACK=None ATLAS=None python setup.py build`.

⁹ This is contingent on what CPU is being used.

This realization that memory issues are the dominant factor in slowing down our code doesn't come as too much of a shock. Computers are very well designed to do exactly the calculations we are requesting them to do with this problem—multiplying and adding numbers together. The bottleneck is in getting those numbers to the CPU fast enough to see it do the calculations as fast as it can.

Memory Allocations and In-Place Operations

To optimize the memory-dominated effects, let's try using the same method we used in [Example 6-6](#) to reduce the number of allocations we make in our `numpy` code. Allocations are quite a bit worse than the cache misses we discussed previously. Instead of simply having to find the right data in RAM when it is not found in the cache, an allocation also must make a request to the operating system for an available chunk of data and then reserve it. The request to the operating system generates quite a lot more overhead than simply filling a cache—while filling a cache miss is a hardware routine that is optimized on the motherboard, allocating memory requires talking to another process, the kernel, in order to complete.

To remove the allocations in [Example 6-9](#), we will preallocate some scratch space at the beginning of the code and then use only in-place operations. In-place operations (such as `+=`, `*=`, etc.) reuse one of the inputs as their output. This means that we don't need to allocate space to store the result of the calculation.

To show this explicitly, we will look at how the `id` of a `numpy` array changes as we perform operations on it ([Example 6-12](#)). The `id` is a good way of tracking this for `numpy` arrays, since the `id` indicates which section of memory is being referenced. If two `numpy` arrays have the same `id`, they are referencing the same section of memory.

¹⁰

Example 6-12. In-place operations reducing memory allocations

```
>>> import numpy as np
>>> array1 = np.random.random((10,10))
>>> array2 = np.random.random((10,10))
>>> id(array1) ❶
140199765947424
>>> array1 += array2
>>> id(array1) ❷
140199765947424
>>> array1 = array1 + array2
```

¹⁰ This is not strictly true, since two `numpy` arrays can reference the same section of memory but use different striding information to represent the same data in different ways. These two `numpy` arrays will have different `ids`. There are many subtleties to the `id` structure of `numpy` arrays that are outside the scope of this discussion.

```
>>> id(array1)
140199765969792 ❸
```

- ❶, ❷ These two `ids` are the same, since we are doing an in-place operation. This means that the memory address of `array1` does not change; we are simply changing the data contained within it.
- ❸ Here, the memory address has changed. When doing `array1 + array2`, a new memory address is allocated and filled with the result of the computation. This does have benefits, however, for when the original data needs to be preserved (i.e., `array3 = array1 + array2` allows you to keep using `array1` and `array2`, while in-place operations destroy some of the original data).

Furthermore, we can see an expected slowdown from the non-in-place operation. In [Example 6-13](#), we see that using in-place operations gives us a 27% speedup for an array of 100×100 elements. This margin will become larger as the arrays grow, since the memory allocations become more strenuous. However, it is important to note that this effect happens only when the array sizes are bigger than the CPU cache! When the arrays are smaller and the two inputs and the output can all fit into cache, the out-of-place operation is faster because it can benefit from vectorization.

Example 6-13. Runtime differences between in-place and out-of-place operations

```
>>> import numpy as np

>>> %%timeit array1, array2 = np.random.random((2, 100, 100)) ❶ ❸
... array1 = array1 + array2
6.45 µs ± 53.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

>>> %%timeit array1, array2 = np.random.random((2, 100, 100)) ❶
... array1 += array2
5.06 µs ± 78.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

>>> %%timeit array1, array2 = np.random.random((2, 5, 5)) ❷
... array1 = array1 + array2
518 ns ± 4.88 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

>>> %%timeit array1, array2 = np.random.random((2, 5, 5)) ❷
... array1 += array2
1.18 µs ± 6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

- ❶ These arrays are too big to fit into the CPU cache, and the in-place operation is faster because of fewer allocations and cache misses.
- ❷ These arrays easily fit into cache, and we see the out-of-place operations as faster.

- ③ Note that we use `%%timeit` instead of `%timeit`, which allows us to specify code to set up the experiment that doesn't get timed.

The downside is that while rewriting our code from [Example 6-9](#) to use in-place operations is not very complicated, it does make the resulting code a bit harder to read. In [Example 6-14](#), we can see the results of this refactoring. We instantiate `grid` and `next_grid` vectors, and we constantly swap them with each other. `grid` is the current information we know about the system, and after running `evolve`, `next_grid` contains the updated information.

Example 6-14. Making most numpy operations in-place

```
def laplacian(grid, out):
    np.copyto(out, grid)
    out *= -4
    out += np.roll(grid, +1, 0)
    out += np.roll(grid, -1, 0)
    out += np.roll(grid, +1, 1)
    out += np.roll(grid, -1, 1)

def evolve(grid, dt, out, D=1):
    laplacian(grid, out)
    out *= D * dt
    out += grid

def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape)
    grid = np.zeros(grid_shape)

    block_low = int(grid_shape[0] * 0.4)
    block_high = int(grid_shape[0] * 0.5)
    grid[block_low:block_high, block_low:block_high] = 0.005

    start = time.time()
    for i in range(num_iterations):
        evolve(grid, 0.1, next_grid)
        grid, next_grid = next_grid, grid ①
    return time.time() - start
```

- ① Since the output of `evolve` gets stored in the output vector `next_grid`, we must swap these two variables so that, for the next iteration of the loop, `grid` has the most up-to-date information. This swap operation is quite cheap because only the references to the data are changed, not the data itself.



It is important to remember that since we want each operation to be in-place, whenever we do a vector operation, we must put it on its own line. This can make something as simple as $A = A * B + C$ become quite convoluted. Since Python has a heavy emphasis on readability, we should make sure that the changes we have made give us sufficient speedups to be justified.

Comparing the performance metrics from Examples 6-15 and 6-10, we see that removing the spurious allocations sped up our code by 30.9%. This speedup comes partly from a reduction in the number of cache misses but mostly from a reduction in minor faults. This comes from reducing the number of memory allocations necessary in the code by reusing already allocated space.

Minor faults are caused when a program accesses newly allocated space in memory. Since memory addresses are lazily allocated by the kernel, when you first access newly allocated data, the kernel pauses your execution while it makes sure that the required space exists and creates references to it for the program to use. This added machinery is quite expensive to run and can slow a program down substantially. On top of those additional operations that need to be run, we also lose any state that we had in cache and any possibility of doing instruction pipelining. In essence, we have to drop everything we're doing, including all associated optimizations, in order to go out and allocate some memory.

Example 6-15. Performance metrics for numpy with in-place memory operations (grid size: 640×640 , 500 iterations)

```
$ perf stat -e cycles,instructions,\  
cache-references,cache-misses,branches,branch-misses,task-clock,faults,\  
minor-faults,cs,migrations python diffusion_numpy_memory.py
```

Performance counter stats for 'python diffusion_numpy_memory.py':

6,880,906,446	cycles	#	2.886 GHz
5,848,134,537	instructions	#	0.85 insn per cycle
1,077,550,720	cache-references	#	452.000 M/sec
217,974,413	cache-misses	#	20.229 % of all cache refs
1,028,769,315	branches	#	431.538 M/sec
7,492,245	branch-misses	#	0.73% of all branches
2383.962679	task-clock (msec)	#	1.373 CPUs utilized
13,521	faults	#	0.006 M/sec
13,521	minor-faults	#	0.006 M/sec
100	cs	#	0.042 K/sec
8	migrations	#	0.003 K/sec

1.736322099 seconds time elapsed

Selective Optimizations: Finding What Needs to Be Fixed

Looking at the code from [Example 6-14](#), it seems like we have addressed most of the issues at hand: we have reduced the CPU burden by using `numpy`, and we have reduced the number of allocations necessary to solve the problem. However, there is always more investigation to be done. If we do a line profile on that code ([Example 6-16](#)), we see that the majority of our work is done within the `laplacian` function. In fact, 84% of the time that `evolve` takes to run is spent in `laplacian`.

Example 6-16. Line profiling shows that `laplacian` is taking too much time

```
Wrote profile results to diffusion_numpy_memory.py.lprof
Timer unit: 1e-06 s
```

```
Total time: 1.58502 s
File: diffusion_numpy_memory.py
Function: evolve at line 21

Line #      Hits            Time  Per Hit   % Time  Line Contents
=====
21                      @profile
22                      def evolve(grid, dt, out, D=1):
23      500    1327910.0    2655.8     83.8    laplacian(grid, out)
24      500    100733.0     201.5      6.4      out *= D * dt
25      500    156377.0     312.8      9.9      out += grid
```

There could be many reasons `laplacian` is so slow. However, there are two main high-level issues to consider. First, it looks like the calls to `np.roll` are allocating new vectors (we can verify this by looking at the documentation for the function). This means that even though we removed seven memory allocations in our previous refactoring, four outstanding allocations remain. Furthermore, `np.roll` is a very generalized function that has a lot of code to deal with special cases. Since we know exactly what we want to do (move just the first column of data to be the last in every dimension), we can rewrite this function to eliminate most of the spurious code. We could even merge the `np.roll` logic with the add operation that happens with the rolled data to make a very specialized `roll_add` function that does exactly what we want with the fewest number of allocations and the least extra logic.

[Example 6-17](#) shows what this refactoring would look like. All we need to do is create our new `roll_add` function and have `laplacian` use it. Since `numpy` supports fancy indexing, implementing such a function is just a matter of not jumbling up the indices. However, as stated earlier, while this code may be more performant, it is much less readable.



Notice the extra work that has gone into having an informative docstring for the function, in addition to full tests. When you are taking a route similar to this one, it is important to maintain the readability of the code, and these steps go a long way toward making sure that your code is always doing what it was intended to do and that future programmers can modify your code and know what things do and when things are not working.

Example 6-17. Creating our own `roll` function

```
import numpy as np

def roll_add(rollee, shift, axis, out):
    """
    Given a matrix, a rollee, and an output matrix, out, this function will
    perform the calculation:

    >>> out += np.roll(rollee, shift, axis=axis)

    This is done with the following assumptions:
    * rollee is 2D
    * shift will only ever be +1 or -1
    * axis will only ever be 0 or 1 (also implied by the first assumption)

    Using these assumptions, we are able to speed up this function by avoiding
    extra machinery that numpy uses to generalize the roll function and also
    by making this operation intrinsically in-place.
    """
    if shift == 1 and axis == 0:
        out[1:, :] += rollee[:-1, :]
        out[0, :] += rollee[-1, :]
    elif shift == -1 and axis == 0:
        out[:-1, :] += rollee[1:, :]
        out[-1, :] += rollee[0, :]
    elif shift == 1 and axis == 1:
        out[:, 1:] += rollee[:, :-1]
        out[:, 0] += rollee[:, -1]
    elif shift == -1 and axis == 1:
        out[:, :-1] += rollee[:, 1:]
        out[:, -1] += rollee[:, 0]

def test_roll_add():
    rollee = np.asarray([[1, 2], [3, 4]])
    for shift in (-1, +1):
        for axis in (0, 1):
            out = np.asarray([[6, 3], [9, 2]])
            expected_result = np.roll(rollee, shift, axis=axis) + out
            roll_add(rollee, shift, axis, out)
            assert np.all(expected_result == out)
```

```

def laplacian(grid, out):
    np.copyto(out, grid)
    out *= -4
    roll_add(grid, +1, 0, out)
    roll_add(grid, -1, 0, out)
    roll_add(grid, +1, 1, out)
    roll_add(grid, -1, 1, out)

```

If we look at the performance counters in [Example 6-18](#) for this rewrite, we see that while it is 22% faster than [Example 6-14](#), most of the counters are about the same. The major difference again is `cache-misses`, which is down 7 \times . This change also seems to have affected the throughput of instructions to the CPU, increasing the instructions per cycle from 0.85 to 0.99 (a 14% gain). Similarly, the faults went down 12.85%. This seems to be a result of first doing the rolls in place as well as reducing the amount of numpy machinery that needs to be in-place to do all of our desired computation. Instead of first rolling our array, doing other computation required by numpy in order to do bounds checking and error handling, and then adding the result, we are doing it all in one shot and not requiring the computer to refill the cache every time. This theme of trimming out unnecessary machinery in both numpy and Python in general will continue in “[Cython](#)” on page 167.

Example 6-18. Performance metrics for numpy with in-place memory operations and custom laplacian function (grid size: 640 × 640, 500 iterations)

```

$ perf stat -e cycles,instructions,\
    cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
    minor-faults,cs,migrations python diffusion_numpy_memory2.py

```

Performance counter stats for 'python diffusion_numpy_memory2.py':

5,971,464,515	cycles	#	2.888 GHz
5,893,131,049	instructions	#	0.99 insn per cycle
1,001,582,133	cache-references	#	484.398 M/sec
30,840,612	cache-misses	#	3.079 % of all cache refs
1,038,649,694	branches	#	502.325 M/sec
7,562,009	branch-misses	#	0.73% of all branches
2067.685884	task-clock (msec)	#	1.456 CPUs utilized
11,981	faults	#	0.006 M/sec
11,981	minor-faults	#	0.006 M/sec
95	cs	#	0.046 K/sec
3	migrations	#	0.001 K/sec

1.419869071 seconds time elapsed

numexpr: Making In-Place Operations Faster and Easier

One downfall of numpy's optimization of vector operations is that it occurs on only one operation at a time. That is to say, when we are doing the operation $A * B + C$ with numpy vectors, first the entire $A * B$ operation completes, and the data is stored in a temporary vector; then this new vector is added with C . The in-place version of the diffusion code in [Example 6-14](#) shows this quite explicitly.

However, many modules can help with this. numexpr is a module that can take an entire vector expression and compile it into very efficient code that is optimized to minimize cache misses and temporary space used. In addition, the expressions can utilize multiple CPU cores (see [Chapter 9](#) for more information) and take advantage of the specialized instructions your CPU may support in order to get even greater speedups. It even supports OpenMP, which parallels out operations across multiple cores on your machine.

It is very easy to change code to use numexpr: all that's required is to rewrite the expressions as strings with references to local variables. The expressions are compiled behind the scenes (and cached so that calls to the same expression don't incur the same cost of compilation) and run using optimized code. [Example 6-19](#) shows the simplicity of changing the evolve function to use numexpr. In this case, we chose to use the out parameter of the evaluate function so that numexpr doesn't allocate a new vector to which to return the result of the calculation.

Example 6-19. Using numexpr to further optimize large matrix operations

```
from numexpr import evaluate

def evolve(grid, dt, next_grid, D=1):
    laplacian(grid, next_grid)
    evaluate("next_grid * D * dt + grid", out=next_grid)
```

An important feature of numexpr is its consideration of CPU caches. It specifically moves data around so that the various CPU caches have the correct data in order to minimize cache misses. When we run perf on the updated code ([Example 6-20](#)), we see a speedup. However, if we look at the performance on a smaller, 256×256 grid, we see a decrease in speed (see [Table 6-2](#)). Why is this?

Example 6-20. Performance metrics for numpy with in-place memory operations, custom laplacian function, and numexpr (grid size: 640×640 , 500 iterations)

```
$ perf stat -e cycles,instructions,\
cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
minor-faults,cs,migrations python diffusion_numpy_memory2_numexpr.py
```

```
Performance counter stats for 'python diffusion_numpy_memory2_numexpr.py':
```

8,856,947,179	cycles	# 2.872 GHz
9,354,357,453	instructions	# 1.06 insn per cycle
1,077,518,384	cache-references	# 349.423 M/sec
59,407,830	cache-misses	# 5.513 % of all cache refs
1,018,525,317	branches	# 330.292 M/sec
11,941,430	branch-misses	# 1.17% of all branches
3083.709890	task-clock (msec)	# 1.991 CPUs utilized
15,820	faults	# 0.005 M/sec
15,820	minor-faults	# 0.005 M/sec
8,671	cs	# 0.003 M/sec
2,096	migrations	# 0.680 K/sec

```
1.548924090 seconds time elapsed
```

Much of the extra machinery we are bringing into our program with `numexpr` deals with cache considerations. When our grid size is small and all the data we need for our calculations fits in the cache, this extra machinery simply adds more instructions that don't help performance. In addition, compiling the vector operation that we encoded as a string adds a large overhead. When the total runtime of the program is small, this overhead can be quite noticeable. However, as we increase the grid size, we should expect to see `numexpr` utilize our cache better than native `numpy` does. In addition, `numexpr` utilizes multiple cores to do its calculation and tries to saturate each of the cores' caches. When the size of the grid is small, the extra overhead of managing the multiple cores overwhelms any possible increase in speed.

The particular computer we are running the code on has a 8,192 KB cache (Intel Core i7-7820HQ). Since we are operating on two arrays, one for input and one for output, we can easily do the calculation for the size of the grid that will fill up our cache. The number of grid elements we can store in total is $8,192\text{ KB} / 64\text{ bit} = 1,024,000$. Since we have two grids, this number is split between two objects (so each one can be at most $1,024,000 / 2 = 512,000$ elements). Finally, taking the square root of this number gives us the size of the grid that uses that many grid elements. All in all, this means that approximately two 2D arrays of size 715×715 would fill up the cache ($\sqrt{8192\text{KB} / 64\text{bit} / 2} = 715.5$). In practice, however, we do not get to fill up the cache ourselves (other programs will fill up parts of the cache), so realistically we can probably fit two 640×640 arrays. Looking at Tables 6-1 and 6-2, we see that when the grid size jumps from 512×512 to $1,024 \times 1,024$, the `numexpr` code starts to outperform pure `numpy`.

A Cautionary Tale: Verify “Optimizations” (scipy)

An important thing to take away from this chapter is the approach we took to every optimization: profile the code to get a sense of what is going on, come up with a possible solution to fix slow parts, and then profile to make sure the fix actually worked. Although this sounds straightforward, things can get complicated quickly, as we saw in how the performance of `numexpr` depended greatly on the size of the grid we were considering.

Of course, our proposed solutions don’t always work as expected. While writing the code for this chapter, we saw that the `laplacian` function was the slowest routine and hypothesized that the `scipy` routine would be considerably faster. This thinking came from the fact that Laplacians are a common operation in image analysis and probably have a very optimized library to speed up the calls. `scipy` has an image submodule, so we must be in luck!

The implementation was quite simple ([Example 6-21](#)) and required little thought about the intricacies of implementing the periodic boundary conditions (or “wrap” condition, as `scipy` calls it).

Example 6-21. Using `scipy`’s `laplace` filter

```
from scipy.ndimage.filters import laplace

def laplacian(grid, out):
    laplace(grid, out, mode="wrap")
```

Ease of implementation is quite important and definitely won this method some points before we considered performance. However, once we benchmarked the `scipy` code ([Example 6-22](#)), we had a revelation: this method offers no substantial speedup compared to the code it is based on ([Example 6-14](#)). In fact, as we increase the grid size, this method starts performing worse (see [Figure 6-4](#) at the end of the chapter).

Example 6-22. Performance metrics for diffusion with `scipy`’s `laplace` function (grid size: 640×640 , 500 iterations)

```
$ perf stat -e cycles,instructions,\
cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
minor-faults,cs,migrations python diffusion_scipy.py
```

Performance counter stats for 'python diffusion_scipy.py':

10,051,801,725	cycles	# 2.886 GHz
16,536,981,020	instructions	# 1.65 insn per cycle
1,554,557,564	cache-references	# 446.405 M/sec

```

126,627,735    cache-misses          #  8.146 % of all cache refs
2,673,416,633  branches             # 767.696 M/sec
9,626,762      branch-misses       #  0.36% of all branches
3482.391211   task-clock (msec)    #  1.228 CPUs utilized
14,013         faults              #  0.004 M/sec
14,013         minor-faults        #  0.004 M/sec
95            cs                  #  0.027 K/sec
5              migrations          #  0.001 K/sec

2.835263796 seconds time elapsed

```

Comparing the performance metrics of the `scipy` version of the code with those of our custom `laplacian` function ([Example 6-18](#)), we can start to get some indication as to why we aren't getting the speedup we were expecting from this rewrite.

The metric that stands out the most is `instructions`. This shows us that the `scipy` code is requesting that the CPU do more than double the amount of work as our custom `laplacian` code. Even though these instructions are more numerically optimized (as we can see with the higher `insn per cycle` count, which says how many instructions the CPU can do in one clock cycle), the extra optimization doesn't win out over the sheer number of added instructions.

This could be in part because the `scipy` code is written very generally so that it can process all sorts of inputs with different boundary conditions (which requires extra code and thus more instructions). We can see this, in fact, by the high number of `branches` that the `scipy` code requires. When code has many branches, it means that we run commands based on a condition (like having code inside an `if` statement). The problem is that we don't know whether we can evaluate an expression until we check the conditional, so vectorizing or pipelining isn't possible. The machinery of branch prediction helps with this, but it isn't perfect. This speaks more to the point of the speed of specialized code: if you don't need to constantly check what you need to do and instead know the specific problem at hand, you can solve it much more efficiently.

Lessons from Matrix Optimizations

Looking back on our optimizations, we seem to have taken two main routes: reducing the time taken to get data to the CPU and reducing the amount of work that the CPU had to do. Tables [6-1](#) and [6-2](#) compare of the results achieved by our various optimization efforts, for various dataset sizes, in relation to the original pure Python implementation.

[Figure 6-4](#) shows graphically how all these methods compared to one another. We can see three bands of performance that correspond to these two methods: the band along the bottom shows the small improvement made in relation to our pure Python

implementation by our first effort at reducing memory allocations; the middle band shows what happened when we used `numpy` and further reduced allocations; and the upper band illustrates the results achieved by reducing the work done by our process.

Table 6-1. Total runtime of all schemes for various grid sizes and 500 iterations of the `evolve` function

Method	256 x 256	512 x 512	1024 x 1024	2048 x 2048	4096 x 4096
Python	2.40s	10.43s	41.75s	168.82s	679.16s
Python + memory	2.26s	9.76s	38.85s	157.25s	632.76s
numpy	0.01s	0.09s	0.69s	3.77s	14.83s
numpy + memory	0.01s	0.07s	0.60s	3.80s	14.97s
numpy + memory + laplacian	0.01s	0.05s	0.48s	1.86s	7.50s
numpy + memory + laplacian + numexpr	0.02s	0.06s	0.41s	1.60s	6.45s
numpy + memory + scipy	0.05s	0.25s	1.15s	6.83s	91.43s

Table 6-2. Speedup compared to naive Python ([Example 6-3](#)) for all schemes and various grid sizes over 500 iterations of the `evolve` function

Method	256 x 256	512 x 512	1024 x 1024	2048 x 2048	4096 x 4096
Python	1.00x	1.00x	1.00x	1.00x	1.00x
Python + memory	1.06x	1.07x	1.07x	1.07x	1.07x
numpy	170.59x	116.16x	60.49x	44.80x	45.80x
numpy + memory	185.97x	140.10x	69.67x	44.43x	45.36x
numpy + memory + laplacian	203.66x	208.15x	86.41x	90.91x	90.53x
numpy + memory + laplacian + numexpr	97.41x	167.49x	102.38x	105.69x	105.25x
numpy + memory + scipy	52.27x	42.00x	36.44x	24.70x	7.43x

One important lesson to take away from this is that you should always take care of any administrative things the code must do during initialization. This may include allocating memory, or reading configuration from a file, or even precomputing values that will be needed throughout the lifetime of the program. This is important for two reasons. First, you are reducing the total number of times these tasks must be done by doing them once up front, and you know that you will be able to use those resources without too much penalty in the future. Second, you are not disrupting the flow of the program; this allows it to pipeline more efficiently and keep the caches filled with more pertinent data.

You also learned more about the importance of data locality and how important simply getting data to the CPU is. CPU caches can be quite complicated, and often it is best to allow the various mechanisms designed to optimize them take care of the issue. However, understanding what is happening and doing all that is possible to optimize the way memory is handled can make all the difference. For example, by

understanding how caches work, we are able to understand that the decrease in performance that leads to a saturated speedup no matter the grid size in [Figure 6-4](#) can probably be attributed to the L3 cache being filled up by our grid. When this happens, we stop benefiting from the tiered memory approach to solving the von Neumann bottleneck.

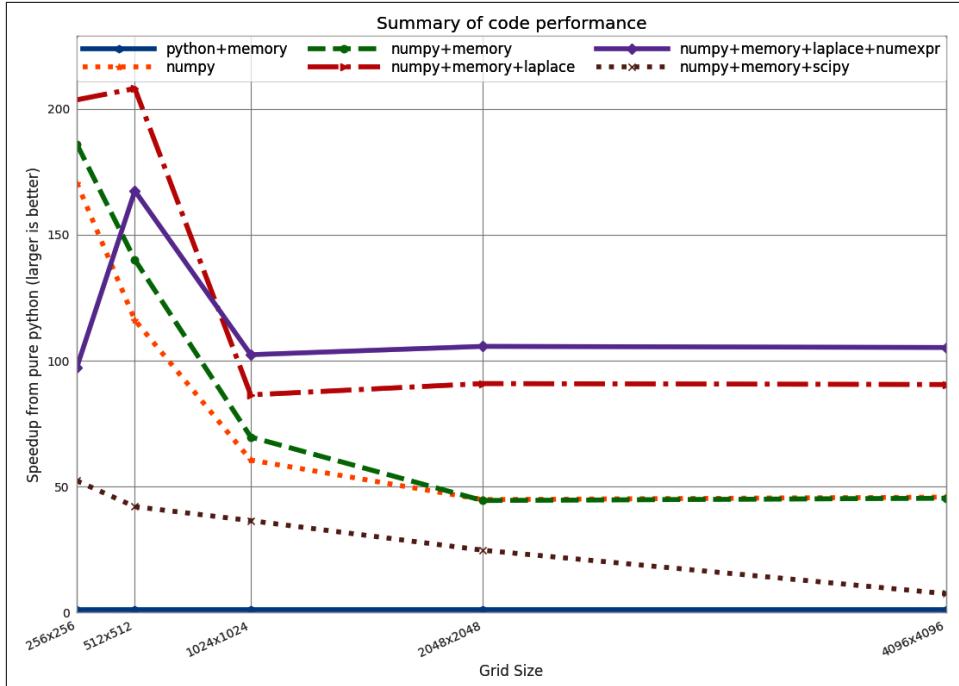


Figure 6-4. Summary of speedups from the methods attempted in this chapter

Another important lesson concerns the use of external libraries. Python is fantastic for its ease of use and readability, which allow you to write and debug code fast. However, tuning performance down to the external libraries is essential. These external libraries can be extremely fast, because they can be written in lower-level languages—but since they interface with Python, you can also still write code that uses them quickly.

Finally, we learned the importance of benchmarking everything and forming hypotheses about performance before running the experiment. By forming a hypothesis before running the benchmark, we are able to form conditions to tell us whether our optimization actually worked. Was this change able to speed up runtime? Did it reduce the number of allocations? Is the number of cache misses lower? Optimization can be an art at times because of the vast complexity of computer systems, and having a quantitative probe into what is actually happening can help enormously.

One last point about optimization is that a lot of care must be taken to make sure that the optimizations you make generalize to different computers (the assumptions you make and the results of benchmarks you do may be dependent on the architecture of the computer you are running on, how the modules you are using were compiled, etc.). In addition, when making these optimizations, it is incredibly important to consider other developers and how the changes will affect the readability of your code. For example, we realized that the solution we implemented in [Example 6-17](#) was potentially vague, so care was taken to make sure that the code was fully documented and tested to help not only us but also other people on the team.

Sometimes, however, your numerical algorithms also require quite a lot of data wrangling and manipulation that aren't just clear-cut mathematical operations. In these cases, Pandas is a very popular solution, and it has its own performance characteristics. We'll now do a deep dive into Pandas and understand how to better use it to write performant numerical code.

Pandas

Pandas is the de facto data manipulation tool in the scientific Python ecosystem for tabular data. It enables easy manipulation with Excel-like tables of heterogeneous datatypes, known as `DataFrames`, and has strong support for time-series operations. Both the public interface and the internal machinery have evolved a lot since 2008, and there's a lot of conflicting information in public forums on "fast ways to solve problems." In this section, we'll fix some misconceptions about common use cases of Pandas.

We'll review the internal model for Pandas, find out how to apply a function efficiently across a `DataFrame`, see why concatenating to a `DataFrame` repeatedly is a poor way to build up a result, and look at faster ways of handling strings.

Pandas's Internal Model

Pandas uses an in-memory, 2D, table-like data structure—if you have in mind an Excel sheet, you have a good initial mental model. Originally, Pandas focused on NumPy's `dtype` objects such as signed and unsigned numbers for each column. As the library evolved, it expanded beyond NumPy types and can now handle both Python strings and extension types (including nullable `Int64` objects—note the capital "I"—and IP addresses).

Operations on a `DataFrame` apply to all cells in a column (or all cells in a row if the `axis=1` parameter is used), all operations are executed eagerly, and there's no support for query planning. Operations on columns often generate temporary intermediate arrays, which consume RAM. The general advice is to expect a temporary memory usage envelope of up to three to five times your current usage when you're

manipulating your DataFrames. Typically, Pandas works well for datasets under 10 GB in size, assuming you have sufficient RAM for temporary results.

Operations can be single-threaded and may be limited by Python’s global interpreter lock (GIL). Increasingly, improved internal implementations are allowing the GIL to be disabled automatically, enabling parallelized operations. We’ll explore an approach to parallelization with Dask in “[Parallel Pandas with Dask](#)” on page 322.

Behind the scenes, columns of the same `dtype` are grouped together by a `BlockManager`. This piece of hidden machinery works to make row-wise operations on columns of the same datatype faster. It is one of the many hidden technical details that make the Pandas code base complex but make the high-level user-facing operations faster.¹¹

Performing operations on a subset of data from a single common block typically generates a view, while taking a slice of rows that cross blocks of different `dtypes` can cause a copy, which may be slower. One consequence is that while numeric columns directly reference their NumPy data, string columns reference a list of Python strings, and these individual strings are scattered in memory—this can lead to unexpected speed differences for numeric and string operations.

Behind the scenes, Pandas uses a mix of NumPy datatypes and its own extension datatypes. Examples from NumPy include `int8` (1 byte), `int64` (8 bytes—and note the lowercase “i”), `float16` (2 bytes), `float64` (8 bytes), and `bool` (1 byte). Additional types provided by Pandas include `categorical` and `datetimetz`. Externally, they appear to work similarly, but behind the scenes in the Pandas code base they cause a lot of type-specific Pandas code and duplication.



Whilst Pandas originally used only NumPy datatypes, it has evolved its own set of additional Pandas datatypes that understand missing data (NaN) behavior with three-valued logic. You must distinguish the NumPy `int64`, which is not NaN-aware, from the Pandas `Int64`, which uses two columns of data behind the scenes for the integers and for the NaN bit mask. Note that the NumPy `float64` is naturally NaN-aware.

¹¹ See the DataQuest blog post “[Tutorial: Using Pandas with Large Data Sets in Python](#)” for more details.

One side effect of using NumPy's datatypes has been that, while a `float` has a NaN (missing value) state, the same is not true for `int` and `bool` objects. If you introduce a NaN value into an `int` or `bool` Series in Pandas, your Series will be promoted to a `float`. Promoting `int` types to a `float` may reduce the numeric accuracy that can be represented in the same bits, and the smallest `float` is `float16`, which takes twice as many bytes as a `bool`.

The nullable `Int64` (note the capitalized "I") was introduced in version 0.24 as an extension type in Pandas. Internally, it uses a NumPy `int64` and a second Boolean array as a NaN-mask. Equivalents exist for `Int32` and `Int8`. As of Pandas version 1.0, there is also an equivalent nullable Boolean (with `dtype boolean` as opposed to the numpy `bool`, which isn't NaN-aware). A `StringDType` has been introduced that may in the future offer higher performance and less memory usage than the standard Python `str`, which is stored in a column of `object` `dtype`.

Applying a Function to Many Rows of Data

It is very common to apply functions to rows of data in Pandas. There's a selection of approaches, and the idiomatic Python approaches using loops are generally the slowest. We'll work through an example based on a real-world challenge, showing different ways of solving this problem and ending with a reflection on the trade-off between speed and maintainability.

Ordinary Least Squares (OLS) is a bread-and-butter method in data science for fitting a line to data. It solves for the slope and intercept in the $m * x + c$ equation, given some data. This can be incredibly useful when trying to understand the trend of the data: is it generally increasing, or is it decreasing?

An example of its use from our work is a research project for a telecommunications company where we want to analyze a set of potential user-behavior signals (e.g., marketing campaigns, demographics, and geographic behavior). The company has the number of hours a person spends on their cell phone every day, and its question is: is this person increasing or decreasing their usage, and how does this change over time?

One way to approach this problem is to take the company's large dataset of millions of users over years of data and break it into smaller windows of data (each window, for example, representing 14 days out of the years of data). For each window, we model the users' use through OLS and record whether they are increasing or decreasing their usage.

In the end, we have a sequence for each user showing whether, for a given 14-day period, their use was generally increasing or decreasing. However, to get there, we have to run OLS a massive number of times!

For one million users and two years of data, we might have 730 windows,¹² and thus 730,000,000 calls to OLS! To solve this problem practically, our OLS implementation should be fairly well tuned.

In order to understand the performance of various OLS implementations, we will generate some smaller but representative synthetic data to give us good indications of what to expect on the larger dataset. We'll generate data for 100,000 rows, each representing a synthetic user, and each containing 14 columns representing "hours used per day" for 14 days, as a continuous variable.

We'll draw from a Poisson distribution (with `lambda==60` as minutes) and divide by 60 to give us simulated hours of usage as continuous values. The true nature of the random data doesn't matter for this experiment; it is convenient to use a distribution that has a minimum value of 0 as this represents the real-world minimum. You can see a sample in [Example 6-23](#).

Example 6-23. A snippet of our data

	0	1	2	...	12	13
0	1.016667	0.883333	1.033333	...	1.016667	0.833333
1	1.033333	1.016667	0.833333	...	1.133333	0.883333
2	0.966667	1.083333	1.183333	...	1.000000	0.950000

¹² If we're going to use a sliding window, it might be possible to apply rolling window optimized functions such as `RollingOLS` from `statsmodels`.

In Figure 6-5, we see three rows of 14 days of synthetic data.

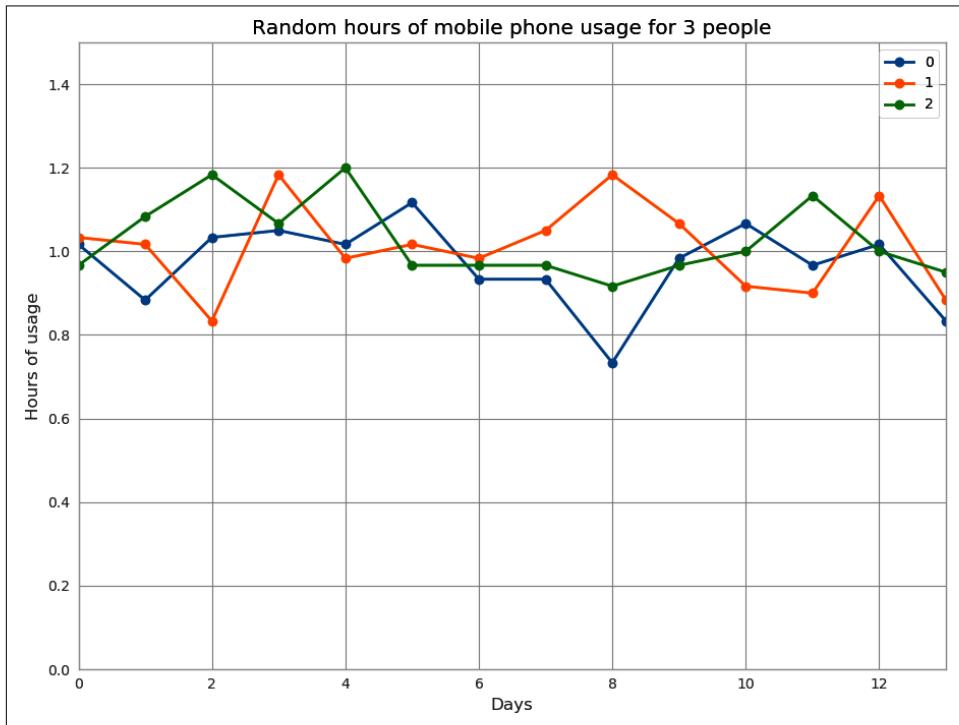


Figure 6-5. Synthetic data for the first three simulated users showing 14 days of cell phone usage

A bonus of generating 100,000 rows of data is that some rows will, by random variation alone, exhibit “increasing counts,” and some will exhibit “decreasing counts.” Note that there is no signal behind this in our synthetic data since the points are drawn independently; simply because we generate many rows of data, we’re going to see a variance in the ultimate slopes of the lines we calculate.

This is convenient, as we can identify the “most growing” and “most declining” lines and draw them as a validation that we’re identifying the sort of signal we hope to export on the real-world problem. Figure 6-6 shows two of our random traces with maximal and minimal slopes (m).

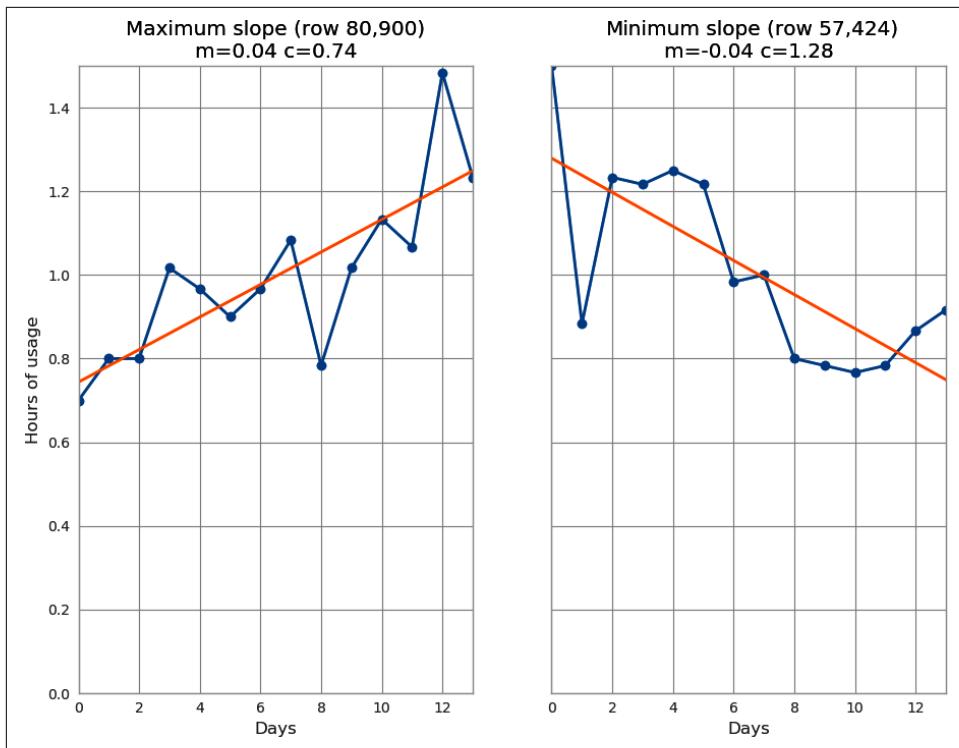


Figure 6-6. The “most increasing” and “most decreasing” usage in our randomly generated dataset

We'll start with scikit-learn's `LinearRegression` estimator to calculate each m . While this method is correct, we'll see in the following section that it incurs a surprising overhead against another approach.

Which OLS implementation should we use?

Example 6-24 shows three implementations that we'd like to try. We'll evaluate the scikit-learn implementation against the direct linear algebra implementation using NumPy. Both methods ultimately perform the same job and calculate the slopes (m) and intercept (c) of the target data from each Pandas row given an increasing x range (with values $[0, 1, \dots, 13]$).

scikit-learn will be a default choice for many machine learning practitioners, while a linear algebra solution may be preferred by those coming from other disciplines.

Example 6-24. Solving Ordinary Least Squares with NumPy and scikit-learn

```
def ols_sklearn(row):
    """Solve OLS using scikit-learn's LinearRegression"""
    est = LinearRegression()
    X = np.arange(row.shape[0]).reshape(-1, 1) # shape (14, 1)
    # note that the intercept is built inside LinearRegression
    est.fit(X, row.values)
    m = est.coef_[0] # note c is in est.intercept_
    return m

def ols_lstsq(row):
    """Solve OLS using numpy.linalg.lstsq"""
    # build X values for [0, 13]
    X = np.arange(row.shape[0]) # shape (14,)
    ones = np.ones(row.shape[0]) # constant used to build intercept
    A = np.vstack((X, ones)).T # shape(14, 2)
    # lstsq returns the coefficient and intercept as the first result
    # followed by the residuals and other items
    m, c = np.linalg.lstsq(A, row.values, rcond=-1)[0]
    return m

def ols_lstsq_raw(row):
    """Variant of `ols_lstsq` where row is a numpy array (not a Series)"""
    X = np.arange(row.shape[0])
    ones = np.ones(row.shape[0])
    A = np.vstack((X, ones)).T
    m, c = np.linalg.lstsq(A, row, rcond=-1)[0]
    return m
```

Surprisingly, if we call `ols_sklearn` 10,000 times with the `timeit` module, we find that it takes at least 0.483 microseconds to execute, while `ols_lstsq` on the same data takes 0.182 microseconds. The popular scikit-learn solution takes more than twice as long as the terse NumPy variant!

Building on the profiling from “[Using line_profiler for Line-by-Line Measurements](#)” on page 40, we can use the object interface (rather than the command line or Jupyter magic interfaces) to learn *why* the scikit-learn implementation is slower. In Example 6-25, we tell `LineProfiler` to profile `est.fit` (that’s the scikit-learn `fit` method on our `LinearRegression` estimator) and then call `run` with arguments based on the DataFrame we used before.

We see a couple of surprises. The very last line of `fit` calls the same `linalg.lstsq` that we’ve called in `ols_lstsq`—so what else is going on to cause our slowdown? `LineProfiler` reveals that scikit-learn is calling two other expensive methods, namely `check_X_y` and `_preprocess_data`.

Both of these are designed to help us avoid making mistakes—indeed, your author Ian has been saved repeatedly from passing in inappropriate data such as a wrongly

shaped array or one containing NaNs to scikit-learn estimators. A consequence of this checking is that it takes more time—more safety makes things run slower! We’re trading developer time (and sanity) against execution time.

Example 6-25. Digging into scikit-learn’s `LinearRegression.fit` call

```
...
lp = LineProfiler(est.fit)
print("Run on a single row")
lp.run("est.fit(X, row.values)")
lp.print_stats()

Line #  % Time  Line Contents
=====
438          def fit(self, X, y, sample_weight=None):
...
462      0.3          X, y = check_X_y(X, y,
463                                accept_sparse=['csr', 'csc', 'coo'],
464                                y_numeric=True, multi_output=True)
...
468      0.3          X, y, X_offset, y_offset, X_scale = \
469                                self._preprocess_data(
470                                X, y,
471                                fit_intercept=self.fit_intercept,
472                                normalize=self.normalize,
473                                copy=self.copy_X,
474                                sample_weight=sample_weight,
475                                return_mean=True)
...
502          self.coef_, self._residues,
503          self.rank_, self.singular_ = \
linalg.lstsq(X, y)
```

Behind the scenes, these two methods are performing various checks, including these:

- Checking for appropriate sparse NumPy arrays (even though we’re using dense arrays in this example)
- Offsetting the input array to a mean of 0 to improve numerical stability on wider data ranges than we’re using
- Checking that we’re providing a 2D X array
- Checking that we’re not providing NaN or Inf values
- Checking that we’re providing non-empty arrays of data

Generally, we prefer to have all of these checks enabled—they’re here to help us avoid painful debugging sessions, which kill developer productivity. If we know that our data is of the correct form for our chosen algorithm, these checks will add a penalty.

It is up to you to decide when the safety of these methods is hurting your overall productivity.

As a general rule—stay with the safer implementations (scikit-learn, in this case) *unless* you’re confident that your data is in the right form and you’re optimizing for performance. We’re after increased performance, so we’ll continue with the `ols_lstsq` approach.

Applying `lstsq` to our rows of data

We’ll start with an approach that many Python developers who come from other programming languages may try. This is *not* idiomatic Python, nor is it common or even efficient for Pandas. It does have the advantage of being very easy to understand. In [Example 6-26](#), we’ll iterate over the index of the DataFrame from row 0 to row 99,999; on each iteration we’ll use `iloc` to retrieve a row, and then we’ll calculate OLS on that row.

The calculation is common to each of the following methods—what’s different is how we iterate over the rows. This method takes 18.6 seconds; it is by far the slowest approach (by a factor of 3) among the options we’re evaluating.

Behind the scenes, each dereference is expensive—`iloc` does a lot of work to get to the row using a fresh `row_idx`, which is then converted into a new Series object, which is returned and assigned to `row`.

Example 6-26. Our worst implementation—counting and fetching rows one at a time with `iloc`

```
ms = []
for row_idx in range(df.shape[0]):
    row = df.iloc[row_idx]
    m = ols_lstsq(row)
    ms.append(m)
results = pd.Series(ms)
```

Next, we’ll take a more idiomatic Python approach: in [Example 6-27](#), we iterate over the rows using `iterrows`, which looks similar to how we might iterate over a Python iterable (e.g., a `list` or `set`) with a `for` loop. This method looks sensible and is a little faster—it takes 12.4 seconds.

This is more efficient, as we don’t have to do so many lookups—`iterrows` can walk along the rows without doing lots of sequential lookups. `row` is still created as a fresh Series on each iteration of the loop.

Example 6-27. iterrows for more efficient and “Python-like” row operations

```
ms = []
for row_idx, row in df.iterrows():
    m = ols_lstsq(row)
    ms.append(m)
results = pd.Series(ms)
```

Example 6-28 skips a lot of the Pandas machinery, so a lot of overhead is avoided. `apply` passes the function `ols_lstsq` a new row of data directly (again, a fresh Series is constructed behind the scenes for each row) without creating Python intermediate references. This costs 6.8 seconds—this is a significant improvement, and the code is more compact and readable!

Example 6-28. apply for idiomatic Pandas function application

```
ms = df.apply(ols_lstsq, axis=1)
results = pd.Series(ms)
```

Our final variant in **Example 6-29** uses the same `apply` call with an additional `raw=True` argument. Using `raw=True` stops the creation of an intermediate Series object. As we don’t have a Series object, we have to use our third OLS function, `ols_lstsq_raw`; this variant has direct access to the underlying NumPy array.

By avoiding the creation and dereferencing of an intermediate Series object, we shave our execution time a little more, down to 5.3 seconds.

Example 6-29. Avoiding intermediate Series creation using raw=True

```
ms = df.apply(ols_lstsq_raw, axis=1, raw=True)
results = pd.Series(ms)
```

The use of `raw=True` gives us the option to compile with Numba (“[Numba to Compile NumPy for Pandas](#)” on page 182) or with Cython as it removes the complication of compiling Pandas layers that currently aren’t supported.

We’ll summarize the execution times in **Table 6-3** for 100,000 rows of data on a single window of 14 columns of simulated data. New Pandas users often use `iloc` and `iterrows` (or the similar `itertuples`) when `apply` would be preferred.

By performing our analysis and considering our potential need to perform OLS on 1,000,000 rows by up to 730 windows of data, we can see that a first naive approach combining `iloc` with `ols_sklearn` might cost $10 \times 730 \times 18 \times 2 = 73$ hours.

If we used `ols_lstsq_raw` and our fastest approach, the same calculations might take $10 * 730 * 5.3$ seconds == 10 hours. This is a significant saving for a task that represents what might be a suite of similar operations. We'll see even faster solutions if we compile and run on multiple cores.

Table 6-3. Cost for using `lstsq` with various Pandas row-wise approaches

Method	Time in seconds
<code>iloc</code>	18.6
<code>iterrows</code>	12.4
<code>apply</code>	6.8
<code>apply raw=True</code>	5.3

Earlier we discovered that the scikit-learn approach adds significant overhead to our execution time by covering our data with a safety net of checks. We can remove this safety net but with a potential cost on developer debugging time. Your authors strongly urge you to consider adding unit-tests to your code that would verify that a well-known and well-debugged method is used to test any optimized method you settle on. If you added a unit test to compare the scikit-learn `LinearRegression` approach against `ols_lstsq`, you'd be giving yourself and other colleagues a future hint about why you developed a less obvious solution to what appeared to be a standard problem.

Having experimented, you may also conclude that the heavily tested scikit-learn approach is more than fast enough for your application and that you're more comfortable using a library that is well known by other developers. This could be a very sane conclusion.

Later in “[Parallel Pandas with Dask](#)” on page 322, we’ll look at running Pandas operations across multiple cores by dividing data into groups of rows using Dask and Swifter. In “[Numba to Compile NumPy for Pandas](#)” on page 182, we look at compiling the `raw=True` variant of `apply` to achieve an order of magnitude speedup. Compilation and parallelization can be combined for a really significant final speedup, dropping our expected runtime from around 10 hours to just 30 minutes.

Building DataFrames and Series from Partial Results Rather than Concatenating

You may have wondered in [Example 6-26](#) why we built up a list of partial results that we then turned into a Series, rather than incrementally building up the Series as we went. Our earlier approach required building up a list (which has a memory overhead) and then building a *second* structure for the Series, giving us two objects in memory. This brings us to another common mistake when using Pandas and NumPy.

As a general rule, you should avoid repeated calls to `concat` in Pandas (and to the equivalent `concatenate` in NumPy). In [Example 6-30](#), we see a similar solution to the preceding one but without the intermediate `ms` list. This solution takes 56 seconds, as opposed to the solution using a list at 18.6 seconds!

Example 6-30. Concatenating each result incurs a significant overhead—avoid this!

```
results = None
for row_idx in range(df.shape[0]):
    row = df.iloc[row_idx]
    m = ols_lstsq(row)
    if results is None:
        results = pd.Series([m])
    else:
        results = pd.concat((results, pd.Series([m])))
```

Each concatenation creates an entirely *new* Series object in a new section of memory that is one row longer than the previous item. In addition, we have to make a temporary Series object for each new `m` on each iteration. We strongly recommend building up lists of intermediate results and then constructing a Series or DataFrame from this list, rather than concatenating to an existing object.

There's More Than One (and Possibly a Faster) Way to Do a Job

Because of the evolution of Pandas, there are typically a couple of approaches to solving the same task, some of which incur more overhead than others. Let's take the OLS DataFrame and convert one column into a string; we'll then time some string operations. With string-based columns containing names, product identifiers, or codes, it is common to have to preprocess the data to turn it into something we can analyze.

Let's say that we need to find the location, if it exists, of the number 9 in the digits from one of the columns. While this operation serves no real purpose, it is very similar to checking for the presence of a code-bearing symbol in an identifier's sequence or checking for an honorific in a name. Typically for these operations, we'd use `strip` to remove extraneous whitespace, `lower` and `replace` to normalize the string, and `find` to locate something of interest.

In [Example 6-31](#), we first build a new Series named `theta_as_str`, which is the zeroth Series of random numbers converted into a printable string form. We'll then run two variants of string manipulation code—both will remove the leading digit and decimal point and then use Python's `find` to locate the first 9 if it exists, returning `-1` otherwise.

Example 6-31. `str` Series operations versus `apply` for string processing

```
In [10]: df['0_as_str'] = df[0].apply(lambda v: str(v))
Out[10]:
          0           0_as_str
0    1.016667  1.0166666666666666
1    1.033333  1.0333333333333334
2    0.966667  0.9666666666666667
...
def find_9(s):
    """Return -1 if '9' not found else its location at position >= 0"""
    return s.split('.')[1].find('9')

In [11]: df['0_as_str'].str.split('.', expand=True)[1].str.find('9')
Out[11]:
0      -1
1      -1
2       0

In [12]: %timeit df['0_as_str'].str.split('.', expand=True)[1].str.find('9')
Out[12]: 183 ms ± 4.62 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [13]: %timeit df['0_as_str'].apply(find_9)
Out[13]: 51 ms ± 987 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The one-line approach uses Pandas’s `str` operations to access Python’s string methods for a Series. For `split`, we expand the returned result into two columns (the first column contains the leading digit, and the second contains everything after the decimal place), and we select column 1. We then apply `find` to locate the digit 9. The second approach uses `apply` and the function `find_9`, which reads like a regular Python string-processing function.

We can use `%timeit` to check the runtime—this shows us that there’s a $3.5\times$ speed difference between the two methods, even though they both produce the same result! In the former one-line case, Pandas has to make several new intermediate Series objects, which adds overhead; in the `find_9` case, all of the string-processing work occurs one line at a time without creating new intermediate Pandas objects.

Further benefits of the `apply` approach are that we could parallelize this operation (see “Parallel Pandas with Dask” on page 322 for an example with Dask and Swifter), and we can write a unit test that succinctly confirms the operations performed by `find_9`, which will aid in readability and maintenance.

Advice for Effective Pandas Development

Install the optional dependencies `numexpr` and `bottleneck` for additional performance improvements. These don’t get installed by default, and you won’t be told if they’re missing. `bottleneck` is rarely used in the code base; `numexpr`, however, will give significant speedups in some situations when you use `exec`. You can test for the presence of both in your environment with `import bottleneck` and `import numexpr`.

Don’t write your code too tersely; remember to make your code easy to read and debug to help your future self. While the “method chaining” style is supported, your authors would caution against chaining too many rows of Pandas operations in sequence. It typically becomes difficult to figure out which line has problems when debugging, and then you have to split up the lines—you’re better off chaining only a couple of operations together at most to simplify your maintenance.

Avoid doing more work than necessary: it is preferable to filter your data before calculating on the remaining rows rather than filtering after calculating, if possible. For high performance in general, we want to ask the machine to do as little computation as possible; if you can filter out or mask away portions of your data, you’re probably winning. If you’re consuming from a SQL source and later joining or filtering in Pandas, you might want to try to filter first at the SQL level, to avoid pulling more data than necessary into Pandas. You may *not* want to do this at first if you’re investigating data quality, as having a simplified view on the variety of datatypes you have might be more beneficial.

Check the schema of your DataFrames as they evolve; with a tool like `bulwark`, you guarantee at runtime that your schema is being met, and you can visually confirm when you’re reviewing code that your expectations are being met. Keep renaming your columns as you generate new results so that your DataFrame’s contents make sense to you; sometimes `groupby` and other operations give you silly default names, which can later be confusing. Drop columns that you no longer need with `.drop()` to reduce bloat and memory usage.

For large Series containing strings with low cardinality (“yes” and “no,” for example, or “`type_a`,” “`type_b`,” and “`type_c`”), try converting the Series to a Category `dtype` with `df['series_of_strings'].astype('category')`; you may find that operations like `value_counts` and `groupby` run faster, and the Series is likely to consume less RAM.

Similarly, you may want to convert 8-byte `float64` and `int64` columns to smaller datatypes—perhaps the 2-byte `float16` or 1-byte `int8` if you need a smaller range to further save RAM.

As you evolve DataFrames and generate new copies, remember that you can use the `del` keyword to delete earlier references and clear them from memory, if they’re large and wasting space. You can also use the Pandas `drop` method to delete unused columns.

If you’re manipulating large DataFrames while you prepare your data for processing, it may make sense to do these operations once in a function or a separate script and then persist the prepared version to disk by using `to_pickle`. You can subsequently work on the prepared DataFrame without having to process it each time.

Avoid the `inplace=True` operator—in-place operations are scheduled to be removed from the library over time.

Finally, always add unit tests to any processing code, as it will quickly become more complex and harder to debug. Developing your tests up front guarantees that your code meets your expectations and helps you to avoid silly mistakes creeping in later that cost developer time to debug.

Existing tools for making Pandas go faster include [Modin](#) and the GPU-focused [cuDF](#). Modin and cuDF take different approaches to parallelizing common data operations on a Pandas DataFrame–like object.

We’d like to also give an honorable mention to the new [Vaex library](#). Vaex is designed to work on very large datasets that exceed RAM by using lazy evaluation while retaining a similar interface to that of Pandas. In addition, Vaex offers a slew of built-in visualization functions. One design goal is to use as many CPUs as possible, offering parallelism for free where possible.

Vaex specializes in both larger datasets and string-heavy operations; the authors have rewritten many of the string operations to avoid the standard Python functions and instead use faster Vaex implementations in C++. Note that Vaex is not guaranteed to work in the same way as Pandas, so it is possible that you’ll find edge cases with different behavior—as ever, back your code with unit tests to gain confidence if you’re trying both Pandas and Vaex to process the same data.

Wrap-Up

In the next chapter, we will talk about how to create your own external modules that can be finely tuned to solve specific problems with much greater efficiencies. This allows us to follow the rapid prototyping method of making our programs—first solve the problem with slow code, then identify the elements that are slow, and finally, find ways to make those elements faster. By profiling often and trying to optimize only the sections of code we *know* are slow, we can save ourselves time while still making our programs run as fast as possible.

Compiling to C

Questions You'll Be Able to Answer After This Chapter

- How can I have my Python code run as lower-level code?
- What is the difference between a JIT compiler and an AOT compiler?
- What tasks can compiled Python code perform faster than native Python?
- Why do type annotations speed up compiled Python code?
- How can I write modules for Python using C or Fortran?
- How can I use libraries from C or Fortran in Python?

The easiest way to get your code to run faster is to make it do less work. Assuming you've already chosen good algorithms and you've reduced the amount of data you're processing, the easiest way to execute fewer instructions is to compile your code down to machine code.

Python offers a number of options for this, including pure C-based compiling approaches like Cython; LLVM-based compiling via Numba; and the replacement virtual machine PyPy, which includes a built-in just-in-time (JIT) compiler. You need to balance the requirements of code adaptability and team velocity when deciding which route to take.

Each of these tools adds a new dependency to your toolchain, and Cython requires you to write in a new language type (a hybrid of Python and C), which means you need a new skill. Cython's new language may hurt your team's velocity, as team members without knowledge of C may have trouble supporting this code; in practice,

though, this is probably a minor concern, as you'll use Cython only in well-chosen, small regions of your code.

It is worth noting that performing CPU and memory profiling on your code will probably start you thinking about higher-level algorithmic optimizations that you might apply. These algorithmic changes (such as additional logic to avoid computations or caching to avoid recalculation) could help you avoid doing unnecessary work in your code, and Python's expressivity helps you to spot these algorithmic opportunities. Radim Řehůřek discusses how a Python implementation can beat a pure C implementation in “[Making Deep Learning Fly with RadimRehurek.com \(2014\)](#)” on [page 415](#).

In this chapter we'll review the following:

- Cython, the most commonly used tool for compiling to C, covering both `numpy` and normal Python code (requires some knowledge of C)
- Numba, a new compiler specialized for `numpy` code
- PyPy, a stable just-in-time compiler for non-`numpy` code that is a replacement for the normal Python executable

Later in the chapter we'll look at foreign function interfaces, which allow C code to be compiled into extension modules for Python. Python's native API is used with `ctypes` or with `cffi` (from the authors of PyPy), along with the `f2py` Fortran-to-Python converter.

What Sort of Speed Gains Are Possible?

Gains of an order of magnitude or more are quite possible if your problem yields to a compiled approach. Here, we'll look at various ways to achieve speedups of one to two orders of magnitude on a single core, along with using multiple cores through OpenMP.

Python code that tends to run faster after compiling is mathematical, and it has lots of loops that repeat the same operations many times. Inside these loops, you're probably making lots of temporary objects.

Code that calls out to external libraries (such as regular expressions, string operations, and calls to database libraries) is unlikely to show any speedup after compiling. Programs that are I/O-bound are also unlikely to show significant speedups.

Similarly, if your Python code focuses on calling vectorized `numpy` routines, it may not run any faster after compilation—it'll run faster only if the code being compiled is mainly Python (and probably if it is mainly looping). We looked at `numpy`

operations in [Chapter 6](#); compiling doesn't really help because there aren't many intermediate objects.

Overall, it is very unlikely that your compiled code will run any faster than a hand-crafted C routine, but it is also unlikely to run much slower. It is quite possible that the generated C code from your Python will run as fast as a handwritten C routine, unless the C coder has particularly good knowledge of ways to tune the C code to the target machine's architecture.

For math-focused code, it is possible that a handcoded Fortran routine will beat an equivalent C routine, but again, this probably requires expert-level knowledge. Overall, a compiled result (probably using Cython) will be as close to a handcoded-in-C result as most programmers will need.

Keep the diagram in [Figure 7-1](#) in mind when you profile and work on your algorithm. A small amount of work understanding your code through profiling should enable you to make smarter choices at an algorithmic level. After this, some focused work with a compiler should buy you an additional speedup. It will probably be possible to keep tweaking your algorithm, but don't be surprised to see increasingly small improvements coming from increasingly large amounts of work on your part. Know when additional effort isn't useful.

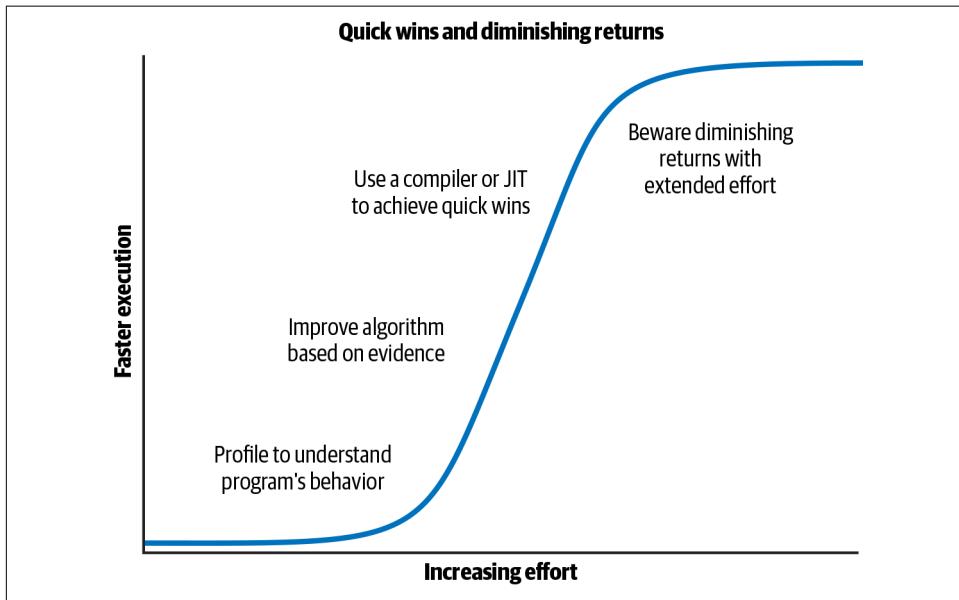


Figure 7-1. Some effort profiling and compiling brings a lot of reward, but continued effort tends to pay increasingly less

If you’re dealing with Python code and batteries-included libraries without numpy, Cython and PyPy are your main choices. If you’re working with numpy, Cython and Numba are the right choices. These tools all support Python 3.6+.

Some of the following examples require a little understanding of C compilers and C code. If you lack this knowledge, you should learn a little C and compile a working C program before diving in too deeply.

JIT Versus AOT Compilers

The tools we’ll look at split roughly into two sets: tools for compiling ahead of time, or AOT (Cython), and tools for compiling “just in time,” or JIT (Numba, PyPy).

By compiling AOT, you create a static library that’s specialized to your machine. If you download numpy, scipy, or scikit-learn, it will compile parts of the library using Cython on your machine (or you’ll use a prebuilt compiled library, if you’re using a distribution like Continuum’s Anaconda). By compiling ahead of use, you’ll have a library that can instantly be used to work on solving your problem.

By compiling JIT, you don’t have to do much (if any) work up front; you let the compiler step in to compile just the right parts of the code at the time of use. This means you have a “cold start” problem—if most of your program could be compiled and currently none of it is, when you start running your code, it’ll run very slowly while it compiles. If this happens every time you run a script and you run the script many times, this cost can become significant. PyPy suffers from this problem, so you may not want to use it for short but frequently running scripts.

The current state of affairs shows us that compiling ahead of time buys us the best speedups, but often this requires the most manual effort. Just-in-time compiling offers some impressive speedups with very little manual intervention, but it can also run into the problem just described. You’ll have to consider these trade-offs when choosing the right technology for your problem.

Why Does Type Information Help the Code Run Faster?

Python is dynamically typed—a variable can refer to an object of any type, and any line of code can change the type of the object that is referred to. This makes it difficult for the virtual machine to optimize how the code is executed at the machine code level, as it doesn’t know which fundamental datatype will be used for future operations. Keeping the code generic makes it run more slowly.

In the following example, `v` is either a floating-point number or a pair of floating-point numbers that represent a complex number. Both conditions could occur in the same loop at different points in time, or in related serial sections of code:

```
v = -1.0
print(type(v), abs(v))
<class 'float'> 1.0

v = 1-1j
print(type(v), abs(v))
<class 'complex'> 1.4142135623730951
```

The `abs` function works differently depending on the underlying datatype. Using `abs` for an integer or a floating-point number turns a negative value into a positive value. Using `abs` for a complex number involves taking the square root of the sum of the squared components:

$$abs(c) = \sqrt{c.real^2 + c.imag^2}$$

The machine code for the `complex` example involves more instructions and will take longer to run. Before calling `abs` on a variable, Python first has to look up the type of the variable and then decide which version of a function to call—this overhead adds up when you make a lot of repeated calls.

Inside Python, every fundamental object, such as an integer, will be wrapped up in a higher-level Python object (e.g., an `int` for an integer). The higher-level object has extra functions like `__hash__` to assist with storage and `__str__` for printing.

Inside a section of code that is CPU-bound, it is often the case that the types of variables do not change. This gives us an opportunity for static compilation and faster code execution.

If all we want are a lot of intermediate mathematical operations, we don't need the higher-level functions, and we may not need the machinery for reference counting either. We can drop down to the machine code level and do our calculations quickly using machine code and bytes, rather than manipulating the higher-level Python objects, which involves greater overhead. To do this, we determine the types of our objects ahead of time so we can generate the correct C code.

Using a C Compiler

In the following examples, we'll use `gcc` and `g++` from the GNU C Compiler toolset. You could use an alternative compiler (e.g., Intel's `icc` or Microsoft's `cl`) if you configure your environment correctly. Cython uses `gcc`.

`gcc` is a very good choice for most platforms; it is well supported and quite advanced. It is often possible to squeeze out more performance by using a tuned compiler (e.g., Intel's `icc` may produce faster code than `gcc` on Intel devices), but the cost is that you

have to gain more domain knowledge and learn how to tune the flags on the alternative compiler.

C and C++ are often used for static compilation rather than other languages like Fortran because of their ubiquity and the wide range of supporting libraries. The compiler and the converter, such as Cython, can study the annotated code to determine whether static optimization steps (like inlining functions and unrolling loops) can be applied.

Aggressive analysis of the intermediate abstract syntax tree (performed by Numba and PyPy) provides opportunities to combine knowledge of Python's way of expressing things to inform the underlying compiler how best to take advantage of the patterns that have been seen.

Reviewing the Julia Set Example

Back in [Chapter 2](#) we profiled the Julia set generator. This code uses integers and complex numbers to produce an output image. The calculation of the image is CPU-bound.

The main cost in the code was the CPU-bound nature of the inner loop that calculates the `output` list. This list can be drawn as a square pixel array, where each value represents the cost to generate that pixel.

The code for the inner function is shown in [Example 7-1](#).

Example 7-1. Reviewing the Julia function's CPU-bound code

```
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 n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

On Ian's laptop, the original Julia set calculation on a $1,000 \times 1,000$ grid with `max_iter=300` takes approximately 8 seconds using a pure Python implementation running on CPython 3.7.

Cython

Cython is a compiler that converts type-annotated Python into a compiled extension module. The type annotations are C-like. This extension can be imported as a regular Python module using `import`. Getting started is simple, but a learning curve must be climbed with each additional level of complexity and optimization. For Ian, this is the tool of choice for turning calculation-bound functions into faster code, because of its wide usage, its maturity, and its OpenMP support.

With the OpenMP standard, it is possible to convert parallel problems into multiprocessing-aware modules that run on multiple CPUs on one machine. The threads are hidden from your Python code; they operate via the generated C code.

Cython (announced in 2007) is a fork of Pyrex (announced in 2002) that expands the capabilities beyond the original aims of Pyrex. Libraries that use Cython include SciPy, scikit-learn, lxml, and ZeroMQ.

Cython can be used via a `setup.py` script to compile a module. It can also be used interactively in IPython via a “magic” command. Typically, the types are annotated by the developer, although some automated annotation is possible.

Compiling a Pure Python Version Using Cython

The easy way to begin writing a compiled extension module involves three files. Using our Julia set as an example, they are as follows:

- The calling Python code (the bulk of our Julia code from earlier)
- The function to be compiled in a new `.pyx` file
- A `setup.py` that contains the instructions for calling Cython to make the extension module

Using this approach, the `setup.py` script is called to use Cython to compile the `.pyx` file into a compiled module. On Unix-like systems, the compiled module will probably be a `.so` file; on Windows it should be a `.pyd` (DLL-like Python library).

For the Julia example, we’ll use the following:

- `julia1.py` to build the input lists and call the calculation function
- `cythonfn.pyx`, which contains the CPU-bound function that we can annotate
- `setup.py`, which contains the build instructions

The result of running `setup.py` is a module that can be imported. In our `julia1.py` script in [Example 7-2](#), we need only to make some tiny changes to `import` the new module and call our function.

Example 7-2. Importing the newly compiled module into our main code

```
...
import cythonfn # as defined in setup.py
...
def calc_pure_python(desired_width, max_iterations):
    # ...
    start_time = time.time()
    output = cythonfn.calculate_z(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print(f"Took {secs:.2f} seconds")
...

```

In [Example 7-3](#), we will start with a pure Python version without type annotations.

Example 7-3. Unmodified pure Python code in cythonfn.pyx (renamed from .py) for Cython's setup.py

```
# cythonfn.pyx
def calculate_z(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 n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

The *setup.py* script shown in [Example 7-4](#) is short; it defines how to convert *cythonfn.pyx* into *calculate.so*.

Example 7-4. setup.py, which converts cythonfn.pyx into C code for compilation by Cython

```
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize("cythonfn.pyx",
                           compiler_directives={"language_level": "3"}))
```

When we run the *setup.py* script in [Example 7-5](#) with the argument `build_ext`, Cython will look for *cythonfn.pyx* and build *cythonfn[...].so*. The `language_level` is hardcoded to 3 here to force Python 3.x support.



Remember that this is a manual step—if you update your `.pyx` or `setup.py` and forget to rerun the build command, you won’t have an updated `.so` module to import. If you’re unsure whether you compiled the code, check the timestamp for the `.so` file. If in doubt, delete the generated C files and the `.so` file and build them again.

Example 7-5. Running setup.py to build a new compiled module

```
$ python setup.py build_ext --inplace
Compiling cythonfn.pyx because it changed.
[1/1] Cythonizing cythonfn.pyx
running build_ext
building 'cythonfn' extension
gcc -pthread -B /home/ian/miniconda3/envs/high_performance_python_book_2e/...
gcc -pthread -shared -B /home/ian/miniconda3/envs/high_performance_python_...
```

The `--inplace` argument tells Cython to build the compiled module into the current directory rather than into a separate `build` directory. After the build has completed, we’ll have the intermediate `cythonfn.c`, which is rather hard to read, along with `cythonfn[...].so`.

Now when the `julia1.py` code is run, the compiled module is imported, and the Julia set is calculated on Ian’s laptop in 4.7 seconds, rather than the more usual 8.3 seconds. This is a useful improvement for very little effort.

pyximport

A simplified build system has been introduced via `pyximport`. If your code has a simple setup and doesn’t require third-party modules, you may be able to do away with `setup.py` completely.

By importing `pyximport` as seen in [Example 7-6](#) and calling `install`, any subsequently imported `.pyx` file will be automatically compiled. This `.pyx` file can include annotations, or in this case, it can be the unannotated code. The result runs in 4.7 seconds, as before; the only difference is that we didn’t have to write a `setup.py` file.

Example 7-6. Using pyximport to replace setup.py

```
import pyximport
pyximport.install(language_level=3)
import cythonfn
# followed by the usual code
```

Cython Annotations to Analyze a Block of Code

The preceding example shows that we can quickly build a compiled module. For tight loops and mathematical operations, this alone often leads to a speedup. Obviously, though, we should not optimize blindly—we need to know which lines of code take a lot of time so we can decide where to focus our efforts.

Cython has an annotation option that will output an HTML file we can view in a browser. We use the command `cython -a cythonfn.pyx`, and the output file `cythonfn.html` is generated. Viewed in a browser, it looks something like Figure 7-2. A similar image is available in the [Cython documentation](#).

Generated by Cython 0.29.13

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: [cythonfn.c](#)

```
+01: def calculate_z(maxiter, zs, cs):
+02:     """Calculate output list using Julia update rule"""
+03:     output = [0] * len(zs)
+04:     for i in range(len(zs)):
+05:         n = 0
+06:         z = zs[i]
+07:         c = cs[i]
+08:         while n < maxiter and abs(z) < 2:
+09:             z = z * z + c
+10:             n += 1
+11:             output[i] = n
+12:     return output
```

Figure 7-2. Colored Cython output of unannotated function

Each line can be expanded with a double-click to show the generated C code. More yellow means “more calls into the Python virtual machine,” while more white means “more non-Python C code.” The goal is to remove as many of the yellow lines as possible and end up with as much white as possible.

Although “more yellow lines” means more calls into the virtual machine, this won’t necessarily cause your code to run slower. Each call into the virtual machine has a cost, but the cost of those calls will be significant only if the calls occur inside large loops. Calls outside large loops (for example, the line used to create `output` at the start of the function) are not expensive relative to the cost of the inner calculation loop. Don’t waste your time on the lines that don’t cause a slowdown.

In our example, the lines with the most calls back into the Python virtual machine (the “most yellow”) are lines 4 and 8. From our previous profiling work, we know that line 8 is likely to be called over 30 million times, so that’s a great candidate to focus on.

Lines 9, 10, and 11 are almost as yellow, and we also know they’re inside the tight inner loop. In total they’ll be responsible for the bulk of the execution time of this function, so we need to focus on these first. Refer back to “[Using line_profiler for](#)

[“Line-by-Line Measurements” on page 40](#) if you need to remind yourself of how much time is spent in this section.

Lines 6 and 7 are less yellow, and since they’re called only 1 million times, they’ll have a much smaller effect on the final speed, so we can focus on them later. In fact, since they are `list` objects, there’s actually nothing we can do to speed up their access except, as you’ll see in “[Cython and numpy](#)” on page 176, to replace the `list` objects with `numpy` arrays, which will buy a small speed advantage.

To better understand the yellow regions, you can expand each line. In [Figure 7-3](#), we can see that to create the `output` list, we iterate over the length of `zs`, building new Python objects that are reference-counted by the Python virtual machine. Even though these calls are expensive, they won’t really affect the execution time of this function.

To improve the execution time of our function, we need to start declaring the types of objects that are involved in the expensive inner loops. These loops can then make fewer of the relatively expensive calls back into the Python virtual machine, saving us time.

In general, the lines that probably cost the most CPU time are those:

- Inside tight inner loops
- Dereferencing `list`, `array`, or `np.array` items
- Performing mathematical operations

```
Generated by Cython 0.29.13

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: cythonfn.c

+01: def calculate_z(maxiter, zs, cs):
+02:     """Calculate output list using Julia update rule"""
+03:     output = [0] * len(zs)
+04:     for i in range(len(zs)):
+05:         n = 0
+06:         z = zs[i]
+07:         c = cs[i]
+08:         while n < maxiter and abs(z) < 2:
+09:             z = z * z + c
+10:             n += 1
+11:         output[i] = n
+12:     return output
```

Figure 7-3. C code behind a line of Python code



If you don't know which lines are most frequently executed, using a profiling tool—`line_profiler`, discussed in “[Using `line_profiler` for Line-by-Line Measurements](#)” on page 40, would be the most appropriate. You'll learn which lines are executed most frequently and which lines cost the most inside the Python virtual machine, so you'll have clear evidence of which lines you need to focus on to get the best speed gain.

Adding Some Type Annotations

[Figure 7-2](#) shows that almost every line of our function is calling back into the Python virtual machine. All of our numeric work is also calling back into Python as we are using the higher-level Python objects. We need to convert these into local C objects, and then, after doing our numerical coding, we need to convert the result back to a Python object.

In [Example 7-7](#), we see how to add some primitive types by using the `cdef` syntax.



It is important to note that these types will be understood only by Cython and *not* by Python. Cython uses these types to convert the Python code to C objects, which do not have to call back into the Python stack; this means the operations run at a faster speed, but they lose flexibility and development speed.

The types we add are as follows:

- `int` for a signed integer
- `unsigned int` for an integer that can only be positive
- `double complex` for double-precision complex numbers

The `cdef` keyword lets us declare variables inside the function body. These must be declared at the top of the function, as that's a requirement from the C language specification.

Example 7-7. Adding primitive C types to start making our compiled function run faster by doing more work in C and less via the Python virtual machine

```
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    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

```



When adding Cython annotations, you're adding non-Python code to the `.pyx` file. This means you lose the interactive nature of developing Python in the interpreter. For those of you familiar with coding in C, we go back to the code-compile-run-debug cycle.

You might wonder if we could add a type annotation to the lists that we pass in. We can use the `list` keyword, but this has no practical effect for this example. The `list` objects still have to be interrogated at the Python level to pull out their contents, and this is very slow.

The act of giving types to some of the primitive objects is reflected in the annotated output in [Figure 7-4](#). Critically, lines 11 and 12—two of our most frequently called lines—have now turned from yellow to white, indicating that they no longer call back to the Python virtual machine. We can anticipate a great speedup compared to the previous version.

Generated by Cython 0.29.13

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: [cythonfn.c](#)

```

+01: def calculate_z(int maxiter, zs, cs):
02:     """Calculate output list using Julia update rule"""
03:     cdef unsigned int i, n
04:     cdef double complex z, c
+05:     output = [0] * len(zs)
+06:     for i in range(len(zs)):
+07:         n = 0
+08:         z = zs[i]
+09:         c = cs[i]
+10:         while n < maxiter and abs(z) < 2:
+11:             z = z * z + c
+12:             n += 1
+13:         output[i] = n
+14:     return output

```

Figure 7-4. Our first type annotations

After compiling, this version takes 0.49 seconds to complete. With only a few changes to the function, we are running at 15 times the speed of the original Python version.

It is important to note that the reason we are gaining speed is that more of the frequently performed operations are being pushed down to the C level—in this case, the updates to `z` and `n`. This means that the C compiler can optimize the way the lower-level functions are operating on the bytes that represent these variables, without calling into the relatively slow Python virtual machine.

As noted earlier in this chapter, `abs` for a complex number involves taking the square root of the sum of the squares of the real and imaginary components. In our test, we want to see if the square root of the result is less than 2. Rather than taking the square root, we can instead square the other side of the comparison, so we turn `< 2` into `< 4`. This avoids having to calculate the square root as the final part of the `abs` function.

In essence, we started with

$$\sqrt{c.real^2 + c.imag^2} < \sqrt{4}$$

and we have simplified the operation to

$$c.real^2 + c.imag^2 < 4$$

If we retained the `sqrt` operation in the following code, we would still see an improvement in execution speed. One of the secrets to optimizing code is to make it do as little work as possible. Removing a relatively expensive operation by considering the ultimate aim of a function means that the C compiler can focus on what it is good at, rather than trying to intuit the programmer's ultimate needs.

Writing equivalent but more specialized code to solve the same problem is known as *strength reduction*. You trade worse flexibility (and possibly worse readability) for faster execution.

This mathematical unwinding leads to [Example 7-8](#), in which we have replaced the relatively expensive `abs` function with a simplified line of expanded mathematics.

Example 7-8. Expanding the `abs` function by using Cython

```
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

By annotating the code, we see that the `while` on line 10 ([Figure 7-5](#)) has become a little more yellow—it looks as though it might be doing more work rather than less. It

isn't immediately obvious how much of a speed gain we'll get, but we know that this line is called over 30 million times, so we anticipate a good improvement.

This change has a dramatic effect—by reducing the number of Python calls in the innermost loop, we greatly reduce the calculation time of the function. This new version completes in just 0.19 seconds, an amazing 40× speedup over the original version. As ever, take a guide from what you see, but *measure* to test all of your changes!

```
Generated by Cython 0.29.13

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: cythonfn.c

+01: def calculate(z(int maxiter, zs, cs):
+02:     """Calculate output list using Julia update rule"""
+03:     cdef unsigned int i, n
+04:     cdef double complex z, c
+05:     output = [0] * len(zs)
+06:     for i in range(len(zs)):
+07:         n = 0
+08:         z = zs[i]
+09:         c = cs[i]
+10:         while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
+11:             z = z * z + c
+12:             n += 1
+13:         output[i] = n
+14:     return output
```

Figure 7-5. Expanded math to get a final win



Cython supports several methods of compiling to C, some easier than the full-type-annotation method described here. You should familiarize yourself with the **pure Python mode** if you'd like an easier start to using Cython, and look at `pyximport` to ease the introduction of Cython to colleagues.

For a final possible improvement on this piece of code, we can disable bounds checking for each dereference in the list. The goal of the bounds checking is to ensure that the program does not access data outside the allocated array—in C it is easy to accidentally access memory outside the bounds of an array, and this will give unexpected results (and probably a segmentation fault!).

By default, Cython protects the developer from accidentally addressing outside the list's limits. This protection costs a little bit of CPU time, but it occurs in the outer loop of our function, so in total it won't account for much time. Disabling bounds checking is usually safe unless you are performing your own calculations for array addressing, in which case you will have to be careful to stay within the bounds of the list.

Cython has a set of flags that can be expressed in various ways. The easiest is to add them as single-line comments at the start of the `.pyx` file. It is also possible to use a decorator or compile-time flag to change these settings. To disable bounds checking, we add a directive for Cython inside a comment at the start of the `.pyx` file:

```
#cython: boundscheck=False
def calculate_z(int maxiter, zs, cs):
```

As noted, disabling the bounds checking will save only a little bit of time as it occurs in the outer loop, not in the inner loop, which is more expensive. For this example, it doesn't save us any more time.



Try disabling bounds checking and wraparound checking if your CPU-bound code is in a loop that is dereferencing items frequently.

Cython and numpy

`list` objects (for background, see [Chapter 3](#)) have an overhead for each dereference, as the objects they reference can occur anywhere in memory. In contrast, `array` objects store primitive types in contiguous blocks of RAM, which enables faster addressing.

Python has the `array` module, which offers 1D storage for basic primitives (including integers, floating-point numbers, characters, and Unicode strings). NumPy's `numpy.array` module allows multidimensional storage and a wider range of primitive types, including complex numbers.

When iterating over an `array` object in a predictable fashion, the compiler can be instructed to avoid asking Python to calculate the appropriate address and instead move to the next primitive item in the sequence by going directly to its memory address. Since the data is laid out in a contiguous block, it is trivial to calculate the address of the next item in C by using an offset, rather than asking CPython to calculate the same result, which would involve a slow call back into the virtual machine.

You should note that if you run the following NumPy version *without* any Cython annotations (that is, if you just run it as a plain Python script), it'll take about 21 seconds to run—far in excess of the plain Python `list` version, which takes around 8 seconds. The slowdown is because of the overhead of dereferencing individual elements in the NumPy lists—it was never designed to be used this way, even though to a beginner this might feel like the intuitive way of handling operations. By compiling the code, we remove this overhead.

Cython has two special syntax forms for this. Older versions of Cython had a special access type for NumPy arrays, but more recently the generalized buffer interface protocol has been introduced through the `memoryview`—this allows the same low-level access to any object that implements the buffer interface, including NumPy arrays and Python arrays.

An added bonus of the buffer interface is that blocks of memory can easily be shared with other C libraries, without any need to convert them from Python objects into another form.

The code block in [Example 7-9](#) looks a little like the original implementation, except that we have added `memoryview` annotations. The function's second argument is `double complex[:,] zs`, which means we have a double-precision `complex` object using the buffer protocol as specified using `[]`, which contains a one-dimensional data block specified by the single colon `:`.

Example 7-9. Annotated numpy version of the Julia calculation function

```
# cythonfn.pyx
import numpy as np
cimport numpy as np

def calculate_z(int maxiter, double complex[:, ] zs, double complex[:, ] cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

In addition to specifying the input arguments by using the buffer annotation syntax, we also annotate the `output` variable, assigning a 1D `numpy` array to it via `empty`. The call to `empty` will allocate a block of memory but will not initialize the memory with sane values, so it could contain anything. We will overwrite the contents of this array in the inner loop so we don't need to reassign it with a default value. This is slightly faster than allocating and setting the contents of the array with a default value.

We also expanded the call to `abs` by using the faster, more explicit math version. This version runs in 0.18 seconds—a slightly faster result than the original Cythonized version of the pure Python Julia example in [Example 7-8](#). The pure Python version has an overhead every time it dereferences a Python `complex` object, but these dereferences occur in the outer loop and so don't account for much of the execution time. After the outer loop, we make native versions of these variables, and they operate at “C speed.” The inner loop for both this `numpy` example and the former pure Python

example are doing the same work on the same data, so the time difference is accounted for by the outer loop dereferences and the creation of the output arrays.

For reference, if we use the preceding code but don't expand the `abs` math, then the Cythonized result takes 0.49 seconds. This result makes it identical to the earlier equivalent pure Python version's runtime.

Parallelizing the Solution with OpenMP on One Machine

As a final step in the evolution of this version of the code, let's look at the use of the OpenMP C++ extensions to parallelize our embarrassingly parallel problem. If your problem fits this pattern, you can quickly take advantage of multiple cores in your computer.

Open Multi-Processing (OpenMP) is a well-defined cross-platform API that supports parallel execution and memory sharing for C, C++, and Fortran. It is built into most modern C compilers, and if the C code is written appropriately, the parallelization occurs at the compiler level, so it comes with relatively little effort to the developer through Cython.

With Cython, OpenMP can be added by using the `prange` (parallel range) operator and adding the `-fopenmp` compiler directive to `setup.py`. Work in a `prange` loop can be performed in parallel because we disable the (GIL). The GIL protects access to Python objects, preventing multiple threads or processes from accessing the same memory simultaneously, which might lead to corruption. By manually disabling the GIL, we're asserting that we won't corrupt our own memory. Be careful when you do this, and keep your code as simple as possible to avoid subtle bugs.

A modified version of the code with `prange` support is shown in [Example 7-10](#). with `nogil:` specifies the block, where the GIL is disabled; inside this block, we use `prange` to enable an OpenMP parallel for loop to independently calculate each `i`.



When disabling the GIL, we must *not* operate on regular Python objects (such as lists); we must operate only on primitive objects and objects that support the `memoryview` interface. If we operated on normal Python objects in parallel, we'd have to solve the associated memory-management problems that the GIL deliberately avoids. Cython doesn't prevent us from manipulating Python objects, and only pain and confusion can result if you do this!

Example 7-10. Adding `prange` to enable parallelization using OpenMP

```
# cythonfn.pyx
from cython.parallel import prange
import numpy as np
cimport numpy as np
```

```

def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, length
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
    length = len(zs)
    with nogil:
        for i in prange(length, schedule="guided"):
            z = zs[i]
            c = cs[i]
            output[i] = 0
            while output[i] < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
                z = z * z + c
                output[i] += 1
    return output

```

To compile `cythonfn.pyx`, we have to modify the `setup.py` script as shown in [Example 7-11](#). We tell it to inform the C compiler to use `-fopenmp` as an argument during compilation to enable OpenMP and to link with the OpenMP libraries.

Example 7-11. Adding the OpenMP compiler and linker flags to setup.py for Cython

```

#setup.py
from distutils.core import setup
from distutils.extension import Extension
import numpy as np

ext_modules = [Extension("cythonfn",
                        ["cythonfn.pyx"],
                        extra_compile_args=['-fopenmp'],
                        extra_link_args=['-fopenmp'])]

from Cython.Build import cythonize
setup(ext_modules=cythonize(ext_modules,
                           compiler_directives={"language_level": "3"},),
      include_dirs=[np.get_include()])

```

With Cython's `prange`, we can choose different scheduling approaches. With `static`, the workload is distributed evenly across the available CPUs. Some of our calculation regions are expensive in time, and some are cheap. If we ask Cython to schedule the work chunks equally using `static` across the CPUs, the results for some regions will complete faster than others, and those threads will then sit idle.

Both the `dynamic` and `guided` schedule options attempt to mitigate this problem by allocating work in smaller chunks dynamically at runtime, so that the CPUs are more evenly distributed when the workload's calculation time is variable. The correct choice will vary depending on the nature of your workload.

By introducing OpenMP and using `schedule="guided"`, we drop our execution time to approximately 0.05 seconds—the guided schedule will dynamically assign work, so fewer threads will wait for new work.

We also could have disabled the bounds checking for this example by using `#cython: boundscheck=False`, but it wouldn't improve our runtime.

Numba

Numba from Continuum Analytics is a just-in-time compiler that specializes in `numpy` code, which it compiles via the LLVM compiler (*not* via `g++` or `gcc++`, as used by our earlier examples) at runtime. It doesn't require a precompilation pass, so when you run it against new code, it compiles each annotated function for your hardware. The beauty is that you provide a decorator telling it which functions to focus on and then you let Numba take over. It aims to run on all standard `numpy` code.

Numba has been rapidly evolving since the first edition of this book. It is now fairly stable, so if you use `numpy` arrays and have nonvectorized code that iterates over many items, Numba should give you a quick and very painless win. Numba does not bind to external C libraries (which Cython can do), but it can automatically generate code for GPUs (which Cython cannot).

One drawback when using Numba is the toolchain—it uses LLVM, and this has many dependencies. We recommend that you use Continuum's Anaconda distribution, as everything is provided; otherwise, getting Numba installed in a fresh environment can be a very time-consuming task.

[Example 7-12](#) shows the addition of the `@jit` decorator to our core Julia function. This is all that's required; the fact that `numba` has been imported means that the LLVM machinery kicks in at execution time to compile this function behind the scenes.

Example 7-12. Applying the `@jit` decorator to a function

```
from numba import jit
...
@jit()
def calculate_z_serial_purepython(maxiter, zs, cs, output):
```

If the `@jit` decorator is removed, this is just the `numpy` version of the Julia demo running with Python 3.7, and it takes 21 seconds. Adding the `@jit` decorator drops the execution time to 0.75 seconds. This is very close to the result we achieved with Cython, but without all of the annotation effort.

If we run the same function a second time in the same Python session, it runs even faster at 0.47 seconds—there’s no need to compile the target function on the second pass if the argument types are the same, so the overall execution speed is faster. On the second run, the Numba result is equivalent to the Cython with `numpy` result we obtained before (so it came out as fast as Cython for very little work!). PyPy has the same warm-up requirement.

If you’d like to read another view on what Numba offers, see “[Numba](#)” on page 403, where core developer Valentin Haenel talks about the `@jit` decorator, viewing the original Python source, and going further with parallel options and the typed `List` and typed `Dict` for pure Python compiled interoperability.

Just as with Cython, we can add OpenMP parallelization support with `prange`. [Example 7-13](#) expands the decorator to require `nopython` and `parallel`. The `nopython` specifier means that if Numba cannot compile all of the code, it will fail. Without this, Numba can silently fall back on a Python mode that is slower; your code will run correctly, but you won’t see any speedups. Adding `parallel` enables support for `prange`. This version drops the general runtime from 0.47 seconds to 0.06 seconds. Currently Numba lacks support for OpenMP scheduling options (and with Cython, the guided scheduler runs slightly faster for this problem), but we expect support will be added in a future version.

Example 7-13. Using `prange` to add parallelization

```
@jit(nopython=False, parallel=True)
def calculate_z(maxiter, zs, cs, output):
    """Calculate output list using Julia update rule"""
    for i in prange(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real*z.real + z.imag*z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
```

When debugging with Numba, it is useful to note that you can ask Numba to show both the intermediate representation and the types for the function call. In [Example 7-14](#), we can see that `calculate_z` takes an `int64` and three `array` types.

Example 7-14. Debugging inferred types

```
print(calculate_z.inspect_types())
# calculate_z (int64, array(complex128, 1d, C),
#             array(complex128, 1d, C), array(int32, 1d, C))
```

Example 7-15 shows the continued output from the call to `inspect_types()`, where each line of compiled code is shown augmented by type information. This output is invaluable if you find that you can't get `nopython=True` to work; here, you'll be able to discover where your code isn't recognized by Numba.

Example 7-15. Viewing the intermediate representation from Numba

```
...
def calculate_z(maxiter, zs, cs, output):
    # --- LINE 14 ---
    """Calculate output list using Julia update rule"""

    # --- LINE 15 ---
    # maxiter = arg(0, name=maxiter) :: int64
    # zs = arg(1, name=zs) :: array(complex128, 1d, C)
    # cs = arg(2, name=cs) :: array(complex128, 1d, C)
    # output = arg(3, name=output) :: array(int32, 1d, C)
    # jump 2
    # label 2
    # $2.1 = global(range: <class 'range'>) :: Function(<class 'range'>)
...

```

Numba is a powerful JIT compiler that is now maturing. Do not expect magic on your first attempt—you may have to introspect the generated code to figure out how to make your code compile in `nopython` mode. Once you've solved this, you'll likely see good wins. Your best approach will be to break your current code into small (<10 line) and discrete functions and to tackle these one at a time. Do not try to throw a large function into Numba; you can debug the process far more quickly if you have only small, discrete chunks to review individually.

Numba to Compile NumPy for Pandas

In “[Pandas](#)” on page 146, we looked at solving the slope calculation task for 100,000 rows of data in a Pandas DataFrame using Ordinary Least Squares. We can make that approach an order of magnitude faster by using Numba.

We can take the `ols_lsqr_raw` function that we used before and, decorated with `numba.jit` as shown in [Example 7-16](#), can generate a compiled variant. Note the `nopython=True` argument—this forces Numba to raise an exception if we pass in a datatype that it doesn't understand, where it would otherwise fall back to a pure Python mode silently. We don't want it to run correctly but slowly on the wrong datatype if we pass in a Pandas Series; here we want to be informed that we've passed in the wrong data. In this edition, Numba can compile only NumPy datatypes, not Pandas types like Series.

Example 7-16. Solving Ordinary Least Squares with numpy on a Pandas DataFrame

```
def ols_lstsq_raw(row):
    """Variant of `ols_lstsq` where row is a numpy array (not a Series)"""
    X = np.arange(row.shape[0])
    ones = np.ones(row.shape[0])
    A = np.vstack((X, ones)).T
    m, c = np.linalg.lstsq(A, row, rcond=-1)[0]
    return m

# generate a Numba compiled variant
ols_lstsq_raw_values_numba = jit(ols_lstsq_raw, nopython=True)

results = df.apply(ols_lstsq_raw_values_numba, axis=1, raw=True)
```

The first time we call this function in, we get the expected short delay while the function is compiled; processing 100,000 rows takes 2.3 seconds including compilation time. Subsequent calls to process 100,000 rows are very fast—the noncompiled `ols_lstsq_raw` takes 5.3 seconds per 100,000 rows, whereas after using Numba it takes 0.58 seconds. That’s nearly a tenfold speedup!

PyPy

PyPy is an alternative implementation of the Python language that includes a tracing just-in-time compiler; it is compatible with Python 3.5+. Typically, it lags behind the most recent version of Python; at the time of writing this second edition Python 3.7 is standard, and PyPy supports up to Python 3.6.

PyPy is a drop-in replacement for CPython and offers all the built-in modules. The project comprises the RPython Translation Toolchain, which is used to build PyPy (and could be used to build other interpreters). The JIT compiler in PyPy is very effective, and good speedups can be seen with little or no work on your part. See “[PyPy for Successful Web and Data Processing Systems \(2014\)](#)” on page 426 for a large PyPy deployment success story.

PyPy runs our pure Python Julia demo without any modifications. With CPython it takes 8 seconds, and with PyPy it takes 0.9 seconds. This means that PyPy achieves a result that’s very close to the Cython example in [Example 7-8](#), without *any effort at all*—that’s pretty impressive! As we observed in our discussion of Numba, if the calculations are run again *in the same session*, then the second and subsequent runs are faster than the first one, as they are already compiled.

By expanding the math and removing the call to `abs`, the PyPy runtime drops to 0.2 seconds. This is equivalent to the Cython versions using pure Python and `numpy` without any work! Note that this result is true only if you’re not using `numpy` with PyPy.

The fact that PyPy supports all the built-in modules is interesting—this means that `multiprocessing` works as it does in CPython. If you have a problem that runs with the batteries-included modules and can run in parallel with `multiprocessing`, you can expect that all the speed gains you might hope to get will be available.

PyPy's speed has evolved over time. The older chart in [Figure 7-6](#) (from speed.pypy.org) will give you an idea about PyPy's maturity. These speed tests reflect a wide range of use cases, not just mathematical operations. It is clear that PyPy offers a faster experience than CPython.

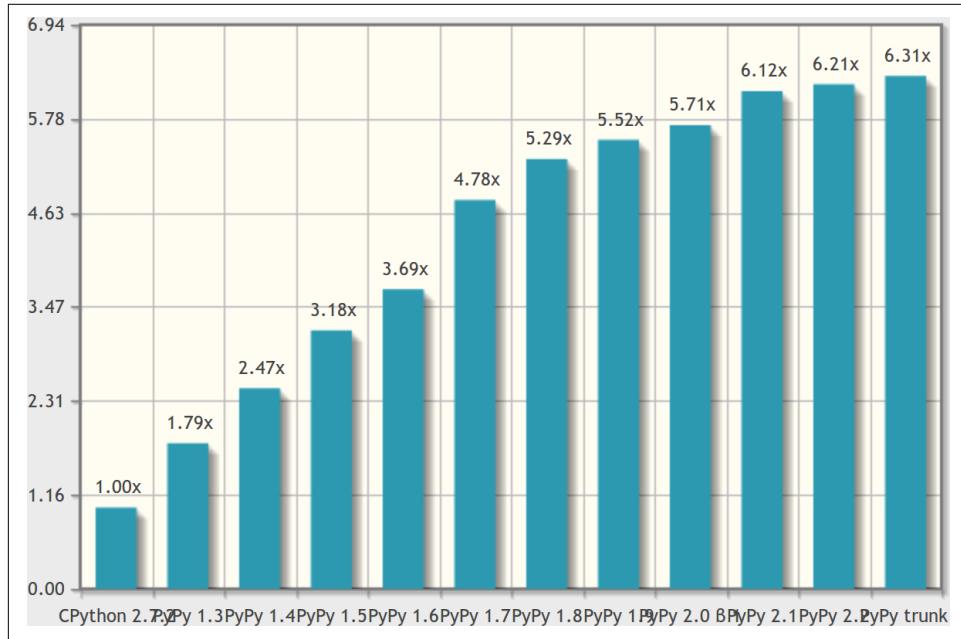


Figure 7-6. Each new version of PyPy offers speed improvements

Garbage Collection Differences

PyPy uses a different type of garbage collector than CPython, and this can cause some nonobvious behavior changes to your code. Whereas CPython uses reference counting, PyPy uses a modified mark-and-sweep approach that may clean up an unused object much later. Both are correct implementations of the Python specification; you just have to be aware that code modifications might be required when swapping.

Some coding approaches seen in CPython depend on the behavior of the reference counter—particularly the flushing of files, if you open and write to them without an explicit file close. With PyPy the same code will run, but the updates to the file might get flushed to disk later, when the garbage collector next runs. An alternative form that works in both PyPy and Python is to use a context manager using `with` to open

and automatically close files. The [Differences Between PyPy and CPython](#) page on the PyPy website lists the details.

Running PyPy and Installing Modules

If you've never run an alternative Python interpreter, you might benefit from a short example. Assuming you've downloaded and extracted PyPy, you'll now have a folder structure containing a `bin` directory. Run it as shown in [Example 7-17](#) to start PyPy.

Example 7-17. Running PyPy to see that it implements Python 3.6

```
...
$ pypy3
Python 3.6.1 (784b254d6699, Apr 14 2019, 10:22:42)
[PyPy 7.1.1-beta0 with GCC 6.2.0 20160901] on linux
Type "help", "copyright", "credits", or "license" for more information.
And now for something completely different
...
```

Note that PyPy 7.1 runs as Python 3.6. Now we need to set up `pip`, and we'll want to install IPython. The steps shown in [Example 7-18](#) are the same as you might have performed with CPython if you've installed `pip` without the help of an existing distribution or package manager. Note that when running IPython, we get the same build number as we see when running `pypy3` in the preceding example.

You can see that IPython runs with PyPy just the same as with CPython, and using the `%run` syntax, we execute the Julia script inside IPython to achieve 0.2-second runtimes.

Example 7-18. Installing pip for PyPy to install third-party modules like IPython

```
...
$ pypy3 -m ensurepip
Collecting setuptools
Collecting pip
  Installing collected packages: setuptools, pip
    Successfully installed pip-9.0.1 setuptools-28.8.0

$ pip3 install ipython
Collecting ipython

$ ipython
Python 3.6.1 (784b254d6699, Apr 14 2019, 10:22:42)
Type 'copyright', 'credits', or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run julia1_nopil_expanded_math.py
Length of x: 1000
```

```
Total elements: 1000000
calculate_z_serial_purepython took 0.2143106460571289 seconds
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 0.1965022087097168 seconds
...
```

Note that PyPy supports projects like `numpy` that require C bindings through the CPython extension compatibility layer `cpyext`, but it has an overhead of 4–6×, which generally makes `numpy` too slow. If your code is mostly pure Python with only a few calls to `numpy`, you may still see significant overall gains. If your code, like the Julia example, makes many calls to `numpy`, then it'll run significantly slower. The Julia benchmark here with `numpy` arrays runs 6× slower than when it is run with CPython.

If you need other packages, they should install thanks to the `cpyext` compatibility module, which is essentially PyPy's version of `python.h`. It handles the different memory management requirements of PyPy and CPython; however, this management incurs a cost of 4–6× per managed call, so the speed advantages of `numpy` can be negated by this overhead. A new project named `HPy` (formerly `PyHandle`) aims to remove this overhead by providing a higher-level object handle—one not tied to CPython's implementation—which can be shared with other projects like Cython.

If you want to understand PyPy's performance characteristics, look at the `vmprof` lightweight sampling profiler. It is thread-safe and supports a web-based user interface.

Another downside of PyPy is that it can use a lot of RAM. Each release is better in this respect, but in practice it may use more RAM than CPython. RAM is fairly cheap, though, so it makes sense to try to trade it for enhanced performance. Some users have also reported *lower* RAM usage when using PyPy. As ever, perform an experiment using representative data if this is important to you.

A Summary of Speed Improvements

To summarize the previous results, in [Table 7-1](#) we see that PyPy on a pure Python math-based code sample is approximately 9× faster than CPython with no code changes, and it's even faster if the `abs` line is simplified. Cython runs faster than PyPy in both instances but requires annotated code, which increases development and support effort.

Table 7-1. Julia (no numpy) results

Speed	
CPython	8.00s
Cython	0.49s
Cython on expanded math	0.19s
PyPy	0.90s
PyPy on expanded math	0.20s

The Julia solver with `numpy` enables the investigation of OpenMP. In [Table 7-2](#), we see that both Cython and Numba run faster than the non-`numpy` versions with expanded math. When we add OpenMP, both Cython and Numba provide further speedups for very little additional coding.

Table 7-2. Julia (with numpy and expanded math) results

Speed	
CPython	21.00s
Cython	0.18s
Cython and OpenMP “guided”	0.05s
Numba (2nd & subsequent runs)	0.17s
Numba and OpenMP	0.06s

For pure Python code, PyPy is an obvious first choice. For `numpy` code, Numba is a great first choice.

When to Use Each Technology

If you’re working on a numeric project, then each of these technologies could be useful to you. [Table 7-3](#) summarizes the main options.

Table 7-3. Compiler options

	Cython	Numba	PyPy
Mature	Y	Y	Y
Widespread	Y	–	–
<code>numpy</code> support	Y	Y	Y
Nonbreaking code changes	–	Y	Y
Needs C knowledge	Y	–	–
Supports OpenMP	Y	Y	–

Numba may offer quick wins for little effort, but it too has limitations that might stop it working well on your code. It is also a relatively young project.

Cython probably offers the best results for the widest set of problems, but it does require more effort and has an additional “support tax” due to mixing Python with C annotations.

PyPy is a strong option if you’re not using `numpy` or other hard-to-port C extensions.

If you’re deploying a production tool, you probably want to stick with well-understood tools—Cython should be your main choice, and you may want to check out “[Making Deep Learning Fly with RadimRehurek.com \(2014\)](#)” on page 415. PyPy is also being used in production settings (see “[PyPy for Successful Web and Data Processing Systems \(2014\)](#)” on page 426).

If you’re working with light numeric requirements, note that Cython’s buffer interface accepts `array.array` matrices—this is an easy way to pass a block of data to Cython for fast numeric processing without having to add `numpy` as a project dependency.

Overall, Numba is maturing and is a promising project, whereas Cython is mature. PyPy is regarded as being fairly mature now and should definitely be evaluated for long-running processes.

In a class run by Ian, a capable student implemented a C version of the Julia algorithm and was disappointed to see it execute more slowly than his Cython version. It transpired that he was using 32-bit floats on a 64-bit machine—these run more slowly than 64-bit doubles on a 64-bit machine. The student, despite being a good C programmer, didn’t know that this could involve a speed cost. He changed his code, and the C version, despite being significantly shorter than the autogenerated Cython version, ran at roughly the same speed. The act of writing the raw C version, comparing its speed, and figuring out how to fix it took longer than using Cython in the first place.

This is just an anecdote; we’re not suggesting that Cython will generate the best code, and competent C programmers can probably figure out how to make *their* code run faster than the version generated by Cython. It is worth noting, though, that the assumption that handwritten C will be faster than converted Python is not a safe assumption. You must always benchmark and make decisions using evidence. C compilers are pretty good at converting code into fairly efficient machine code, and Python is pretty good at letting you express your problem in an easy-to-understand language—combine these two powers sensibly.

Other Upcoming Projects

The [PyData compilers page](#) lists a set of high performance and compiler tools.

Pythran is an AOT compiler aimed at scientists who are using `numpy`. Using few annotations, it will compile Python numeric code to a faster binary—it produces speedups that are very similar to Cython but for much less work. Among other features, it always releases the GIL and can use both SIMD instructions and OpenMP. Like Numba, it doesn't support classes. If you have tight, locally bound loops in Numpy, Pythran is certainly worth evaluating. The associated FluidPython project aims to make Pythran even easier to write and provides JIT capability.

Transonic attempts to unify Cython, Pythran, and Numba, and potentially other compilers, behind one interface to enable quick evaluation of multiple compilers without having to rewrite code.

ShedSkin is an AOT compiler aimed at nonscientific, pure Python code. It has no `numpy` support, but if your code is pure Python, ShedSkin produces speedups similar to those seen by PyPy (without using `numpy`). It supports Python 2.7 with some Python 3.x support.

PyCUDA and **PyOpenCL** offer CUDA and OpenCL bindings into Python for direct access to GPUs. Both libraries are mature and support Python 3.4+.

Nuitka is a Python compiler that aims to be an alternative to the usual CPython interpreter, with the option of creating compiled executables. It supports all of Python 3.7, though in our testing it didn't produce any noticeable speed gains for our plain Python numerical tests.

Our community is rather blessed with a wide array of compilation options. While they all have trade-offs, they also offer a lot of power so that complex projects can take advantage of the full power of CPUs and multicore architectures.

Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) are becoming incredibly popular as a method to speed up arithmetic-heavy computational workloads. Originally designed to help handle the heavy linear algebra requirements of 3D graphics, GPUs are particularly well suited for solving easily parallelizable problems.

Interestingly, GPUs themselves are slower than most CPUs if we just look at clock speeds. This may seem counterintuitive, but as we discussed in “[Computing Units](#)” on page 2, clock speed is just one measurement of hardware’s ability to compute. GPUs excel at massively parallelize tasks because of the staggering number of compute cores they have. CPUs generally have on the order of 12 cores, while modern-day GPUs have thousands. For example, on the machine used to run bench-

marks for this section, the AMD Ryzen 7 1700 CPU has 8 cores, each at 3.2 GHz, while the NVIDIA RTX 2080 TI GPU has 4,352 cores, each at 1.35 GHz.¹

This incredible amount of parallelism can speed up many numerical tasks by a staggering amount. However, programming on these devices can be quite tough. Because of the amount of parallelism, data locality needs to be considered and can be make-or-break in terms of getting speedups. There are many tools out there to write native GPU code (also called *kernels*) in Python, such as **CuPy**. However, the needs of modern deep learning algorithms have been pushing new interfaces into GPUs that are easy and intuitive to use.

The two front-runners in terms of easy-to-use GPU mathematics libraries are TensorFlow and PyTorch. We will focus on PyTorch for its ease of use and great speed.²

Dynamic Graphs: PyTorch

PyTorch is a static computational graph tensor library that is particularly user-friendly and has a very intuitive API for anyone familiar with `numpy`. In addition, since it is a tensor library, it has all the same functionality as `numpy`, with the added advantages of being able to create functions through its static computational graph and calculate derivatives of those functions by using a mechanism called `autograd`.



The `autograd` functionality of PyTorch is left out since it isn't relevant to our discussion. However, this module is quite amazing and can take the derivative of any arbitrary function made up of PyTorch operations. It can do it on the fly and at any value. For a long time, taking derivatives of complex functions could have been the makings of a PhD thesis; however, now we can do it incredibly simply and efficiently. While this may also be off-topic for your work, we recommend learning about `autograd` and automatic-differentiation in general, as it truly is an incredible advancement in numerical computation.

By *static computational graph*, we mean that performing operations on PyTorch objects creates a dynamic definition of a program that gets compiled to GPU code in the background when it is executed (exactly like a JIT from “[JIT Versus AOT Compilers](#)” on page 164). Since it is dynamic, changes to the Python code automatically

¹ The RTX 2080 TI also includes 544 tensor cores specifically created to help with mathematical operations that are particularly useful for deep learning.

² For comparisons of the performance of TensorFlow and PyTorch, see <https://oreil.ly/8NOJW> and <https://oreil.ly/4BKMS>.

get reflected in changes in the GPU code without an explicit compilation step needed. This hugely aids debugging and interactivity, as compared to static graph libraries like TensorFlow.

In a static graph, like TensorFlow, we first set up our computation, then compile it. From then on, our compute is fixed in stone, and we can change it only by recompiling the entire thing. With the dynamic graph of PyTorch, we can conditionally change our compute graph or build it up iteratively. This allows us to have conditional debugging in our code or lets us play with the GPU in an interactive session in IPython. The ability to flexibly control the GPU is a complete game changer when dealing with complex GPU-based workloads.

As an example of the library's ease of use as well as its speed, in [Example 7-19](#) we port the numpy code from [Example 6-9](#) to use the GPU using PyTorch.

Example 7-19. PyTorch 2D diffusion

```
import torch
from torch import (roll, zeros) ①

grid_shape = (640, 640)

def laplacian(grid):
    return (
        roll(grid, +1, 0)
        + roll(grid, -1, 0)
        + roll(grid, +1, 1)
        + roll(grid, -1, 1)
        - 4 * grid
    )

def evolve(grid, dt, D=1):
    return grid + dt * D * laplacian(grid)

def run_experiment(num_iterations):
    grid = zeros(grid_shape)

    block_low = int(grid_shape[0] * 0.4)
    block_high = int(grid_shape[0] * 0.5)
    grid[block_low:block_high, block_low:block_high] = 0.005

    grid = grid.cuda() ②
    for i in range(num_iterations):
        grid = evolve(grid, 0.1)
    return grid
```

①, ② The only changes needed.

Most of the work is done in the modified import, where we changed `numpy` to `torch`. In fact, if we just wanted to run our optimized code on the CPU, we could stop here.³ To use the GPU, we simply need to move our data to the GPU, and then `torch` will automatically compile all computations we do on that data into GPU code.

As we can see in [Figure 7-7](#), this small code change has given us an incredible speedup.⁴ For a 512×512 grid, we have a $5.3\times$ speedup, and for a $4,096 \times 4,096$ grid we have a $102\times$ speedup! It is interesting that the GPU code doesn't seem to be as affected by increases to grid size as the `numpy` code is.

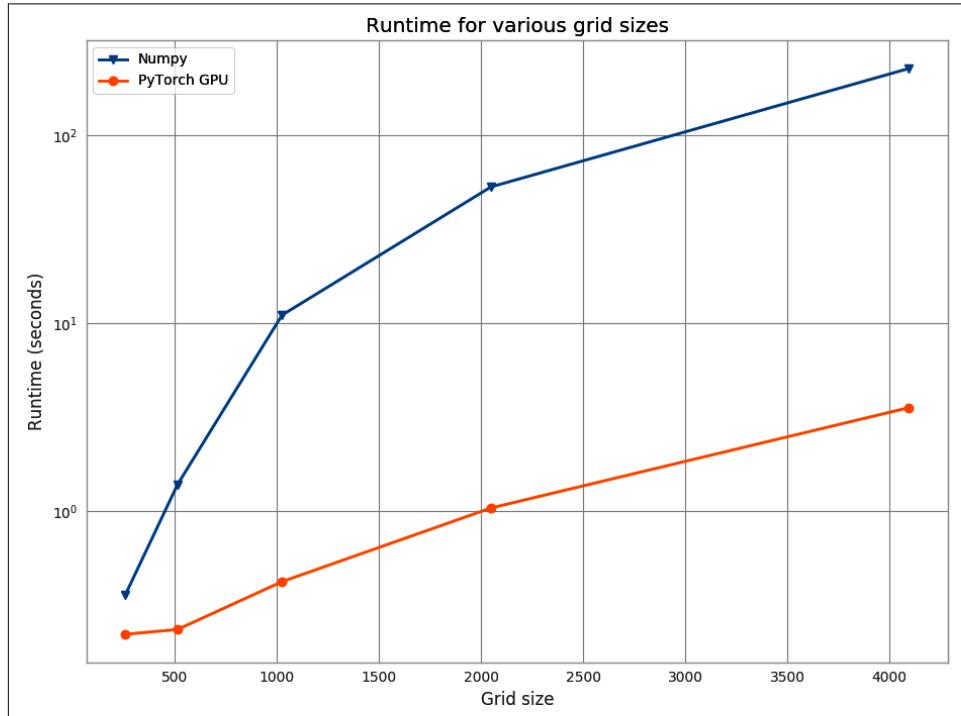


Figure 7-7. PyTorch versus numpy performance

This speedup is a result of how parallelizable the diffusion problem is. As we said before, the GPU we are using has 4,362 independent computation cores. It seems that

³ PyTorch CPU performance isn't terribly great unless you install from source. When installing from source, optimized linear algebra libraries are used to give speeds comparable to NumPy.

⁴ As with any JIT, the first time a function is called, it will have the overhead of having to compile the code. In [Example 7-19](#), we profile the functions multiple times and ignore the first time in order to measure only the runtime speeds.

once the diffusion problem is parallelized, none of these GPU cores are being fully utilized.



When timing the performance of GPU code, it is important to set the environmental flag `CUDA_LAUNCH_BLOCKING=1`. By default, GPU operations are run asynchronously to allow more operations to be pipelined together and thus to minimize the total utilization of the GPU and increase parallelization. When the asynchronous behavior is enabled, we can guarantee that the computations are done only when either the data is copied to another device or a `torch.cuda.synchronize()` command is issued. By enabling the preceding environmental variable, we can make sure that computations are completed when they are issued and that we are indeed measuring compute time.

Basic GPU Profiling

One way to verify exactly how much of the GPU we are utilizing is by using the `nvidia-smi` command to inspect the resource utilization of the GPU. The two values we are most interested in are the power usage and the GPU utilization:

```
$ nvidia-smi
+-----+
| NVIDIA-SMI 440.44      Driver Version: 440.44      CUDA Version: 10.2 |
|-----+-----+-----+-----+-----+-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|-----+-----+-----+-----+-----+-----+-----+-----+
|  0  GeForce RTX 208... Off  | 00000000:06:00.0 Off |                  N/A |
| 30%   58C    P2    96W / 260W |   1200MiB / 11018MiB |    95%     Default |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| Processes:                               GPU Memory |
| GPU  PID  Type  Process name          Usage        |
|-----+-----+-----+-----+
|  0    26329  C    .../.pyenv/versions/3.7.2/bin/python  1189MiB |
+-----+
```

GPU utilization, here at 95%, is a slightly mislabeled field. It tells us what percentage of the last second has been spent running at least one kernel. So it isn't telling us what percentage of the GPU's total computational power we're using but rather how much time was spent *not* being idle. This is a very useful measurement to look at when debugging memory transfer issues and making sure that the CPU is providing the GPU with enough work.

Power usage, on the other hand, is a good proxy for judging how much of the GPU's compute power is being used. As a rule of thumb, the more power the GPU is

drawing, the more compute it is currently doing. If the GPU is waiting for data from the CPU or using only half of the available cores, power use will be reduced from the maximum.

Another useful tool is [gpustat](#). This project provides a nice view into many of NVIDIA's stats using a much friendlier interface than `nvidia-smi`.

To help understand what specifically is causing slowdowns in your PyTorch code, the project provides a special profiling tool. Running your code with `python -m torch.utils.bottleneck` will show both CPU and GPU runtime stats to help you identify potential optimizations to either portion of your code.

Performance Considerations of GPUs

Since a GPU is a completely auxiliary piece of hardware on the computer, with its own architecture as compared with the CPU, there are many GPU-specific performance considerations to keep in mind.

The biggest speed consideration for GPUs is the transfer time of data from the system memory to the GPU memory. When we use `tensor.to(DEVICE)`, we are triggering a transfer of data that may take some time depending on the speed of the GPU's bus and the amount of data being transferred.

Other operations may trigger a transfer as well. In particular, `tensor.items()` and `tensor.tolist()` often cause problems when introduced for debugging purposes. In fact, running `tensor.numpy()` to convert a PyTorch tensor into a `numpy` array specifically requires an explicit copy out of the GPU, which ensures you understand the potential penalty.

As an example, let's add a `grid.cpu()` call inside the solver loop of our diffusion code:

```
grid = grid.to(device)
for i in range(num_iterations):
    grid = evolve(grid, 0.1)
    grid.cpu()
```

To ensure that we have a fair comparison, we will also add `torch.cuda.synchronize()` to the control code so that we are simply testing the time to copy the data from the CPU. In addition to this slowing down your code by triggering a data transfer from the GPU to the system memory, your code will slow down because the GPU will pause code execution that would have continued in the background until the transfer is complete.

This change to the code for a $2,048 \times 2,048$ grid slows down our code by $2.54\times!$ Even though our GPU has an advertised bandwidth of 616.0 GB/s, this overhead can quickly add up. In addition, other overhead costs are associated with a memory copy.

First, we are creating a hard stop to any potential pipelining of our code execution. Then, because we are no longer pipelining, our data on the GPU must all be synchronized out of the memory of the individual CUDA cores. Finally, space on system memory needs to be prepared to receive the new data from the GPU.

While this seems like a ridiculous addition to make to our code, this sort of thing happens all the time. In fact, one of the biggest things slowing down PyTorch code when it comes to deep learning is copying training data from the host into the GPU. Often the training data is simply too big to fit on the GPU, and doing these constant data transfers is an unavoidable penalty.

There are ways to alleviate the overhead from this inevitable data transfer when the problem is going from CPU to GPU. First, the memory region can be marked as `pin` ned. This can be done by calling the `Tensor.pin_memory()` method, which returns a copy of the CPU tensor that is copied to a “page locked” region of memory. This page-locked region can be copied to the GPU much more quickly, and it can be copied asynchronously so as to not disturb any computations being done by the GPU. While training a deep learning model, data loading is generally done with the `DataLoader` class, which conveniently has a `pin_memory` parameter that can automatically do this for all your training data.⁵

The most important step is to profile your code by using the tools outlined in “[Basic GPU Profiling](#)” on page 193. When your code is spending most of its time doing data transfers, you will see a low power draw, a smaller GPU utilization (as reported by `nvidia-smi`), and most of your time being spent in the `to` function (as reported by `bottleneck`). Ideally, you will be using the maximum amount of power the GPU can support and have 100% utilization. This is possible even when large amounts of data transfer are required—even when training deep learning models with a large number of images!



GPUs are not particularly good at running multiple tasks at the same time. When starting up a task that requires heavy use of the GPU, ensure that no other tasks are utilizing it by running `nvidia-smi`. However, if you are running a graphical environment, you may have no choice but to have your desktop and GPU code use the GPU at the same time.

⁵ The `DataLoader` object also supports running with multiple workers. Having several workers is recommended if data is being loaded from disk in order to minimize I/O time.

When to Use GPUs

We've seen that GPUs can be incredibly quick; however, memory considerations can be quite devastating to this runtime. This seems to indicate that if your task requires mainly linear algebra and matrix manipulations (like multiplication, addition, and Fourier transforms), then GPUs are a fantastic tool. This is particularly true if the calculation can happen on the GPU uninterrupted for a period of time before being copied back into system memory.

As an example of a task that requires a lot of branching, we can imagine code where every step of the computation requires the previous result. If we compare running [Example 7-20](#) using PyTorch versus using NumPy, we see that NumPy is consistently faster (98% faster for the included example!). This makes sense given the architecture of the GPU. While the GPU can run many more tasks at once than the CPU can, each of those tasks runs more slowly on the GPU than on the CPU. This example task can run only one computation at a time, so having many compute cores doesn't help at all; it's better to simply have one very fast core.

Example 7-20. Highly branching task

```
import torch

def task(A, target):
    """
    Given an int array of length N with values from (0, N] and a target value,
    iterates through the array, using the current value to find the next array
    item to look at, until we have seen a total value of at least `target`.
    Returns how many iterations until the value was reached.
    """

    result = 0
    i = 0
    N = 0
    while result < target:
        r = A[i]
        result += r
        i = A[i]
        N += 1
    return N

if __name__ == "__main__":
    N = 1000

    A_py = (torch.rand(N) * N).type(torch.int).to('cuda:0')
    A_np = A_py.cpu().numpy()

    task(A_py, 500)
    task(A_np, 500)
```

In addition, because of the limited memory of the GPU, it is not a good tool for tasks that require exceedingly large amounts of data, many conditional manipulations of the data, or changing data. Most GPUs made for computational tasks have around 12 GB of memory, which puts a significant limitation on “large amounts of data.” However, as technology improves, the size of GPU memory increases, so hopefully this limitation becomes less drastic in the future.

The general recipe for evaluating whether to use the GPU consists of the following steps:

1. Ensure that the memory use of the problem will fit within the GPU (in “[Using memory_profiler to Diagnose Memory Usage](#)” on page 46, we explore profiling memory use).
2. Evaluate whether the algorithm requires a lot of branching conditions versus vectorized operations. As a rule of thumb, `numpy` functions generally vectorize very well, so if your algorithm can be written in terms of `numpy` calls, your code probably will vectorize well! You can also check the `branches` result when running `perf` (as explained in “[Understanding perf](#)” on page 122).
3. Evaluate how much data needs to be moved between the GPU and the CPU. Some questions to ask here are “How much computation can I do before I need to plot/save results?” and “Are there times my code will have to copy the data to run in a library I know isn’t GPU-compatible?”
4. Make sure PyTorch supports the operations you’d like to do! PyTorch implements a large portion of the `numpy` API, so this shouldn’t be an issue. For the most part, the API is even the same, so you don’t need to change your code at all. However, in some cases either PyTorch doesn’t support an operation (such as dealing with complex numbers) or the API is slightly different (for example, with generating random numbers).

Considering these four points will help give you confidence that a GPU approach would be worthwhile. There are no hard rules for when the GPU will work better than the CPU, but these questions will help you gain some intuition. However, PyTorch also makes converting code to use the GPU painless, so the barrier to entry is quite low, even if you are just evaluating the GPU’s performance.

Foreign Function Interfaces

Sometimes the automatic solutions just don’t cut it, and you need to write custom C or Fortran code yourself. This could be because the compilation methods don’t find some potential optimizations, or because you want to take advantage of libraries or language features that aren’t available in Python. In all of these cases, you’ll need to

use foreign function interfaces, which give you access to code written and compiled in another language.

For the rest of this chapter, we will attempt to use an external library to solve the 2D diffusion equation in the same way we did in [Chapter 6](#).⁶ The code for this library, shown in [Example 7-21](#), could be representative of a library you've installed or code that you have written. The methods we'll look at serve as great ways to take small parts of your code and move them to another language in order to do very targeted language-based optimizations.

Example 7-21. Sample C code for solving the 2D diffusion problem

```
void evolve(double in[][512], double out[][512], double D, double dt) {
    int i, j;
    double laplacian;
    for (i=1; i<511; i++) {
        for (j=1; j<511; j++) {
            laplacian = in[i+1][j] + in[i-1][j] + in[i][j+1] + in[i][j-1] \
                        - 4 * in[i][j];
            out[i][j] = in[i][j] + D * dt * laplacian;
        }
    }
}
```



We fix the grid size to be 512×512 in order to simplify the example code. To accept an arbitrarily sized grid, you must pass in the `in` and `out` parameters as double pointers and include function arguments for the actual size of the grid.

To use this code, we must compile it into a shared module that creates a `.so` file. We can do this using `gcc` (or any other C compiler) by following these steps:

```
$ gcc -O3 -std=gnu11 -c diffusion.c
$ gcc -shared -o diffusion.so diffusion.o
```

We can place this final shared library file anywhere that is accessible to our Python code, but standard *nix organization stores shared libraries in `/usr/lib` and `/usr/local/lib`.

⁶ For simplicity, we will not implement the boundary conditions.

ctypes

The most basic foreign function interface in CPython is through the `ctypes` module.⁷ The bare-bones nature of this module can be quite inhibitive at times—you are in charge of doing everything, and it can take quite a while to make sure that you have everything in order. This extra level of complexity is evident in our `ctypes` diffusion code, shown in Example 7-22.

Example 7-22. `ctypes` 2D diffusion code

```
import ctypes

grid_shape = (512, 512)
_diffusion = ctypes.CDLL("diffusion.so") ❶

# Create references to the C types that we will need to simplify future code
TYPE_INT = ctypes.c_int
TYPE_DOUBLE = ctypes.c_double
TYPE_DOUBLE_SS = ctypes.POINTER(ctypes.POINTER(ctypes.c_double))

# Initialize the signature of the evolve function to:
# void evolve(int, int, double**, double**, double, double)
_diffusion.evolve.argtypes = [TYPE_DOUBLE_SS, TYPE_DOUBLE_SS, TYPE_DOUBLE,
                               TYPE_DOUBLE]
_diffusion.evolve.restype = None

def evolve(grid, out, dt, D=1.0):
    # First we convert the Python types into the relevant C types
    assert grid.shape == (512, 512)
    cdt = TYPE_DOUBLE(dt)
    cD = TYPE_DOUBLE(D)
    pointer_grid = grid.ctypes.data_as(TYPE_DOUBLE_SS) ❷
    pointer_out = out.ctypes.data_as(TYPE_DOUBLE_SS)

    # Now we can call the function
    _diffusion.evolve(pointer_grid, pointer_out, cD, cdt) ❸
```

- ❶ This is similar to importing the `diffusion.so` library. Either this file is in one of the standard system paths for library files or we can enter an absolute path.
- ❷ `grid` and `out` are both `numpy` arrays.
- ❸ We finally have all the setup necessary and can call the C function directly.

⁷ This is CPython-dependent. Other versions of Python may have their own versions of `ctypes`, which may work differently.

This first thing we do is “import” our shared library. This is done with the `ctypes.CDLL` call. In this line, we can specify any shared library that Python can access (for example, the `ctypes-opencv` module loads the `libcv.so` library). From this, we get a `_diffusion` object that contains all the members that the shared library contains. In this example, `diffusion.so` contains only one function, `evolve`, which is now made available to us as a property of the `_diffusion` object. If `diffusion.so` had many functions and properties, we could access them all through the `_diffusion` object.

However, even though the `_diffusion` object has the `evolve` function available within it, Python doesn’t know how to use it. C is statically typed, and the function has a very specific signature. To properly work with the `evolve` function, we must explicitly set the input argument types and the return type. This can become quite tedious when developing libraries in tandem with the Python interface, or when dealing with a quickly changing library. Furthermore, since `ctypes` can’t check if you have given it the correct types, your code may silently fail or segfault if you make a mistake!

Furthermore, in addition to setting the arguments and return type of the function object, we also need to convert any data we care to use with it (this is called *casting*). Every argument we send to the function must be carefully casted into a native C type. Sometimes this can get quite tricky, since Python is very relaxed about its variable types. For example, if we had `num1 = 1e5`, we would have to know that this is a Python `float`, and thus we should use a `ctype.c_float`. On the other hand, for `num2 = 1e300`, we would have to use `ctype.c_double`, because it would overflow a standard C `float`.

That being said, `numpy` provides a `.ctypes` property to its arrays that makes it easily compatible with `ctypes`. If `numpy` didn’t provide this functionality, we would have had to initialize a `ctypes` array of the correct type and then find the location of our original data and have our new `ctypes` object point there.



Unless the object you are turning into a `ctype` object implements a buffer (as do the `array` module, `numpy` arrays, `io.StringIO`, etc.), your data will be copied into the new object. In the case of casting an `int` to a `float`, this doesn’t mean much for the performance of your code. However, if you are casting a very long Python list, this can incur quite a penalty! In these cases, using the `array` module or a `numpy` array, or even building up your own buffered object using the `struct` module, would help. This does, however, hurt the readability of your code, since these objects are generally less flexible than their native Python counterparts.

This memory management can get even more complicated if you have to send the library a complicated data structure. For example, if your library expects a C struct representing a point in space with the properties `x` and `y`, you would have to define the following:

```
from ctypes import Structure

class cPoint(Structure):
    _fields_ = ("x", c_int), ("y", c_int)
```

At this point you could start creating C-compatible objects by initializing a `cPoint` object (i.e., `point = cPoint(10, 5)`). This isn't a terrible amount of work, but it can become tedious and results in some fragile code. What happens if a new version of the library is released that slightly changes the structure? This will make your code very hard to maintain and generally results in stagnant code, where the developers simply decide never to upgrade the underlying libraries that are being used.

For these reasons, using the `ctypes` module is great if you already have a good understanding of C and want to be able to tune every aspect of the interface. It has great portability since it is part of the standard library, and if your task is simple, it provides simple solutions. Just be careful because the complexity of `ctypes` solutions (and similar low-level solutions) quickly becomes unmanageable.

ffi

Realizing that `ctypes` can be quite cumbersome to use at times, `ffi` attempts to simplify many of the standard operations that programmers use. It does this by having an internal C parser that can understand function and structure definitions.

As a result, we can simply write the C code that defines the structure of the library we wish to use, and then `ffi` will do all the heavy work for us: it imports the module and makes sure we specify the correct types to the resulting functions. In fact, this work can be almost trivial if the source for the library is available, since the header files (the files ending in `.h`) will include all the relevant definitions we need.⁸ [Example 7-23](#) shows the `ffi` version of the 2D diffusion code.

Example 7-23. ffi 2D diffusion code

```
from ffi import FFI, verifier

grid_shape = (512, 512)

ffi = FFI()
```

⁸ In Unix systems, header files for system libraries can be found in `/usr/include`.

```

ffi.cdef(
    "void evolve(double **in, double **out, double D, double dt);"  
①
)
lib = ffi.dlopen("../diffusion.so")

def evolve(grid, dt, out, D=1.0):
    pointer_grid = ffi.cast("double**", grid.ctypes.data) ②
    pointer_out = ffi.cast("double**", out.ctypes.data)
    lib.evolve(pointer_grid, pointer_out, D, dt)

```

- ➊ The contents of this definition can normally be acquired from the manual of the library that you are using or by looking at the library's header files.
- ➋ While we still need to cast nonnative Python objects for use with our C module, the syntax is very familiar to those with experience in C.

In the preceding code, we can think of the `cffi` initialization as being two-stepped. First, we create an FFI object and give it all the global C declarations we need. This can include datatypes in addition to function signatures. These signatures don't necessarily contain any code; they simply need to define what the code will look like. Then we can import a shared library containing the actual implementation of the functions by using `dlopen`. This means we could have told FFI about the function signature for the `evolve` function and then loaded up two different implantations and stored them in different objects (which is fantastic for debugging and profiling!).

In addition to easily importing a shared C library, `cffi` allows you to write C code and have it be dynamically compiled using the `verify` function. This has many immediate benefits—for example, you can easily rewrite small portions of your code to be in C without invoking the large machinery of a separate C library. Alternatively, if there is a library you wish to use, but some glue code in C is required to have the interface work perfectly, you can inline it into your `cffi` code, as shown in [Example 7-24](#), to have everything be in a centralized location. In addition, since the code is being dynamically compiled, you can specify compile instructions to every chunk of code you need to compile. Note, however, that this compilation has a one-time penalty every time the `verify` function is run to actually perform the compilation.

Example 7-24. cffi with inline 2D diffusion code

```

ffi = FFI()
ffi.cdef(
    "void evolve(double **in, double **out, double D, double dt);"
)
lib = ffi.verify(
    """

```

```

void evolve(double in[][512], double out[][512], double D, double dt) {
    int i, j;
    double laplacian;
    for (i=1; i<511; i++) {
        for (j=1; j<511; j++) {
            laplacian = in[i+1][j] + in[i-1][j] + in[i][j+1] + in[i][j-1] \
                        - 4 * in[i][j];
            out[i][j] = in[i][j] + D * dt * laplacian;
        }
    }
}
"""
extra_compile_args=[ "-O3"], ❶
)

```

- ❶ Since we are just-in-time compiling this code, we can also provide relevant compilation flags.

Another benefit of the `verify` functionality is that it plays nicely with complicated `cdef` statements. For example, if we were using a library with a complicated structure but wanted to use only a part of it, we could use the partial struct definition. To do this, we add a `...` in the struct definition in `ffi.cdef` and `#include` the relevant header file in a later `verify`.

For example, suppose we were working with a library with header `complicated.h` that included a structure that looked like this:

```

struct Point {
    double x;
    double y;
    bool isActive;
    char *id;
    int num_times_visited;
}

```

If we cared only about the `x` and `y` properties, we could write some simple `cffi` code that cares only about those values:

```

from cffi import FFI

ffi = FFI()
ffi.cdef(r"""
    struct Point {
        double x;
        double y;
        ...;
    };
    struct Point do_calculation();
""")
lib = ffi.verify(r"""

```

```
#include <complicated.h>
""")
```

We could then run the `do_calculation` function from the `complicated.h` library and have returned to us a `Point` object with its `x` and `y` properties accessible. This is amazing for portability, since this code will run just fine on systems with a different implementation of `Point` or when new versions of `complicated.h` come out, as long as they all have the `x` and `y` properties.

All of these niceties really make `cffi` an amazing tool to have when you're working with C code in Python. It is much simpler than `ctypes`, while still giving you the same amount of fine-grained control you may want when working directly with a foreign function interface.

f2py

For many scientific applications, Fortran is still the gold standard. While its days of being a general-purpose language are over, it still has many niceties that make vector operations easy to write and quite quick. In addition, many performance math libraries are written in Fortran ([LAPACK](#), [BLAS](#), etc.), all of which are fundamental in libraries such as SciPy, and being able to use them in your performance Python code may be critical.

For such situations, `f2py` provides a dead-simple way of importing Fortran code into Python. This module is able to be so simple because of the explicitness of types in Fortran. Since the types can be easily parsed and understood, `f2py` can easily make a CPython module that uses the native foreign function support within C to use the Fortran code. This means that when you are using `f2py`, you are actually autogenerated a C module that knows how to use Fortran code! As a result, a lot of the confusion inherent in the `ctypes` and `cffi` solutions simply doesn't exist.

In [Example 7-25](#), we can see some simple `f2py`-compatible code for solving the diffusion equation. In fact, all native Fortran code is `f2py`-compatible; however, the annotations to the function arguments (the statements prefaced by `!f2py`) simplify the resulting Python module and make for an easier-to-use interface. The annotations implicitly tell `f2py` whether we intend for an argument to be only an output or only an input, or to be something we want to modify in place or hidden completely. The `hidden` type is particularly useful for the sizes of vectors: while Fortran may need those numbers explicitly, our Python code already has this information on hand. When we set the type as “`hidden`,” `f2py` can automatically fill those values for us, essentially keeping them hidden from us in the final Python interface.

Example 7-25. Fortran 2D diffusion code with f2py annotations

```
SUBROUTINE evolve(grid, next_grid, D, dt, N, M)
    !f2py threadsafe
    !f2py intent(in) grid
    !f2py intent(inplace) next_grid
    !f2py intent(in) D
    !f2py intent(in) dt
    !f2py intent(hide) N
    !f2py intent(hide) M
    INTEGER :: N, M
    DOUBLE PRECISION, DIMENSION(N,M) :: grid, next_grid
    DOUBLE PRECISION, DIMENSION(N-2, M-2) :: laplacian
    DOUBLE PRECISION :: D, dt

    laplacian = grid(3:N, 2:M-1) + grid(1:N-2, 2:M-1) + &
                grid(2:N-1, 3:M) + grid(2:N-1, 1:M-2) - 4 * grid(2:N-1, 2:M-1)
    next_grid(2:N-1, 2:M-1) = grid(2:N-1, 2:M-1) + D * dt * laplacian
END SUBROUTINE evolve
```

To build the code into a Python module, we run the following command:

```
$ f2py -c -m diffusion --fcompiler=gfortran --opt='-O3' diffusion.f90
```



We specifically use `gfortran` in the preceding call to `f2py`. Make sure it is installed on your system or that you change the corresponding argument to use the Fortran compiler you have installed.

This will create a library file pinned to your Python version and operating system (`diffusion.cpython-37m-x86_64-linux-gnu.so`, in our case) that can be imported directly into Python.

If we play around with the resulting module interactively, we can see the niceties that `f2py` has given us, thanks to our annotations and its ability to parse the Fortran code:

```
>>> import diffusion

>>> diffusion?
Type:           module
String form:   <module 'diffusion' from '[...]cpython-37m-x86_64-linux-gnu.so'>
File:          [..cut..]/diffusion.cpython-37m-x86_64-linux-gnu.so
Docstring:
This module 'diffusion' is auto-generated with f2py (version:2).
Functions:
    evolve(grid,scratch,d,dt)
    .

>>> diffusion.evolve?
Call signature: diffusion.evolve(*args, **kwargs)
```

```

Type:          fortran
String form:  <fortran object>
Docstring:
evolve(grid,scratch,d,dt)

Wrapper for ``evolve``.

Parameters
grid : input rank-2 array('d') with bounds (n,m)
scratch : rank-2 array('d') with bounds (n,m)
d : input float
dt : input float

```

This code shows that the result from the f2py generation is automatically documented, and the interface is quite simplified. For example, instead of us having to extract the sizes of the vectors, f2py has figured out how to automatically find this information and simply hides it in the resulting interface. In fact, the resulting `evolve` function looks exactly the same in its signature as the pure Python version we wrote in [Example 6-14](#).

The only thing we must be careful of is the ordering of the `numpy` arrays in memory. Since most of what we do with `numpy` and Python focuses on code derived from C, we always use the C convention for ordering data in memory (called *row-major ordering*). Fortran uses a different convention (*column-major ordering*) that we must make sure our vectors abide by. These orderings simply state whether, for a 2D array, columns or rows are contiguous in memory.⁹ Luckily, this simply means we specify the `order='F'` parameter to `numpy` when declaring our vectors.



The difference in ordering basically changes which is the outer loop when iterating over a multidimensional array. In Python and C, if you define an array as `array[X][Y]`, your outer loop will be over X and your inner loop will be over Y. In `fortran`, your outer loop will be over Y and your inner loop will be over X. If you use the wrong loop ordering, you will at best suffer a major performance penalty because of an increase in `cache-misses` (see “[Memory Fragmentation](#)” on page 120) and at worst access the wrong data!

⁹ For more information, see the [Wikipedia page](#) on row-major and column-major ordering.

This results in the following code to use our Fortran subroutine. This code looks exactly the same as what we used in [Example 6-14](#), except for the import from the f2py-derived library and the explicit Fortran ordering of our data:

```
from diffusion import evolve

def run_experiment(num_iterations):
    scratch = np.zeros(grid_shape, dtype=np.double, order="F") ①
    grid = np.zeros(grid_shape, dtype=np.double, order="F")

    initialize_grid(grid)

    for i in range(num_iterations):
        evolve(grid, scratch, 1.0, 0.1)
        grid, scratch = scratch, grid
```

- ① Fortran orders numbers differently in memory, so we must remember to set our numpy arrays to use that standard.

C^{Python} Module

Finally, we can always go right down to the CPython API level and write a CPython module. This requires us to write code in the same way that CPython is developed and take care of all of the interactions between our code and the implementation of CPython.

This has the advantage that it is incredibly portable, depending on the Python version. We don't require any external modules or libraries, just a C compiler and Python! However, this doesn't necessarily scale well to new versions of Python. For example, CPython modules written for Python 2.7 won't work with Python 3, and vice versa.



In fact, much of the slowdown in the Python 3 rollout was rooted in the difficulty in making this change. When creating a CPython module, you are coupled very closely to the actual Python implementation, and large changes in the language (such as the change from 2.7 to 3) require large modifications to your module.

That portability comes at a big cost, though—you are responsible for every aspect of the interface between your Python code and the module. This can make even the simplest tasks take dozens of lines of code. For example, to interface with the diffusion library from [Example 7-21](#), we must write 28 lines of code simply to read the arguments to a function and parse them ([Example 7-26](#)). Of course, this does mean that you have incredibly fine-grained control over what is happening. This goes all the way down to being able to manually change the reference counts for Python's garbage collection (which can be the cause of a lot of pain when creating CPython

modules that deal with native Python types). Because of this, the resulting code tends to be minutely faster than other interface methods.



All in all, this method should be left as a last resort. While it is quite informative to write a CPython module, the resulting code is not as reusable or maintainable as other potential methods. Making subtle changes in the module can often require completely reworking it. In fact, we include the module code and the required `setup.py` to compile it ([Example 7-27](#)) as a cautionary tale.

Example 7-26. CPython module to interface to the 2D diffusion library

```
// python_interface.c
// - cpython module interface for diffusion.c

#define NPY_NO_DEPRECATED_API      NPY_1_7_API_VERSION

#include <Python.h>
#include <numpy/arrayobject.h>
#include "diffusion.h"

/* Docstrings */
static char module_docstring[] =
    "Provides optimized method to solve the diffusion equation";
static char cdiffusion_evolve_docstring[] =
    "Evolve a 2D grid using the diffusion equation";

PyArrayObject *py_evolve(PyObject *, PyObject *);

/* Module specification */
static PyMethodDef module_methods[] =
{
    /* { method name , C function           , argument types , docstring      } */
    { "evolve", (PyCFunction)py_evolve, METH_VARARGS, cdiffusion_evolve_docstring },
    { NULL,     NULL,                 0, NULL }
};

static struct PyModuleDef cdiffusionmodule =
{
    PyModuleDef_HEAD_INIT,
    "cdiffusion",        /* name of module */
    module_docstring,    /* module documentation, may be NULL */
    -1,                  /* size of per-interpreter state of the module,
                           * or -1 if the module keeps state in global variables. */
    module_methods
};

PyArrayObject *py_evolve(PyObject *self, PyObject *args)
{
    PyArrayObject *data;
```

```

PyArrayObject *next_grid;
double          dt, D = 1.0;

/* The "evolve" function will have the signature:
 *   evolve(data, next_grid, dt, D=1)
 */
if (!PyArg_ParseTuple(args, "O0d|d", &data, &next_grid, &dt, &D))
{
    PyErr_SetString(PyExc_RuntimeError, "Invalid arguments");
    return(NULL);
}

/* Make sure that the numpy arrays are contiguous in memory */
if (!PyArray_Check(data) || !PyArray_ISCONTIGUOUS(data))
{
    PyErr_SetString(PyExc_RuntimeError, "data is not a contiguous array.");
    return(NULL);
}
if (!PyArray_Check(next_grid) || !PyArray_ISCONTIGUOUS(next_grid))
{
    PyErr_SetString(PyExc_RuntimeError, "next_grid is not a contiguous array.");
    return(NULL);
}

/* Make sure that grid and next_grid are of the same type and have the same
 * dimensions
 */
if (PyArray_TYPE(data) != PyArray_TYPE(next_grid))
{
    PyErr_SetString(PyExc_RuntimeError,
                   "next_grid and data should have same type.");
    return(NULL);
}
if (PyArray_NDIM(data) != 2)
{
    PyErr_SetString(PyExc_RuntimeError, "data should be two dimensional");
    return(NULL);
}
if (PyArray_NDIM(next_grid) != 2)
{
    PyErr_SetString(PyExc_RuntimeError, "next_grid should be two dimensional");
    return(NULL);
}
if ((PyArray_DIM(data, 0) != PyArray_DIM(next_grid, 0)) ||
      (PyArray_DIM(data, 1) != PyArray_DIM(next_grid, 1)))
{
    PyErr_SetString(PyExc_RuntimeError,
                   "data and next_grid must have the same dimensions");
    return(NULL);
}

evolve(


```

```

    PyArray_DATA(data),
    PyArray_DATA(next_grid),
    D,
    dt
);

Py_XINCREF(next_grid);
return(next_grid);
}

/* Initialize the module */
PyMODINIT_FUNC
PyInit_cdiffusion(void)
{
    PyObject *m;

    m = PyModule_Create(&cdiffusionmodule);
    if (m == NULL)
    {
        return(NULL);
    }

    /* Load `numpy` functionality. */
    import_array();

    return(m);
}

```

To build this code, we need to create a `setup.py` script that uses the `distutils` module to figure out how to build the code such that it is Python-compatible ([Example 7-27](#)). In addition to the standard `distutils` module, `numpy` provides its own module to help with adding `numpy` integration in your CPython modules.

Example 7-27. Setup file for the CPython module diffusion interface

```

"""
setup.py for cpython diffusion module.  The extension can be built by running
$ python setup.py build_ext --inplace
which will create the __cdiffusion.so__ file, which can be directly imported into
Python.
"""

from distutils.core import setup, Extension
import numpy.distutils.misc_util

__version__ = "0.1"

cdiffusion = Extension(
    'cdiffusion',

```

```

sources = ['cdiffusion/cdiffusion.c', 'cdiffusion/python_interface.c'],
extra_compile_args = ["-O3", "-std=c11", "-Wall", "-p", "-pg", ],
extra_link_args = ["-lc"],
)

setup (
    name = 'diffusion',
    version = __version__,
    ext_modules = [cdiffusion,],
    packages = ["diffusion", ],
    include_dirs = numpy.distutils.misc_util.get_numpy_include_dirs(),
)

```

The result from this is a *cdiffusion.so* file that can be imported directly from Python and used quite easily. Since we had complete control over the signature of the resulting function and exactly how our C code interacted with the library, we were able to (with some hard work) create a module that is easy to use:

```

from cdiffusion import evolve

def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape, dtype=np.double)
    grid = np.zeros(grid_shape, dtype=np.double)

    # ... standard initialization ...

    for i in range(num_iterations):
        evolve(grid, next_grid, 1.0, 0.1)
        grid, next_grid = next_grid, grid

```

Wrap-Up

The various strategies introduced in this chapter allow you to specialize your code to different degrees in order to reduce the number of instructions the CPU must execute and to increase the efficiency of your programs. Sometimes this can be done algorithmically, although often it must be done manually (see “[JIT Versus AOT Compilers](#)” on page 164). Furthermore, sometimes these methods must be employed simply to use libraries that have already been written in other languages. Regardless of the motivation, Python allows us to benefit from the speedups that other languages can offer on some problems, while still maintaining verbosity and flexibility when needed.

We also looked into how to use the GPU to use purpose-specific hardware to solve problems faster than a CPU could alone. These devices are very specialized and can have very different performance considerations than classical high performance programming. However, we’ve seen that new libraries like PyTorch make evaluating the GPU much simpler than it ever has been.

It is important to note, though, that these optimizations are done to optimize the efficiency of compute instructions only. If you have I/O-bound processes coupled to a compute-bound problem, simply compiling your code may not provide any reasonable speedups. For these problems, we must rethink our solutions and potentially use parallelism to run different tasks at the same time.

Asynchronous I/O

Questions You'll Be Able to Answer After This Chapter

- What is concurrency, and how is it helpful?
- What is the difference between concurrency and parallelism?
- Which tasks can be done concurrently, and which can't?
- What are the various paradigms for concurrency?
- When is the right time to take advantage of concurrency?
- How can concurrency speed up my programs?

So far we have focused on speeding up code by increasing the number of compute cycles that a program can complete in a given time. However, in the days of big data, getting the relevant data to your code can be the bottleneck, as opposed to the actual code itself. When this is the case, your program is called *I/O bound*; in other words, the speed is bounded by the efficiency of the input/output.

I/O can be quite burdensome to the flow of a program. Every time your code reads from a file or writes to a network socket, it must pause to contact the kernel, request that the actual read happens, and then wait for it to complete. This is because it is not your program but the kernel that does the actual read operation, since the kernel is responsible for managing any interaction with hardware. The additional layer may not seem like the end of the world, especially once you realize that a similar operation happens every time memory is allocated; however, if we look back at [Figure 1-3](#), we see that most of the I/O operations we perform are on devices that are orders of magnitude slower than the CPU. So even if the communication with the kernel is fast,

we'll be waiting quite some time for the kernel to get the result from the device and return it to us.

For example, in the time it takes to write to a network socket, an operation that typically takes about 1 millisecond, we could have completed 2,400,000 instructions on a 2.4 GHz computer. Worst of all, our program is halted for much of this 1 millisecond of time—our execution is paused, and then we wait for a signal that the write operation has completed. This time spent in a paused state is called *I/O wait*.

Asynchronous I/O helps us utilize this wasted time by allowing us to perform other operations while we are in the I/O wait state. For example, in [Figure 8-1](#) we see a depiction of a program that must run three tasks, all of which have periods of I/O wait within them. If we run them serially, we suffer the I/O wait penalty three times. However, if we run these tasks concurrently, we can essentially hide the wait time by running another task in the meantime. It is important to note that this is all still happening on a single thread and still uses only one CPU at a time!

This is possible because while a program is in I/O wait, the kernel is simply waiting for whatever device we've requested to read from (hard drive, network adapter, GPU, etc.) to send a signal that the requested data is ready. Instead of waiting, we can create a mechanism (the event loop) so that we can dispatch requests for data, continue performing compute operations, and be notified when the data is ready to be read. This is in stark contrast to the multiprocessing/multithreading ([Chapter 9](#)) paradigm, where a new process is launched that does experience I/O wait but uses the multitasking nature of modern CPUs to allow the main process to continue. However, the two mechanisms are often used in tandem, where we launch multiple processes, each of which is efficient at asynchronous I/O, in order to fully take advantage of our computer's resources.



Since concurrent programs run on a single thread, they are generally easier to write and manage than standard multithreaded programs. All concurrent functions share the same memory space, so sharing data between them works in the normal ways you would expect. However, you still need to be careful about race conditions since you can't be sure which lines of code get run when.

By modeling a program in this event-driven way, we are able to take advantage of I/O wait to perform more operations on a single thread than would otherwise be possible.

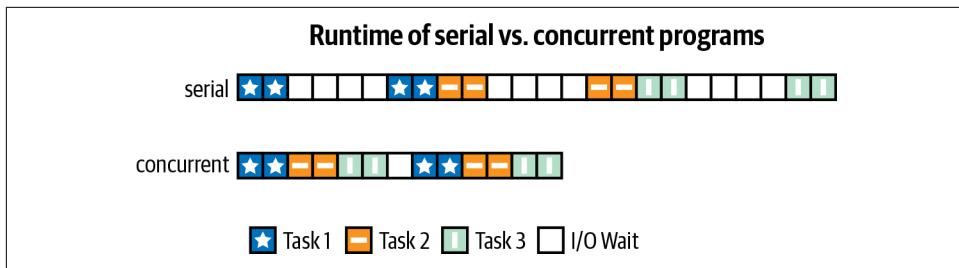


Figure 8-1. Comparison of serial and concurrent programs

Introduction to Asynchronous Programming

Typically, when a program enters I/O wait, the execution is paused so that the kernel can perform the low-level operations associated with the I/O request (this is called a *context switch*), and it is not resumed until the I/O operation is completed. Context switching is quite a heavy operation. It requires us to save the state of our program (losing any sort of caching we had at the CPU level) and give up the use of the CPU. Later, when we are allowed to run again, we must spend time reinitializing our program on the motherboard and getting ready to resume (of course, all this happens behind the scenes).

With concurrency, on the other hand, we typically have an *event loop* running that manages what gets to run in our program, and when. In essence, an event loop is simply a list of functions that need to be run. The function at the top of the list gets run, then the next, etc. [Example 8-1](#) shows a simple example of an event loop.

Example 8-1. Toy event loop

```
from queue import Queue
from functools import partial

eventloop = None

class EventLoop(Queue):
    def start(self):
        while True:
            function = self.get()
            function()

    def do_hello():
        global eventloop
        print("Hello")
        eventloop.put(do_world)

    def do_world():
        global eventloop
```

```

print("world")
eventloop.put(do_hello)

if __name__ == "__main__":
    eventloop = EventLoop()
    eventloop.put(do_hello)
    eventloop.start()

```

This may not seem like a big change; however, we can couple event loops with asynchronous (async) I/O operations for massive gains when performing I/O tasks. In this example, the call `eventloop.put(do_world)` approximates an asynchronous call to the `do_world` function. This operation is called `nonblocking`, meaning it will return immediately but guarantee that `do_world` is called at some point later. Similarly, if this were a network write with an `async` function, it will return right away even though the write has not happened yet. When the write has completed, an event fires so our program knows about it.

Putting these two concepts together, we can have a program that, when an I/O operation is requested, runs other functions while waiting for the original I/O operation to complete. This essentially allows us to still do meaningful calculations when we otherwise would have been in I/O wait.



Switching from function to function does have a cost. The kernel must take the time to set up the function to be called in memory, and the state of our caches won't be as predictable. It is because of this that concurrency gives the best results when your program has a lot of I/O wait—the cost associated with switching can be much less than what is gained by making use of I/O wait time.

Generally, programming using event loops can take two forms: callbacks or futures. In the callback paradigm, functions are called with an argument that is generally called the *callback*. Instead of the function returning its value, it calls the callback function with the value instead. This sets up long chains of functions that are called, with each function getting the result of the previous function in the chain (these chains are sometimes referred to as “callback hell”). [Example 8-2](#) is a simple example of the callback paradigm.

Example 8-2. Example with callbacks

```

from functools import partial
from some_database_library import save_results_to_db

def save_value(value, callback):
    print(f"Saving {value} to database")
    save_result_to_db(result, callback) ①

```

```

def print_response(db_response):
    print("Response from database: {db_response}")

if __name__ == "__main__":
    eventloop.put(partial(save_value, "Hello World", print_response))

```

- ❶ `save_result_to_db` is an asynchronous function; it will return immediately, and the function will end and allow other code to run. However, once the data is ready, `print_response` will be called.

Before Python 3.4, the callback paradigm was quite popular. However, the `asyncio` standard library module and PEP 492 made the future's mechanism native to Python. This was done by creating a standard API for dealing with asynchronous I/O and the new `await` and `async` keywords, which define an asynchronous function and a way to wait for a result.

In this paradigm, an asynchronous function returns a `Future` object, which is a promise of a future result. Because of this, if we want the result at some point we must wait for the future that is returned by this sort of asynchronous function to complete and be filled with the value we desire (either by doing an `await` on it or by running a function that explicitly waits for a value to be ready). However, this also means that the result can be available in the callers context, while in the callback paradigm the result is available only in the callback function. While waiting for the `Future` object to be filled with the data we requested, we can do other calculations. If we couple this with the concept of generators—functions that can be paused and whose execution can later be resumed—we can write asynchronous code that looks very close to serial code in form:

```

from some_async_database_library import save_results_to_db

async def save_value(value):
    print(f"Saving {value} to database")
    db_response = await save_result_to_db(result) ❶
    print("Response from database: {db_response}")

if __name__ == "__main__":
    eventloop.put(
        partial(save_value, "Hello World", print)
    )

```

- ❶ In this case, `save_result_to_db` returns a `Future` type. By awaiting it, we ensure that `save_value` gets paused until the value is ready and then resumes and completes its operations.

It's important to realize that the `Future` object returned by `save_result_to_db` holds the *promise* of a `Future` result and doesn't hold the result itself or even call any of the `save_result_to_db` code. In fact, if we simply did `db_response_future =`

`save_result_to_db(result)`, the statement would complete immediately and we could do other things with the `Future` object. For example, we could collect a list of futures and wait for all of them at the same time.

How Does `async/await` Work?

An `async` function (defined with `async def`) is called a *coroutine*. In Python, coroutines are implemented with the same philosophies as generators. This is convenient because generators already have the machinery to pause their execution and resume later. Using this paradigm, an `await` statement is similar in function to a `yield` statement; the execution of the current function gets paused while other code is run. Once the `await` or `yield` resolves with data, the function is resumed. So in the preceding example, our `save_result_to_db` will return a `Future` object, and the `await` statement pauses the function until that `Future` contains a result. The event loop is responsible for scheduling the resumption of `save_value` after the `Future` is ready to return a result.

For Python 2.7 implementations of future-based concurrency, things can get a bit strange when we're trying to use coroutines as actual functions. Remember that generators cannot return values, so libraries deal with this issue in various ways. The in Python 3.4, new machinery has been introduced in order to easily create coroutines and have them still return values. However, many asynchronous libraries that have been around since Python 2.7 have legacy code meant to deal with this awkward transition (in particular, `tornado`'s `gen` module).

It is critical to realize our reliance on an event loop when running concurrent code. In general, this leads to most fully concurrent code's main code entry point consisting mainly of setting up and starting the event loop. However, this assumes that your entire program is concurrent. In other cases, a set of futures is created within the program, and then a temporary event loop is started simply to manage the existing futures, before the event loop exits and the code can resume normally. This is generally done with either the `loop.run_until_complete(coro)` or `loop.run_forever()` method from the `asyncio.loop` module. However, `asyncio` also provides a convenience function (`asyncio.run(coro)`) to simplify this process.

In this chapter we will analyze a web crawler that fetches data from an HTTP server that has latency built into it. This represents the general response-time latency that will occur whenever dealing with I/O. We will first create a serial crawler that looks at the naive Python solution to this problem. Then we will build up to a full `aiohttp` solution by iterating through `gevent` and then `tornado`. Finally, we will look at combining `async` I/O tasks with CPU tasks in order to effectively hide any time spent doing I/O.



The web server we implemented can support multiple connections at a time. This will be true for most services that you will be performing I/O with—most databases can support multiple requests at a time, and most web servers support 10,000+ simultaneous connections. However, when interacting with a service that cannot handle multiple connections at a time, we will always have the same performance as the serial case.

Serial Crawler

For the control in our experiment with concurrency, we will write a serial web scraper that takes a list of URLs, fetches them, and sums the total length of the content from the pages. We will use a custom HTTP server that takes two parameters, `name` and `delay`. The `delay` field will tell the server how long, in milliseconds, to pause before responding. The `name` field is for logging purposes.

By controlling the `delay` parameter, we can simulate the time it takes a server to respond to our query. In the real world, this could correspond to a slow web server, a strenuous database call, or any I/O call that takes a long time to perform. For the serial case, this leads to more time that our program is stuck in I/O wait, but in the concurrent examples later on, it will provide an opportunity to spend the I/O wait time doing other tasks.

In [Example 8-3](#), we chose to use the `requests` module to perform the HTTP call. We made this choice because of the simplicity of the module. We use HTTP in general for this section because it is a simple example of I/O and can be performed easily. In general, any call to a HTTP library can be replaced with any I/O.

Example 8-3. Serial HTTP scraper

```
import random
import string

import requests

def generate_urls(base_url, num_urls):
    """
    We add random characters to the end of the URL to break any caching
    mechanisms in the requests library or the server
    """
    for i in range(num_urls):
        yield base_url + ''.join(random.sample(string.ascii_lowercase, 10))

def run_experiment(base_url, num_iter=1000):
    response_size = 0
    for url in generate_urls(base_url, num_iter):
        response = requests.get(url)
```

```
    response_size += len(response.text)
    return response_size

if __name__ == "__main__":
    import time

    delay = 100
    num_iter = 1000
    base_url = f"http://127.0.0.1:8080/add?name=serial&delay={delay}&"

    start = time.time()
    result = run_experiment(base_url, num_iter)
    end = time.time()
    print(f"Result: {result}, Time: {end - start}")
```

When running this code, an interesting metric to look at is the start and stop time of each request as seen by the HTTP server. This tells us how efficient our code was during I/O wait—since our task is to launch HTTP requests and then sum the number of characters that were returned, we should be able to launch more HTTP requests, and process any responses, while waiting for other requests to complete.

We can see in [Figure 8-2](#) that, as expected, there is no interleaving of our requests. We do one request at a time and wait for the previous request to happen before we move to the next request. In fact, the total runtime of the serial process makes perfect sense knowing this. Since each request takes 0.1 seconds (because of our `delay` parameter) and we are doing 500 requests, we expect this runtime to be about 50 seconds.

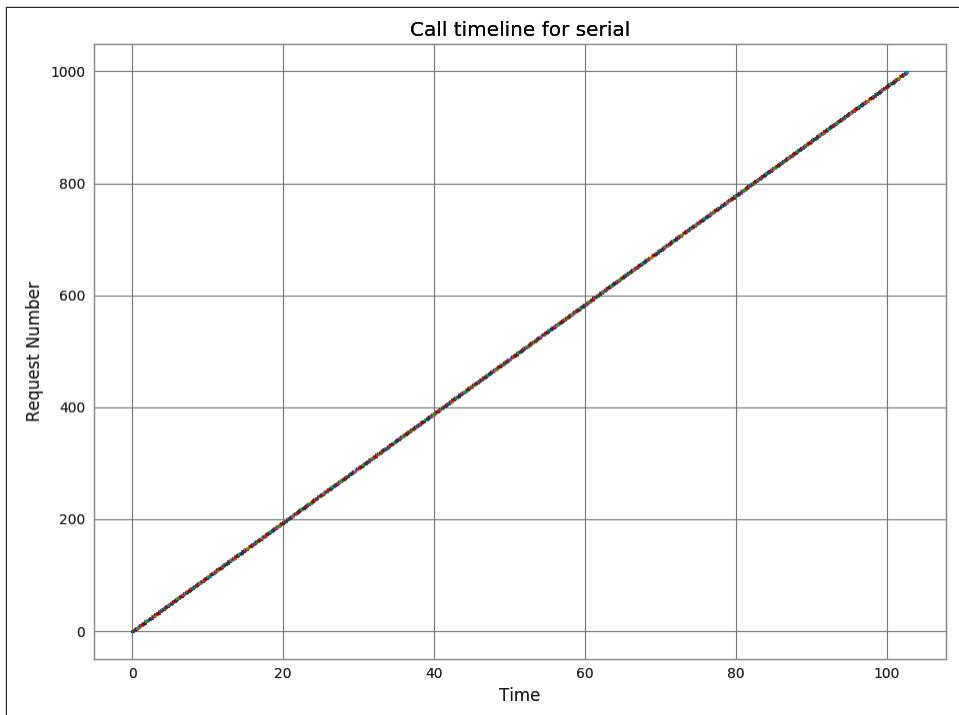


Figure 8-2. Request time for [Example 8-3](#)

Gevent

One of the simplest asynchronous libraries is `gevent`. It follows the paradigm of having asynchronous functions return futures, which means most of the logic in your code can stay the same. In addition, `gevent` monkey patches the standard I/O functions to be asynchronous, so most of the time you can simply use the standard I/O packages and benefit from asynchronous behavior.

`Gevent` provides two mechanisms to enable asynchronous programming—as mentioned before, it patches the standard library with asynchronous I/O functions, and it has a `Greenlets` object that can be used for concurrent execution. A *greenlet* is a type of coroutine and can be thought of as a thread (see [Chapter 9](#) for a discussion of threads); however, all greenlets run on the same physical thread. Instead of using multiple CPUs to run all the greenlets, we have an event loop on a single CPU that is able to switch between them during I/O wait. For the most part, `gevent` tries to make the handling of the event loop as transparent as possible through the use of `wait` functions. The `wait` function will start an event loop and run it as long as is needed for all greenlets to finish. Because of this, most of your `gevent` code will run serially; then at some point you will set up many greenlets to do a concurrent task and start

the event loop with the `wait` function. While the `wait` function is executing, all of the concurrent tasks you have queued up will run until completion (or some stopping condition), and then your code will go back to being serial again.

The futures are created with `gevent.spawn`, which takes a function and the arguments to that function and launches a greenlet that is responsible for running that function. The greenlet can be thought of as a future since, once the function we specified completes, its value will be contained within the greenlet's `value` field.

This patching of Python standard modules can make it harder to control the subtleties of what is going on. For example, one thing we want to ensure when doing async I/O is that we don't open too many files or connections at one time. If we do, we can overload the remote server or slow down our process by having to context-switch between too many operations.

To limit the number of open files manually, we use a semaphore to only do HTTP requests from one hundred greenlets at a time. A semaphore works by making sure that only a certain number of coroutines can enter the context block at a time. As a result, we launch all the greenlets that we need to fetch the URLs right away; however, only one hundred of them can make HTTP calls at a time. Semaphores are one type of locking mechanism used a lot in various parallel code flows. By restricting the progression of your code based on various rules, locks can help you make sure that the various components of your program don't interfere with one another.

Now that we have all the futures set up and have put in a locking mechanism to control the flow of the greenlets, we can wait until we start having results by using the `gevent.iwait` function, which will take a sequence of futures and iterate over the ready items. Conversely, we could have used `gevent.wait`, which would block execution of our program until all requests are done.

We go through the trouble of grouping our requests with the semaphore instead of sending them all at once because overloading the event loop can cause performance decreases (and this is true for all asynchronous programming). In addition, the server we are communicating with will have a limit to the number of concurrent requests it can respond to at the same time.

From experimentation (shown in [Figure 8-3](#)), we generally see that one hundred or so open connections at a time is optimal for requests with a reply time of about 50 milliseconds. If we were to use fewer connections, we would still have wasted time during I/O wait. With more, we are switching contexts too often in the event loop and adding unnecessary overhead to our program. We can see this effect come into play with four hundred concurrent requests for 50-millisecond requests. That being said, this value of one hundred depends on many things—the computer the code is being run on, the implementation of the event loop, the properties of the remote

host, the expected time to respond of the remote server, and so on. We recommend doing some experimentation before settling on a choice.

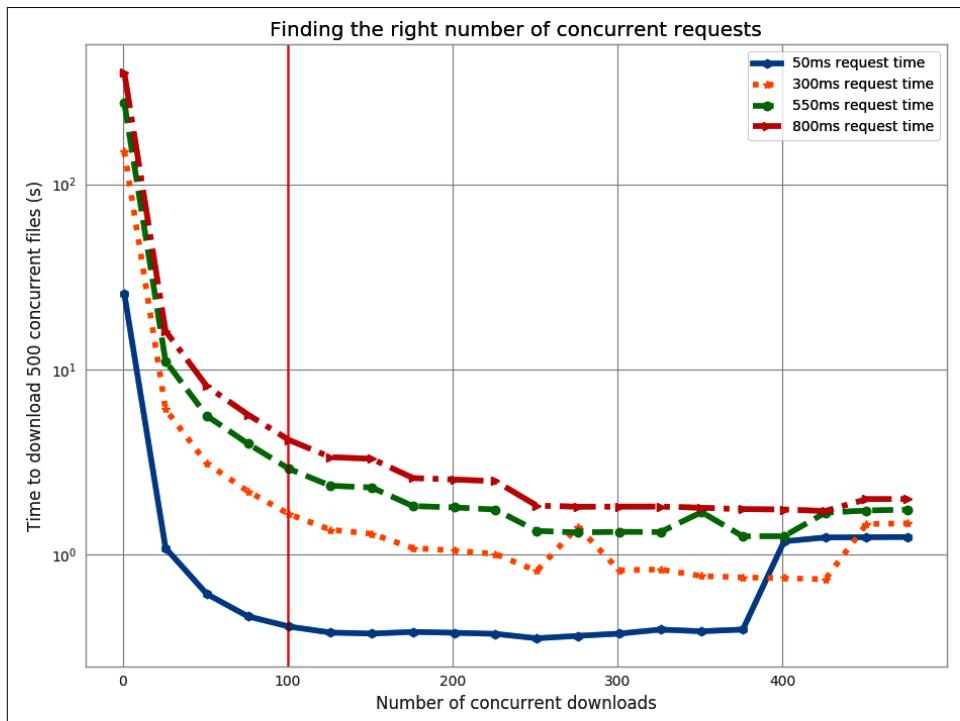


Figure 8-3. Experimenting with different numbers of concurrent requests for various request times

In Example 8-4, we implement the gevent scraper by using a semaphore to ensure only 100 requests at a time.

Example 8-4. gevent HTTP scraper

```
import random
import string
import urllib.error
import urllib.parse
import urllib.request
from contextlib import closing

import gevent
from gevent import monkey
from gevent.lock import Semaphore

monkey.patch_socket()
```

```

def generate_urls(base_url, num_urls):
    for i in range(num_urls):
        yield base_url + ''.join(random.sample(string.ascii_lowercase, 10))

def download(url, semaphore):
    with semaphore: ②
        with closing(urllib.request.urlopen(url)) as data:
            return data.read()

def chunked_requests(urls, chunk_size=100):
    """
    Given an iterable of urls, this function will yield back the contents of the
    URLs. The requests will be batched up in "chunk_size" batches using a
    semaphore
    """
    semaphore = Semaphore(chunk_size) ①
    requests = [gevent.spawn(download, u, semaphore) for u in urls] ③
    for response in gevent.iwait(requests):
        yield response

def run_experiment(base_url, num_iter=1000):
    urls = generate_urls(base_url, num_iter)
    response_futures = chunked_requests(urls, 100) ④
    response_size = sum(len(r.value) for r in response_futures)
    return response_size

if __name__ == "__main__":
    import time

    delay = 100
    num_iter = 1000
    base_url = f"http://127.0.0.1:8080/add?name=gevent&delay={delay}&"

    start = time.time()
    result = run_experiment(base_url, num_iter)
    end = time.time()
    print(f"Result: {result}, Time: {end - start}")

```

- ① Here we generate a semaphore that lets `chunk_size` downloads happen.
- ② By using the semaphore as a context manager, we ensure that only `chunk_size` greenlets can run the body of the context at one time.
- ③ We can queue up as many greenlets as we need, knowing that none of them will run until we start an event loop with `wait` or `iwait`.
- ④ `response_futures` now holds a generator over completed futures, all of which have our desired data in the `.value` property.

An important thing to note is that we have used gevent to make our I/O requests asynchronous, but we are not doing any non-I/O computations while in I/O wait. However, in [Figure 8-4](#) we can see the massive speedup we get (see [Table 8-1](#)). By launching more requests while waiting for previous requests to finish, we are able to achieve a 90x speedup! We can explicitly see that requests are being sent out before previous requests finish through the stacked horizontal lines representing the requests. This is in sharp contrast to the case of the serial crawler ([Figure 8-2](#)), where a line starts only when the previous line finishes.

Furthermore, we can see more interesting effects reflected in the shape of the gevent request timeline in [Figure 8-4](#). For example, at around the 100th request, we see a pause where new requests are not launched. This is because it is the first time that our semaphore is hit, and we are able to lock the semaphore before any previous requests finish. After this, the semaphore goes into an equilibrium: it locks just as another request finishes and unlocks it.

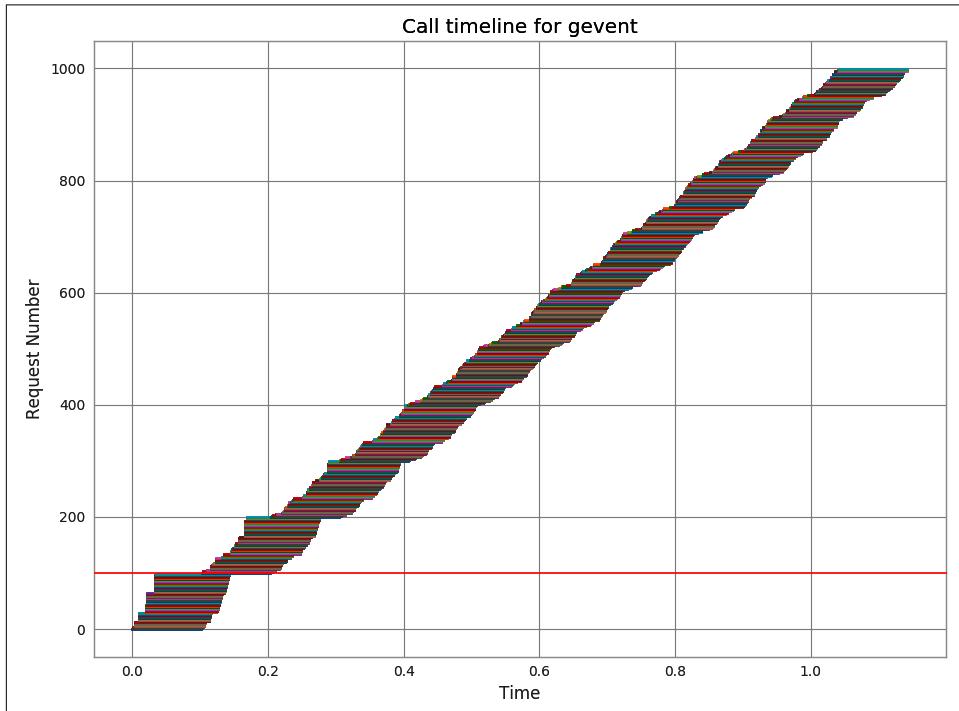


Figure 8-4. Request times for gevent scraper—the red line highlights the 100th request, where we can see a pause before subsequent requests are issued

tornado

Another frequently used package for asynchronous I/O in Python is `tornado`, originally developed by Facebook primarily for HTTP clients and servers. This framework has been around since Python 3.5, when `async/await` was introduced, and originally used a system of callbacks to organize asynchronous calls. Recently, however, the maintainers of the project have chosen to embrace the use of coroutines and in general have been critical in the architecture of the `asyncio` module.

Currently, `tornado` can be used either by using `async/await` syntax, as is standard in Python, or by using Python's `tornado.gen` module. This module was provided as a precursor to the native coroutines in Python. It did so by providing a decorator to turn a method into a coroutine (i.e., a way to get the same result as defining a function with `async def`) and various utilities to manage the runtime of coroutines. Currently this decorator approach is necessary only if you intend on providing support to Python versions older than 3.5.¹



When using `tornado`, make sure to have `pycurl` installed. It is an optional backend for `tornado` but performs better, especially with DNS requests, than the default backend.

In [Example 8-5](#), we implement the same web crawler as we did for `gevent`, but we use the `tornado` I/O loop (its version of an event loop) and HTTP client. This saves us the trouble of having to batch our requests and deal with other, more low-level aspects of our code.

Example 8-5. tornado HTTP scraper

```
import asyncio
import random
import string
from functools import partial

from tornado.httpclient import AsyncHTTPClient

AsyncHTTPClient.configure(
    "tornado.curl_httpclient.CurlAsyncHTTPClient",
    max_clients=100 ①
)

def generate_urls(base_url, num_urls):
```

¹ Which we're sure you're not doing!

```

for i in range(num_urls):
    yield base_url + ''.join(random.sample(string.ascii_lowercase, 10))

async def run_experiment(base_url, num_iter=1000):
    http_client = AsyncHTTPClient()
    urls = generate_urls(base_url, num_iter)
    response_sum = 0
    tasks = [http_client.fetch(url) for url in urls] ②
    for task in asyncio.as_completed(tasks): ③
        response = await task ④
        response_sum += len(response.body)
    return response_sum

if __name__ == "__main__":
    import time

    delay = 100
    num_iter = 1000
    run_func = partial(
        run_experiment,
        f"http://127.0.0.1:8080/add?name=tornado&delay={delay}&",
        num_iter,
    )

    start = time.time()
    result = asyncio.run(run_func) ⑤
    end = time.time()
    print(f"Result: {result}, Time: {end - start}")

```

- ➊ We can configure our HTTP client and pick what backend library we wish to use and how many requests we would like to batch together. Tornado defaults to a max of 10 concurrent requests.
- ➋ We generate many Future objects to queue the task of fetching the URL contents.
- ➌ This will run all of the coroutines that are queued in the tasks list and yield them as they complete.
- ➍ Since the coroutine is already completed, the await statement here returns immediately with the result of the earliest completed task.
- ➎ `ioloop.run_sync` will start the IOLoop just for the duration of the runtime of the specified function. `ioloop.start()`, on the other hand, starts an IOLoop that must be terminated manually.

An important difference between the tornado code in [Example 8-5](#) and the gevent code in [Example 8-4](#) is when the event loop runs. For gevent, the event loop is

running only while the `iwait` function is running. On the other hand, in `tornado` the event loop is running the entire time and controls the complete execution flow of the program, not just the asynchronous I/O parts.

This makes `tornado` ideal for any application that is mostly I/O-bound and where most, if not all, of the application should be asynchronous. This is where `tornado` makes its biggest claim to fame, as a performant web server. In fact, Micha has on many occasions written `tornado`-backed databases and data structures that require a lot of I/O.²

On the other hand, since `gevent` makes no requirements of your program as a whole, it is an ideal solution for mainly CPU-based problems that sometimes involve heavy I/O—for example, a program that does a lot of computations over a dataset and then must send the results back to the database for storage. This becomes even simpler with the fact that most databases have simple HTTP APIs, which means you can even use `requests`.

Another interesting difference between `gevent` and `tornado` is the way the internals change the request call graphs. Compare [Figure 8-5](#) with [Figure 8-4](#). For the `gevent` call graph, we see a very uniform call graph in which new requests are issued the second a slot in the semaphore opens up. On the other hand, the call graph for `tornado` is very stop-and-go. This means that the internal mechanism limiting the number of open connects is not reacting fast enough to request finishing. These areas of the call graph in which the line seems to be thinner/thicker than usual represent times when the event loop isn't doing its job optimally—times when we are either underutilizing or overutilizing our resources.



For all libraries that use `asyncio` to run the event loop, we can actually change the backend library that is being used. For example, the `uvloop` project supplies a drop-in replacement to `asyncio`'s event loop that claims massive speedups. These speedups are mainly seen server-side; in the client-side examples outlined in this chapter, they provide only a small performance boost. However, since it takes only two extra lines of code to use this event loop, there aren't many reasons not to use it!

We can start to understand this slowdown in light of the lesson we've been learning over and over again: that generalized code is useful because it solves all problems well but no individual problem perfectly. The mechanism to limit one hundred ongoing connections is fantastic when working with a large web app or a code base that may

² For example, `fuggetaboutit` is a special type of probabilistic data structure (see “[Probabilistic Data Structures](#)” on page 371) that uses the `tornado` `IOLoop` to schedule time-based tasks.

make HTTP requests in many different places. One simple configuration guarantees that overall we won't have more than the defined connections opened. However, in our situation we can benefit from being very specific as to how this is handled (as we did in the `gevent` example).

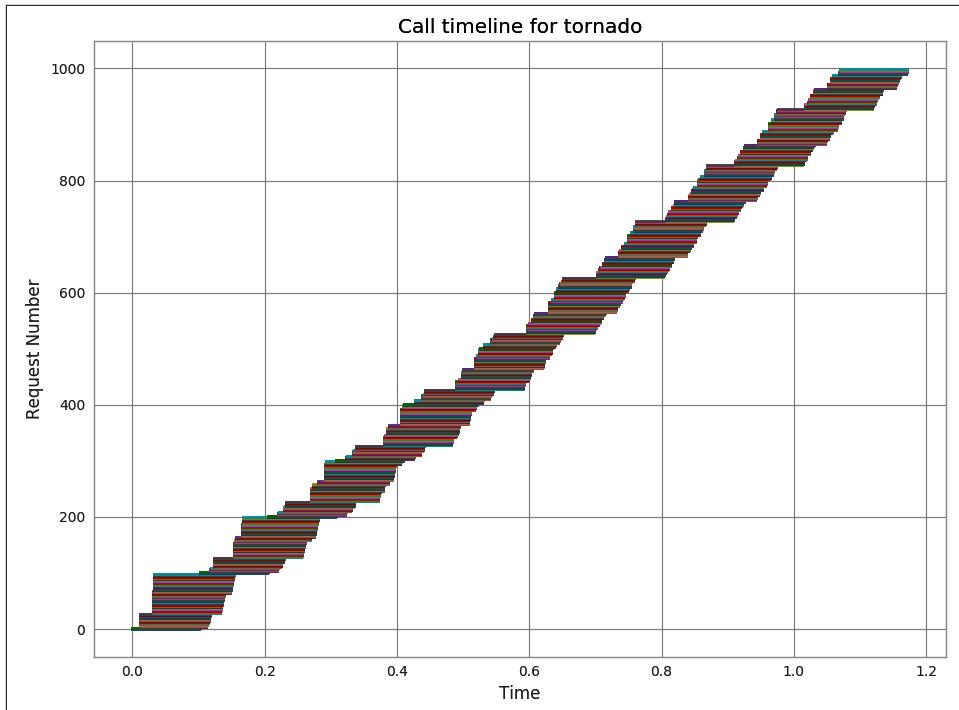


Figure 8-5. Chronology of HTTP requests for [Example 8-5](#)

aiohttp

In response to the popularity of using `async` functionality to deal with heavy I/O systems, Python 3.4+ introduced a revamping of the old `asyncio` standard library module. At the time, however, this module was quite low level, providing all of the low-level mechanisms for third-party libraries to create easy-to-use asynchronous libraries. `aiohttp` arose as the first popular library built entirely on the new `asyncio` library. It provides both HTTP client and server functionality and uses a similar API to those familiar with `tornado`. The entire project, [aio-libs](#), provides native asynchronous libraries for a wide variety of uses. In [Example 8-6](#), we show how to implement the `asyncio` scraper using `aiohttp`.

Example 8-6. asyncio HTTP scraper

```
import asyncio
import random
import string

import aiohttp

def generate_urls(base_url, num_urls):
    for i in range(num_urls):
        yield base_url + ''.join(random.sample(string.ascii_lowercase, 10))

def chunked_http_client(num_chunks):
    """
    Returns a function that can fetch from a URL, ensuring that only
    "num_chunks" of simultaneous connects are made.
    """
    semaphore = asyncio.Semaphore(num_chunks) ①

    async def http_get(url, client_session): ②
        nonlocal semaphore
        async with semaphore:
            async with client_session.request("GET", url) as response:
                return await response.content.read()

    return http_get

async def run_experiment(base_url, num_iter=1000):
    urls = generate_urls(base_url, num_iter)
    http_client = chunked_http_client(100)
    responses_sum = 0
    async with aiohttp.ClientSession() as client_session:
        tasks = [http_client(url, client_session) for url in urls] ③
        for future in asyncio.as_completed(tasks): ④
            data = await future
            responses_sum += len(data)
    return responses_sum

if __name__ == "__main__":
    import time

    loop = asyncio.get_event_loop()
    delay = 100
    num_iter = 1000

    start = time.time()
    result = loop.run_until_complete(
        run_experiment(
            f"http://127.0.0.1:8080/add?name=asyncio&delay={delay}&", num_iter
        )
    )
```

```
end = time.time()
print(f"Result: {result}, Time: {end - start}")
```

- ❶ As in the `gevent` example, we must use a semaphore to limit the number of requests.
- ❷ We return a new coroutine that will asynchronously download files and respect the locking of the semaphore.
- ❸ The `http_client` function returns futures. To keep track of progress, we save the futures into a list.
- ❹ As with `gevent`, we can wait for futures to become ready and iterate over them.

One immediate reaction to this code is the number of `async with`, `async def`, and `await` calls. In the definition for `http_get`, we use the `async` context manager to get access to shared resources in a concurrent-friendly way. That is to say, by using `async with`, we allow other coroutines to run while waiting to acquire the resources we are requesting. As a result, sharing things such as open semaphore slots or already opened connections to our host can be done more efficiently than we experienced with `tornado`.

In fact, the call graph in [Figure 8-6](#) shows a smooth transition similar to that of `gevent` in [Figure 8-4](#). Furthermore, the `asyncio` code runs slightly faster than the `gevent` code overall (1.10 seconds versus 1.14 seconds—see [Table 8-1](#)), even though the time for each request is slightly longer. This can be explained only by a faster resumption of coroutines paused by the semaphore or waiting for the HTTP client.

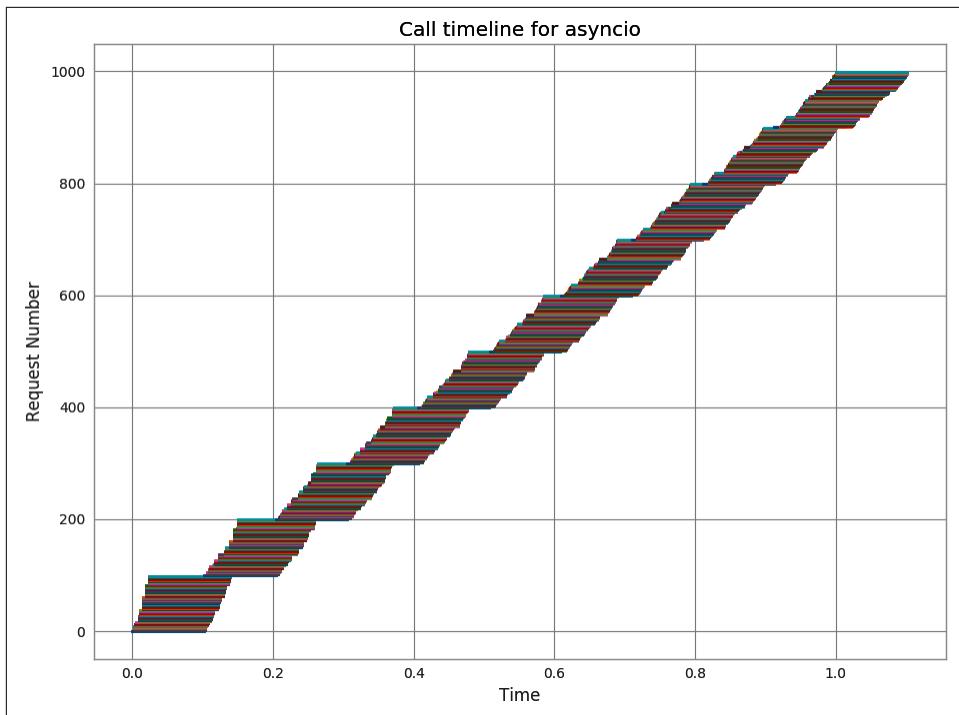


Figure 8-6. Chronology of HTTP requests for Example 8-6

The code sample also shows a big difference between using `aiohttp` and using `tornado` in that with `aiohttp`, we are very much in control of the event loop and the various subtleties of the request we are making. For example, we manually acquire the client session, which is responsible for caching open connections, and we manually read from the connection. If we wanted, we could change what time of connection caching is happening or decide to only write to the server and never read its response.

While this control may be a bit of overkill for such a simple example, in real-world applications we can use this to really tune the performance of our applications. Tasks can easily be added to the event loop without waiting for their response, and we can easily add time-outs to tasks so that their runtime is limited; we can even add functions that automatically get triggered when a task is completed. This allows us to create complicated runtime patterns that optimally utilize the time we gain by being able to run code during I/O wait. In particular, when we are running a web service (such as an API that may need to perform computational tasks for each request), this control can allow us to write “defensive” code that knows how to concede runtime to other tasks if a new request comes in. We will discuss this aspect more in “[Full Async](#)” on page 238.

Table 8-1. Comparison of total runtime for crawlers

	serial	gevent	tornado	aiohttp
Runtime (s)	102.684	1.142	1.171	1.101

Shared CPU–I/O Workload

To make the preceding examples more concrete, we will create another toy problem in which we have a CPU-bound problem that needs to communicate frequently with a database to save results. The CPU workload can be anything; in this case, we are taking the bcrypt hash of a random string with larger and larger workload factors to increase the amount of CPU-bound work (see [Table 8-2](#) to understand how the “difficulty” parameter affects runtime). This problem is representative of any sort of problem in which your program has heavy calculations to do, and the results of those calculations must be stored into a database, potentially incurring a heavy I/O penalty. The only restrictions we are putting on our database are as follows:

- It has an HTTP API so we can use code like that in the earlier examples.³
- Response times are on the order of 100 milliseconds.
- The database can satisfy many requests at a time.⁴

The response time of this “database” was chosen to be higher than usual in order to exaggerate the turning point in the problem, where the time to do one of the CPU tasks is longer than one of the I/O tasks. For a database that is being used only to store simple values, a response time greater than 10 milliseconds should be considered slow!

Table 8-2. Time to calculate a single hash

Difficulty parameter	8	10	11	12
Seconds per iteration	0.0156	0.0623	0.1244	0.2487

Serial

We start with some simple code that calculates the bcrypt hash of a string and makes a request to the database’s HTTP API every time a result is calculated:

```
import random
import string
```

³ This is not necessary; it just serves to simplify our code.

⁴ This is true for all distributed databases and other popular databases, such as Postgres, MongoDB, and so on.

```

import bcrypt
import requests

def do_task(difficulty):
    """
    Hash a random 10 character string using bcrypt with a specified difficulty
    rating.
    """
    passwd = ("".join(random.sample(string.ascii_lowercase, 10)) ①
              .encode("utf8"))
    salt = bcrypt.gensalt(difficulty) ②
    result = bcrypt.hashpw(passwd, salt)
    return result.decode("utf8")

def save_result_serial(result):
    url = f"http://127.0.0.1:8080/add"
    response = requests.post(url, data=result)
    return response.json()

def calculate_task_serial(num_iter, task_difficulty):
    for i in range(num_iter):
        result = do_task(task_difficulty)
        save_number_serial(result)

```

- ① We generate a random 10-character byte array.
- ② The `difficulty` parameter sets how hard it is to generate the password by increasing the CPU and memory requirements of the hashing algorithm.

Just as in our serial example ([Example 8-3](#)), the request times for each database save (100 milliseconds) do not stack, and we must pay this penalty for each result. As a result, iterating six hundred times with a task difficulty of 8 takes 71 seconds. We know, however, that because of the way our serial requests work, we are spending 40 seconds at minimum doing I/O! 56% of our program's runtime is being spent doing I/O and, moreover, just sitting around in "I/O wait," when it could be doing something else!

Of course, as the CPU problem takes more and more time, the relative slowdown of doing this serial I/O decreases. This is simply because the cost of having a 100-millisecond pause after each task pales in comparison to the long amount of time needed to do this computation (as we can see in [Figure 8-7](#)). This fact highlights how important it is to understand your workload before considering which optimizations to make. If you have a CPU task that takes hours and an I/O task that takes only seconds, work done to speed up the I/O task will not bring the huge speedups you may be looking for!

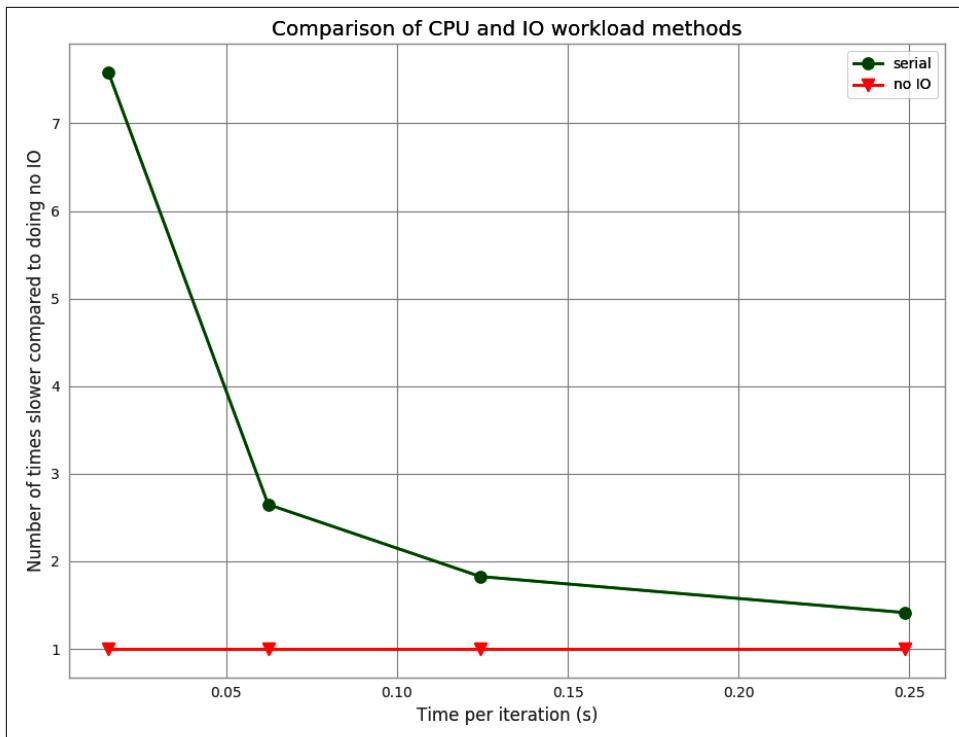


Figure 8-7. Comparison of the serial code to the CPU task with no I/O

Batched Results

Instead of immediately going to a full asynchronous solution, let's try an intermediate solution. If we don't need to know the results in our database right away, we can batch up the results and send them to the database asynchronously. To do this, we create an object, `AsyncBatcher`, that will take care of queuing results to be sent to the database in small asynchronous bursts. This will still pause the program and put it into I/O wait with no CPU tasks; however, during this time we can issue many concurrent requests instead of issuing them one at a time:

```
import asyncio
import aiohttp

class AsyncBatcher(object):
    def __init__(self, batch_size):
        self.batch_size = batch_size
        self.batch = []
        self.client_session = None
        self.url = f"http://127.0.0.1:8080/add"

    def __enter__(self):
```

```

        return self

    def __exit__(self, *args, **kwargs):
        self.flush()

    def save(self, result):
        self.batch.append(result)
        if len(self.batch) == self.batch_size:
            self.flush()

    def flush(self):
        """
        Synchronous flush function which starts an IOLoop for the purposes of
        running our async flushing function
        """
        loop = asyncio.get_event_loop()
        loop.run_until_complete(self.__aflush()) ①

    async def __aflush(self): ②
        async with aiohttp.ClientSession() as session:
            tasks = [self.fetch(result, session) for result in self.batch]
            for task in asyncio.as_completed(tasks):
                await task
        self.batch.clear()

    async def fetch(self, result, session):
        async with session.post(self.url, data=result) as response:
            return await response.json()

```

- ① We are able to start up an event loop just to run a single asynchronous function. The event loop will run until the asynchronous function is complete, and then the code will resume as normal.
- ② This function is nearly identical to that of [Example 8-6](#).

Now we can proceed almost in the same way as we did before. The main difference is that we add our results to our `AsyncBatcher` and let it take care of when to send the requests. Note that we chose to make this object into a context manager so that once we are done batching, the final `flush()` gets called. If we didn't do this, there would be a chance that we still have some results queued that didn't trigger a flush:

```

def calculate_task_batch(num_iter, task_difficulty):
    with AsyncBatcher(100) as batcher: ①
        for i in range(num_iter):
            result = do_task(i, task_difficulty)
            batcher.save(result)

```

- ① We choose to batch at 100 requests, for reasons similar to those illustrated in [Figure 8-3](#).

With this change, we are able to bring our runtime for a difficulty of 8 down to 10.21 seconds. This represents a $6.95\times$ speedup without our having to do much work. In a constrained environment such as a real-time data pipeline, this extra speed could mean the difference between a system being able to keep up with demand and that system falling behind (in which case a queue will be required; you'll learn about these in [Chapter 10](#)).

To understand what is happening in this timing, let's consider the variables that could affect the timings of this batched method. If our database had infinite throughput (i.e., if we could send an infinite number of requests at the same time without penalty), we could take advantage of the fact that we get only the 100-millisecond penalty when our `AsyncBatcher` is full and does a flush. In this case, we'd get the best performance by just saving all of our requests to the database and doing them all at once when the calculation was finished.

However, in the real world, our databases have a maximum throughput that limits the number of concurrent requests they can process. In this case, our server is limited at 100 requests a second, which means we must flush our batcher every one hundred results and take the 100-millisecond penalty then. This is because the batcher still pauses the execution of the program, just as the serial code did; however, in that paused time it performs many requests instead of just one.

If we tried to save all our results to the end and then issued them all at once, the server would only *process* one hundred at a time, and we'd have an extra penalty in terms of the overhead to making all those requests at the same time in addition to overloading our database, which can cause all sorts of unpredictable slowdowns.

On the other hand, if our server had terrible throughput and could deal with only one request at a time, we may as well run our code in serial! Even if we kept our batching at one hundred results per batch, when we actually go to make the requests, only one would get responded to at a time, effectively invalidating any batching we made.

This mechanism of batching results, also known as *pipelining*, can help tremendously when trying to lower the burden of an I/O task (as seen in [Figure 8-8](#)). It offers a good compromise between the speeds of asynchronous I/O and the ease of writing serial programs. However, a determination of how much to pipeline at a time is very case-dependent and requires some profiling and tuning to get the best performance.

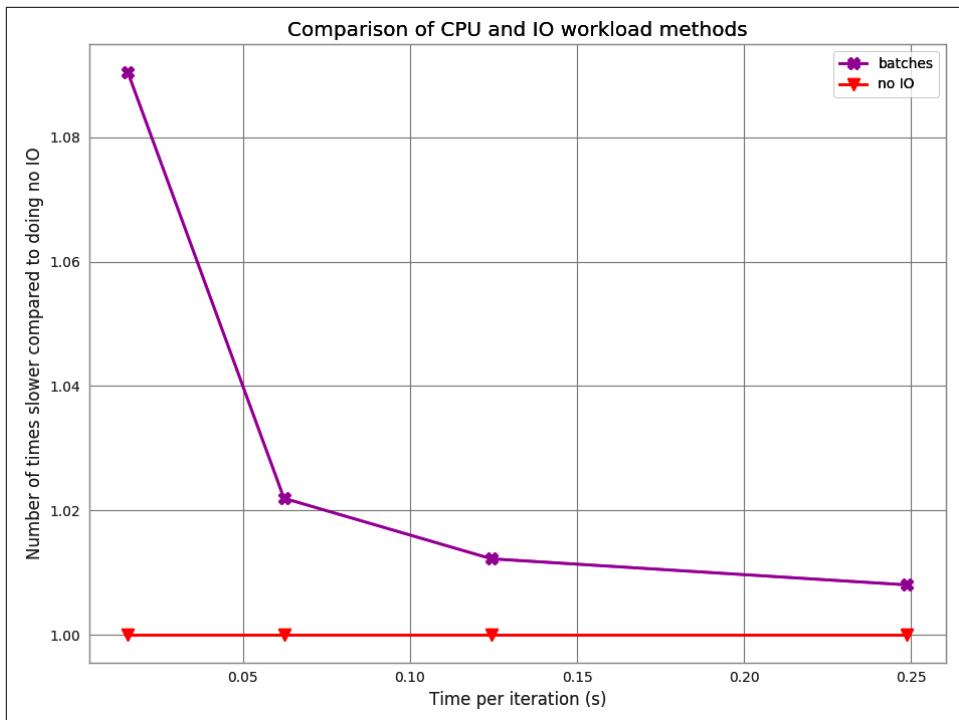


Figure 8-8. Comparison of batching requests versus not doing any I/O

Full Async

In some cases, we may need to implement a full asynchronous solution. This may happen if the CPU task is part of a larger I/O-bound program, such as an HTTP server. Imagine that you have an API service that, in response to some of its endpoints, has to perform heavy computational tasks. We still want the API to be able to handle concurrent requests and be performant in its tasks, but we also want the CPU task to run quickly.

The implementation of this solution in [Example 8-7](#) uses code very similar to that of [Example 8-6](#).

Example 8-7. Async CPU workload

```
def save_result_aiohttp(client_session):
    sem = asyncio.Semaphore(100)

    async def saver(result):
        nonlocal sem, client_session
        url = f"http://127.0.0.1:8080/add"
        async with sem:
```

```

        async with client_session.post(url, data=result) as response:
            return await response.json()

    return saver

async def calculate_task_aiohttp(num_iter, task_difficulty):
    tasks = []
    async with aiohttp.ClientSession() as client_session:
        saver = save_result_aiohttp(client_session)
        for i in range(num_iter):
            result = do_task(i, task_difficulty)
            task = asyncio.create_task(saver(result)) ❶
            tasks.append(task)
            await asyncio.sleep(0) ❷
    await asyncio.wait(tasks) ❸

```

- ❶ Instead of awaiting our database save immediately, we queue it into the event loop using `asyncio.create_task` and keep track of it so we can ensure that the task has completed before the end of the function.
- ❷ This is arguably the most important line in the function. Here, we pause the main function to allow the event loop to take care of any pending tasks. Without this, none of our queued tasks would run until the end of the function.
- ❸ Here we wait for any tasks that haven't completed yet. If we had not done the `asyncio.sleep` in the `for` loop, all the saves would happen here!

Before we go into the performance characteristics of this code, we should first talk about the importance of the `asyncio.sleep(0)` statement. It may seem strange to be sleeping for zero seconds, but this statement is a way to force the function to defer execution to the event loop and allow other tasks to run. In general in asynchronous code, this deferring happens every time an `await` statement is run. Since we generally don't `await` in CPU-bound code, it's important to force this deferment, or else no other task will run until the CPU-bound code is complete. In this case, if we didn't have the sleep statement, all the HTTP requests would be paused until the `asyncio.wait` statement, and then all the requests would be issued at once, which is definitely not what we want!

One nice thing about having this control is that we can choose the best times to defer back to the event loop. There are many considerations when doing this. Since the run state of the program changes when we defer, we don't want to do it in the middle of a calculation and potentially change our CPU cache. In addition, deferring to the event loop has an overhead cost, so we don't want to do it too frequently. However, while we are bound up doing the CPU task, we cannot do any I/O tasks. So if our full application is an API, no requests can be handled during the CPU time!

Our general rule of thumb is to try to issue an `asyncio.sleep(0)` at any loop that we expect to iterate every 50 to 100 milliseconds or so. Some applications use `time.perf_counter` and allow the CPU task to have a specific amount of runtime before forcing a sleep. For a situation such as this, though, since we have control of the number of CPU and I/O tasks, we just need to make sure that the time between sleeps coincides with the time needed for pending I/O tasks to complete.

One major performance benefit to the full asynchronous solution is that we can perform all of our I/O *while* we are doing our CPU work, effectively hiding it from our total runtime (as we can see from the overlapping lines in Figure 8-9). While it will never be completely hidden because of the overhead costs of the event loop, we can get very close. In fact, for a difficulty of 8 with 600 iterations, our code runs 7.3× faster than the serial code and performs its total I/O workload 2× faster than the batched code (and this benefit over the batched code would only get better as we do more iterations, since the batched code loses time versus the asynchronous code every time it has to pause the CPU task to flush a batch).

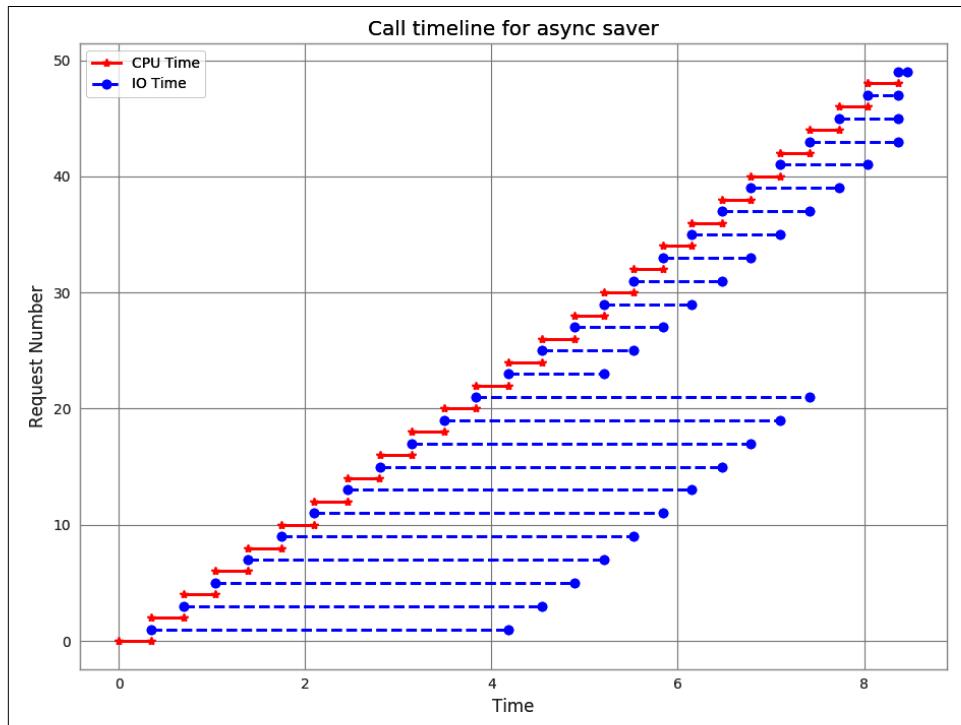


Figure 8-9. Call graph for 25 difficulty-8 CPU tasks using the `aiohttp` solution—the red lines represent time working on a CPU task, while blue lines represent time sending a result to the server

In the call timeline, we can really see what is going on. What we've done is to mark the beginning and end of each CPU and I/O task for a short run of 25 CPU tasks with difficulty 8. The first several I/O tasks are the slowest, taking a while to make the initial connection to our server. Because of our use of `aiohttp`'s `ClientSession`, these connections are cached, and all subsequent connections to the same server are much faster.

After this, if we just focus on the blue lines, they seem to happen very regularly without much of a pause between CPU tasks. Indeed, we don't see the 100-millisecond delay from the HTTP request between tasks. Instead, we see the HTTP request being issued quickly at the end of each CPU task and later being marked as completed at the end of another CPU task.

We do see, though, that each individual I/O task takes longer than the 100-millisecond response time from the server. This longer wait time is given by the frequency of our `asyncio.sleep(0)` statements (since each CPU task has one `await`, while each I/O task has three) and the way the event loop decides which tasks come next. For the I/O task, this extra wait time is OK because it doesn't interrupt the CPU task at hand. In fact, at the end of the run we can see the I/O runtimes shorten until the final I/O task is run. This final blue line is triggered by the `asyncio.wait` statement and runs incredibly quickly since it is the only remaining task and never needs to switch to other tasks.

In Figures 8-10 and 8-11, we can see a summary of how these changes affect the runtime of our code for different workloads. The speedup in the `async` code over the serial code is significant, although we are still a ways away from the speeds achieved in the raw CPU problem. For this to be completely remedied, we would need to use modules like `multiprocessing` to have a completely separate process that can deal with the I/O burden of our program without slowing down the CPU portion of the problem.

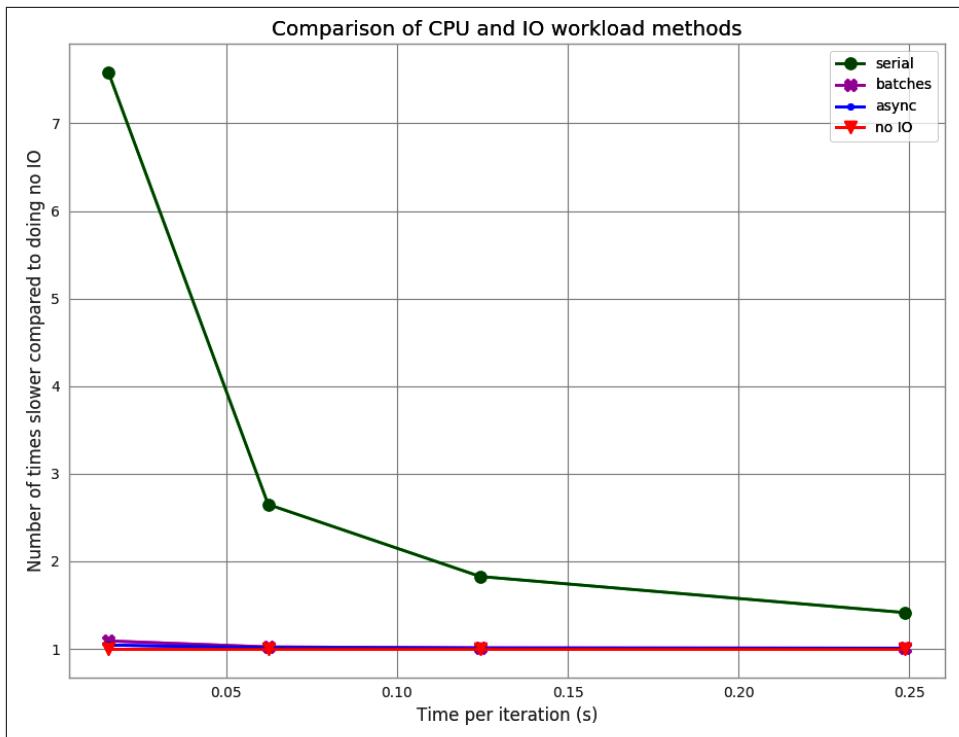


Figure 8-10. Processing time difference between serial I/O, batched async I/O, full async I/O, and a control case where I/O is completely disabled

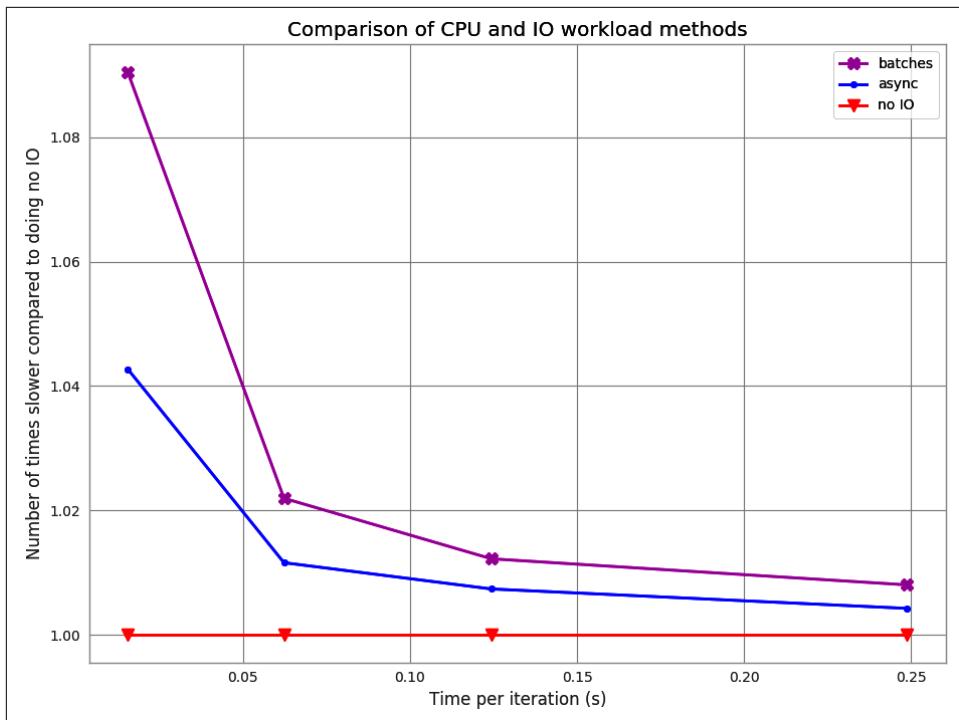


Figure 8-11. Processing time difference between batched async, full async I/O, and I/O disabled

Wrap-Up

When solving problems in real-world and production systems, it is often necessary to communicate with an outside source. This outside source could be a database running on another server, another worker computer, or a data service that is providing the raw data that must be processed. Whenever this is the case, your problem can quickly become I/O-bound, meaning that most of the runtime is dominated by dealing with input/output.

Concurrency helps with I/O-bound problems by allowing you to interleave computation with potentially multiple I/O operations. This allows you to exploit the fundamental difference between I/O and CPU operations in order to speed up overall runtime.

As we saw, `gevent` provides the highest-level interface for asynchronous I/O. On the other hand, `tornado` and `aiohttp` allow full control of an asynchronous I/O stack. In addition to the various levels of abstraction, every library uses a different paradigm for its syntax. However, `asyncio` is the binding glue for the asynchronous solutions and provides the fundamental mechanisms to control them all.

We also saw how to mesh CPU and I/O tasks together and how to consider the various performance characteristics of each to come up with a good solution to the problem. While it may be appealing to go to a full asynchronous code immediately, sometimes intermediate solutions work almost as well without having quite the engineering burden.

In the next chapter, we will take this concept of interleaving computation from I/O-bound problems and apply it to CPU-bound problems. With this new ability, we will be able to perform not only multiple I/O operations at once but also many computational operations. This capability will allow us to start to make fully scalable programs where we can achieve more speed by simply adding more computer resources that can each handle a chunk of the problem.

The multiprocessing Module

Questions You'll Be Able to Answer After This Chapter

- What does the `multiprocessing` module offer?
- What's the difference between processes and threads?
- How do I choose the right size for a process pool?
- How do I use nonpersistent queues for work processing?
- What are the costs and benefits of interprocess communication?
- How can I process `numpy` data with many CPUs?
- How would I use Joblib to simplify parallelized and cached scientific work?
- Why do I need locking to avoid data loss?

C`Python` doesn't use multiple CPUs by default. This is partly because Python was designed back in a single-core era, and partly because parallelizing can actually be quite difficult to do efficiently. Python gives us the tools to do it but leaves us to make our own choices. It is painful to see your multicore machine using just one CPU on a long-running process, though, so in this chapter we'll review ways of using all the machine's cores at once.



We just mentioned *C`Python`*—the common implementation that we all use. Nothing in the Python language stops it from using multicore systems. C`Python`'s implementation cannot efficiently use multiple cores, but future implementations may not be bound by this restriction.

We live in a multicore world—4 cores are common in laptops, and 32-core desktop configurations are available. If your job can be split to run on multiple CPUs *without* too much engineering effort, this is a wise direction to consider.

When Python is used to parallelize a problem over a set of CPUs, you can expect *up to* an n -times ($n\times$) speedup with n cores. If you have a quad-core machine and you can use all four cores for your task, it might run in a quarter of the original runtime. You are unlikely to see a greater than $4\times$ speedup; in practice, you'll probably see gains of $3\text{--}4\times$.

Each additional process will increase the communication overhead and decrease the available RAM, so you rarely get a full n -times speedup. Depending on which problem you are solving, the communication overhead can even get so large that you can see very significant slowdowns. These sorts of problems are often where the complexity lies for any sort of parallel programming and normally require a change in algorithm. This is why parallel programming is often considered an art.

If you're not familiar with [Amdahl's law](#), it is worth doing some background reading. The law shows that if only a small part of your code can be parallelized, it doesn't matter how many CPUs you throw at it; it still won't run much faster overall. Even if a large fraction of your runtime could be parallelized, there's a finite number of CPUs that can be used efficiently to make the overall process run faster before you get to a point of diminishing returns.

The `multiprocessing` module lets you use process- and thread-based parallel processing, share work over queues, and share data among processes. It is mostly focused on single-machine multicore parallelism (there are better options for multimachine parallelism). A very common use is to parallelize a task over a set of processes for a CPU-bound problem. You might also use OpenMP to parallelize an I/O-bound problem, but as we saw in [Chapter 8](#), there are better tools for this (e.g., the new `asyncio` module in Python 3 and `tornado`).



OpenMP is a low-level interface to multiple cores—you might wonder whether to focus on it rather than `multiprocessing`. We introduced it with Cython back in [Chapter 7](#), but we don't cover it in this chapter. `multiprocessing` works at a higher level, sharing Python data structures, while OpenMP works with C primitive objects (e.g., integers and floats) once you've compiled to C. Using it makes sense only if you're compiling your code; if you're not compiling (e.g., if you're using efficient `numpy` code and you want to run on many cores), then sticking with `multiprocessing` is probably the right approach.

To parallelize your task, you have to think a little differently from the normal way of writing a serial process. You must also accept that debugging a parallelized task is

harder—often, it can be very frustrating. We’d recommend keeping the parallelism as simple as possible (even if you’re not squeezing every last drop of power from your machine) so that your development velocity is kept high.

One particularly difficult topic is the sharing of state in a parallel system—it feels like it should be easy, but it incurs lots of overhead and can be hard to get right. There are many use cases, each with different trade-offs, so there’s definitely no one solution for everyone. In “[Verifying Primes Using Interprocess Communication](#)” on page 278, we’ll go through state sharing with an eye on the synchronization costs. Avoiding shared state will make your life far easier.

In fact, an algorithm can be analyzed to see how well it’ll perform in a parallel environment almost entirely by how much state must be shared. For example, if we can have multiple Python processes all solving the same problem without communicating with one another (a situation known as *embarrassingly parallel*), not much of a penalty will be incurred as we add more and more Python processes.

On the other hand, if each process needs to communicate with every other Python process, the communication overhead will slowly overwhelm the processing and slow things down. This means that as we add more and more Python processes, we can actually slow down our overall performance.

As a result, sometimes some counterintuitive algorithmic changes must be made to efficiently solve a problem in parallel. For example, when solving the diffusion equation ([Chapter 6](#)) in parallel, each process actually does some redundant work that another process also does. This redundancy reduces the amount of communication required and speeds up the overall calculation!

Here are some typical jobs for the `multiprocessing` module:

- Parallelize a CPU-bound task with `Process` or `Pool` objects
- Parallelize an I/O-bound task in a `Pool` with threads using the (oddly named) `dummy` module
- Share pickled work via a `Queue`
- Share state between parallelized workers, including bytes, primitive datatypes, dictionaries, and lists

If you come from a language where threads are used for CPU-bound tasks (e.g., C++ or Java), you should know that while threads in Python are OS-native (they’re not simulated—they are actual operating system threads), they are bound by the GIL, so only one thread may interact with Python objects at a time.

By using processes, we run a number of Python interpreters in parallel, each with a private memory space with its own GIL, and each runs in series (so there’s no competition for each GIL). This is the easiest way to speed up a CPU-bound task in Python.

If we need to share state, we need to add some communication overhead; we'll explore that in "[Verifying Primes Using Interprocess Communication](#)" on page 278.

If you work with `numpy` arrays, you might wonder if you can create a larger array (e.g., a large 2D matrix) and ask processes to work on segments of the array in parallel. You can, but it is hard to discover how by trial and error, so in "[Sharing numpy Data with multiprocessing](#)" on page 295 we'll work through sharing a 25 GB `numpy` array across four CPUs. Rather than sending partial copies of the data (which would at least double the working size required in RAM and create a massive communication overhead), we share the underlying bytes of the array among the processes. This is an ideal approach to sharing a large array among local workers on one machine.

In this chapter we also introduce the `Joblib` library—this builds on the `multiprocessing` library and offers improved cross-platform compatibility, a simple API for parallelization, and convenient persistence of cached results. `Joblib` is designed for scientific use, and we urge you to check it out.



Here, we discuss `multiprocessing` on *nix-based machines (this chapter is written using Ubuntu; the code should run unchanged on a Mac). Since Python 3.4, the quirks that appeared on Windows have been dealt with. `Joblib` has stronger cross-platform support than `multiprocessing`, and we recommend you review it ahead of `multiprocessing`.

In this chapter we'll hardcode the number of processes (`NUM_PROCESSES=4`) to match the four physical cores on Ian's laptop. By default, `multiprocessing` will use as many cores as it can see (the machine presents eight—four CPUs and four hyperthreads). Normally you'd avoid hardcoding the number of processes to create unless you were specifically managing your resources.

An Overview of the `multiprocessing` Module

The `multiprocessing` module provides a low-level interface to process- and thread-based parallelism. Its main components are as follows:

Process

A forked copy of the current process; this creates a new process identifier, and the task runs as an independent child process in the operating system. You can start and query the state of the `Process` and provide it with a `target` method to run.

Pool

Wraps the `Process` or `threading.Thread` API into a convenient pool of workers that share a chunk of work and return an aggregated result.

Queue

A FIFO queue allowing multiple producers and consumers.

Pipe

A uni- or bidirectional communication channel between two processes.

Manager

A high-level managed interface to share Python objects between processes.

ctypes

Allows sharing of primitive datatypes (e.g., integers, floats, and bytes) between processes after they have forked.

Synchronization primitives

Locks and semaphores to synchronize control flow between processes.



In Python 3.2, the `concurrent.futures` module was introduced (via [PEP 3148](#)); this provides the core behavior of `multiprocessing`, with a simpler interface based on Java's `java.util.concurrent`. It is available as a [backport to earlier versions of Python](#). We expect `multiprocessing` to continue to be preferred for CPU-intensive work and won't be surprised if `concurrent.futures` becomes more popular for I/O-bound tasks.

In the rest of the chapter, we'll introduce a set of examples to demonstrate common ways of using the `multiprocessing` module.

We'll estimate pi using a Monte Carlo approach with a `Pool` of processes or threads, using normal Python and `numpy`. This is a simple problem with well-understood complexity, so it parallelizes easily; we can also see an unexpected result from using threads with `numpy`. Next, we'll search for primes using the same `Pool` approach; we'll investigate the nonpredictable complexity of searching for primes and look at how we can efficiently (and inefficiently!) split the workload to best use our computing resources. We'll finish the primes search by switching to queues, where we introduce `Process` objects in place of a `Pool` and use a list of work and poison pills to control the lifetime of workers.

Next, we'll tackle interprocess communication (IPC) to validate a small set of possible primes. By splitting each number's workload across multiple CPUs, we use IPC to end the search early if a factor is found so that we can significantly beat the speed of a single-CPU search process. We'll cover shared Python objects, OS primitives, and a Redis server to investigate the complexity and capability trade-offs of each approach.

We can share a 25 GB `numpy` array across four CPUs to split a large workload *without* copying data. If you have large arrays with parallelizable operations, this technique

should buy you a great speedup, since you have to allocate less space in RAM and copy less data. Finally, we'll look at synchronizing access to a file and a variable (as a `Value`) between processes without corrupting data to illustrate how to correctly lock shared state.



PyPy (discussed in [Chapter 7](#)) has full support for the `multiprocessing` library, and the following CPython examples (though not the `numpy` examples, at the time of this writing) all run far quicker using PyPy. If you're using only CPython code (no C extensions or more complex libraries) for parallel processing, PyPy might be a quick win for you.

Estimating Pi Using the Monte Carlo Method

We can estimate pi by throwing thousands of imaginary darts into a “dartboard” represented by a unit circle. The relationship between the number of darts falling inside the circle’s edge and the number falling outside it will allow us to approximate pi.

This is an ideal first problem, as we can split the total workload evenly across a number of processes, each one running on a separate CPU. Each process will end at the same time since the workload for each is equal, so we can investigate the speedups available as we add new CPUs and hyperthreads to the problem.

In [Figure 9-1](#), we throw 10,000 darts into the unit square, and a percentage of them fall into the quarter of the unit circle that's drawn. This estimate is rather bad—10,000 dart throws does not reliably give us a three-decimal-place result. If you ran your own code, you'd see this estimate vary between 3.0 and 3.2 on each run.

To be confident of the first three decimal places, we need to generate 10,000,000 random dart throws.¹ This is massively inefficient (and better methods for pi's estimation exist), but it is rather convenient to demonstrate the benefits of parallelization using `multiprocessing`.

With the Monte Carlo method, we use the [Pythagorean theorem](#) to test if a dart has landed inside our circle:

$$x^2 + y^2 \leq 1^2 = 1$$

¹ See [Brett Foster's PowerPoint presentation](#) on using the Monte Carlo method to estimate pi.

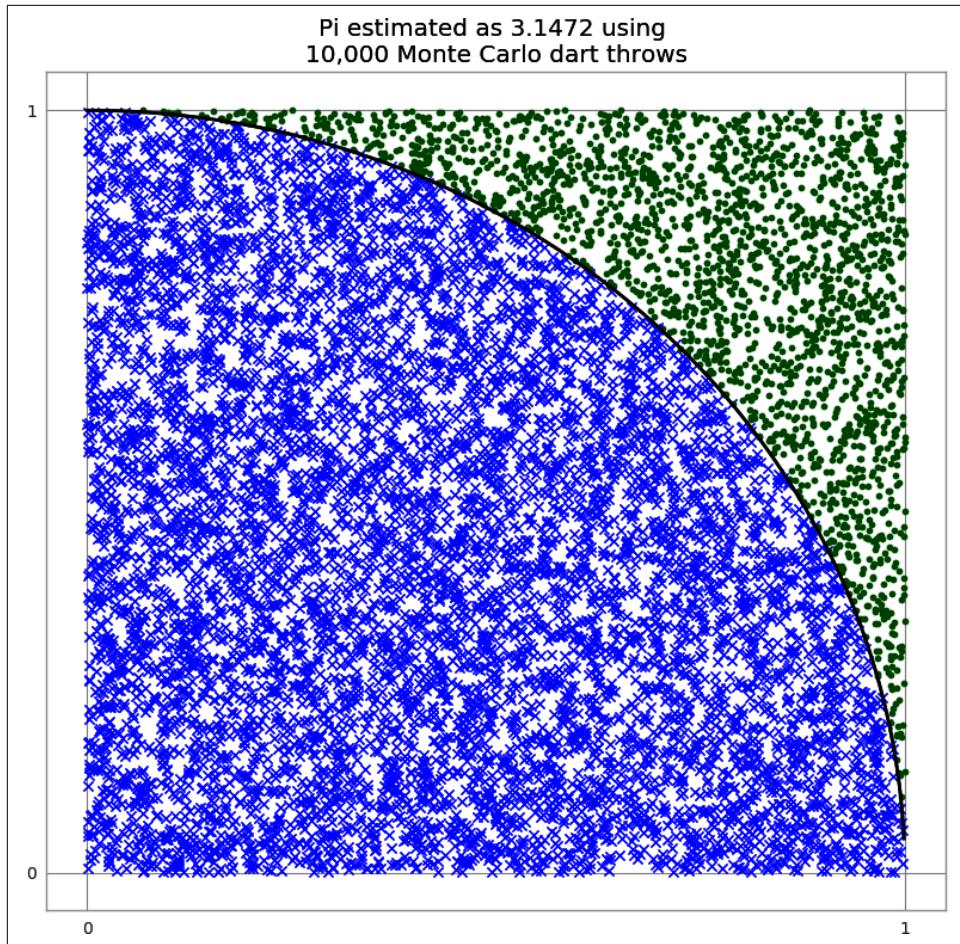


Figure 9-1. Estimating pi using the Monte Carlo method

We'll look at a loop version of this in [Example 9-1](#). We'll implement both a normal Python version and, later, a `numpy` version, and we'll use both threads and processes to parallelize the problem.

Estimating Pi Using Processes and Threads

It is easier to understand a normal Python implementation, so we'll start with that in this section, using float objects in a loop. We'll parallelize this using processes to use all of our available CPUs, and we'll visualize the state of the machine as we use more CPUs.

Using Python Objects

The Python implementation is easy to follow, but it carries an overhead, as each Python float object has to be managed, referenced, and synchronized in turn. This overhead slows down our runtime, but it has bought us thinking time, as the implementation was quick to put together. By parallelizing this version, we get additional speedups for very little extra work.

Figure 9-2 shows three implementations of the Python example:

- No use of `multiprocessing` (named “Serial”—one for loop in the main process
- Using threads
- Using processes

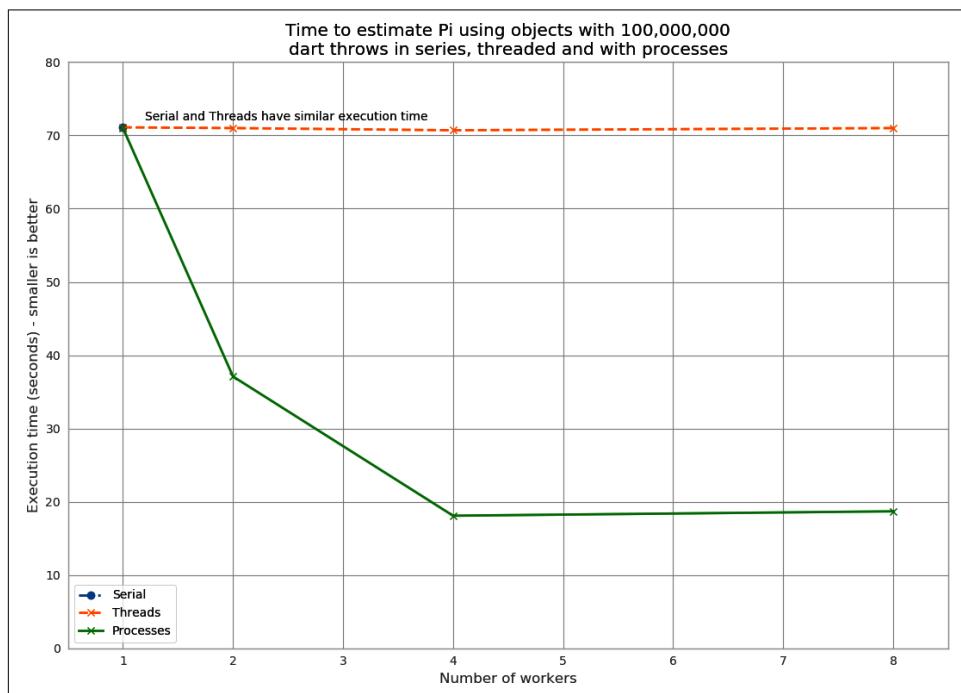


Figure 9-2. Working in series, with threads, and with processes

When we use more than one thread or process, we’re asking Python to calculate the same total number of dart throws and to divide the work evenly between workers. If we want 100,000,000 dart throws in total using our Python implementation and we use two workers, we’ll be asking both threads or both processes to generate 50,000,000 dart throws per worker.

Using one thread takes approximately 71 seconds, with no speedup when using more threads. By using two or more processes, we make the runtime *shorter*. The cost of using no processes or threads (the series implementation) is the same as running with one process.

By using processes, we get a linear speedup when using two or four cores on Ian’s laptop. For the eight-worker case, we’re using Intel’s Hyper-Threading Technology—the laptop has only four physical cores, so we get barely any change in speedup by running eight processes.

Example 9-1 shows the Python version of our pi estimator. If we’re using threads, each instruction is bound by the GIL, so although each thread could run on a separate CPU, it will execute only when no other threads are running. The process version is not bound by this restriction, as each forked process has a private Python interpreter running as a single thread—there’s no GIL contention, as no objects are shared. We use Python’s built-in random number generator, but see “[Random Numbers in Parallel Systems](#)” on page 263 for some notes about the dangers of parallelized random number sequences.

Example 9-1. Estimating pi using a loop in Python

```
def estimate_nbr_points_in_quarter_circle(nbr_estimates):
    """Monte Carlo estimate of the number of points in a
    quarter circle using pure Python"""
    print(f"Executing estimate_nbr_points_in_quarter_circle  \
          with {nbr_estimates:,} on pid {os.getpid()}")
    nbr_trials_in_quarter_unit_circle = 0
    for step in range(int(nbr_estimates)):
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        is_in_unit_circle = x * x + y * y <= 1.0
        nbr_trials_in_quarter_unit_circle += is_in_unit_circle

    return nbr_trials_in_quarter_unit_circle
```

Example 9-2 shows the `__main__` block. Note that we build the `Pool` before we start the timer. Spawning threads is relatively instant; spawning processes involves a fork, and this takes a measurable fraction of a second. We ignore this overhead in [Figure 9-2](#), as this cost will be a tiny fraction of the overall execution time.

Example 9-2. main for estimating pi using a loop

```
from multiprocessing import Pool
...
if __name__ == "__main__":
    nbr_samples_in_total = 1e8
```

```

nbr_parallel_blocks = 4
pool = Pool(processes=nbr_parallel_blocks)
nbr_samples_per_worker = nbr_samples_in_total / nbr_parallel_blocks
print("Making {:,} samples per {} worker".format(nbr_samples_per_worker,
                                                nbr_parallel_blocks))
nbr_trials_per_process = [nbr_samples_per_worker] * nbr_parallel_blocks
t1 = time.time()
nbr_in_quarter_unit_circles = pool.map(estimate_nbr_points_in_quarter_circle,
                                         nbr_trials_per_process)
pi_estimate = sum(nbr_in_quarter_unit_circles) * 4 / float(nbr_samples_in_total)
print("Estimated pi", pi_estimate)
print("Delta:", time.time() - t1)

```

We create a list containing `nbr_estimates` divided by the number of workers. This new argument will be sent to each worker. After execution, we'll receive the same number of results back; we'll sum these to estimate the number of darts in the unit circle.

We import the process-based `Pool` from `multiprocessing`. We also could have used `from multiprocessing.dummy import Pool` to get a threaded version. The “dummy” name is rather misleading (we confess to not understanding why it is named this way); it is simply a light wrapper around the `threading` module to present the same interface as the process-based `Pool`.



Each process we create consumes some RAM from the system. You can expect a forked process using the standard libraries to take on the order of 10–20 MB of RAM; if you're using many libraries and lots of data, you might expect each forked copy to take hundreds of megabytes. On a system with a RAM constraint, this might be a significant issue—if you run out of RAM and the system reverts to using the disk's swap space, any parallelization advantage will be massively lost to the slow paging of RAM back and forth to disk!

The following figures plot the average CPU utilization of Ian's laptop's four physical cores and their four associated hyperthreads (each hyperthread runs on unutilized silicon in a physical core). The data gathered for these figures *includes* the startup time of the first Python process and the cost of starting subprocesses. The CPU sampler records the entire state of the laptop, not just the CPU time used by this task.

Note that the following diagrams are created using a different timing method with a slower sampling rate than [Figure 9-2](#), so the overall runtime is a little longer.

The execution behavior in [Figure 9-3](#) with one process in the Pool (along with the parent process) shows some overhead in the first seconds as the Pool is created, and then a consistent close-to-100% CPU utilization throughout the run. With one process, we're efficiently using one core.

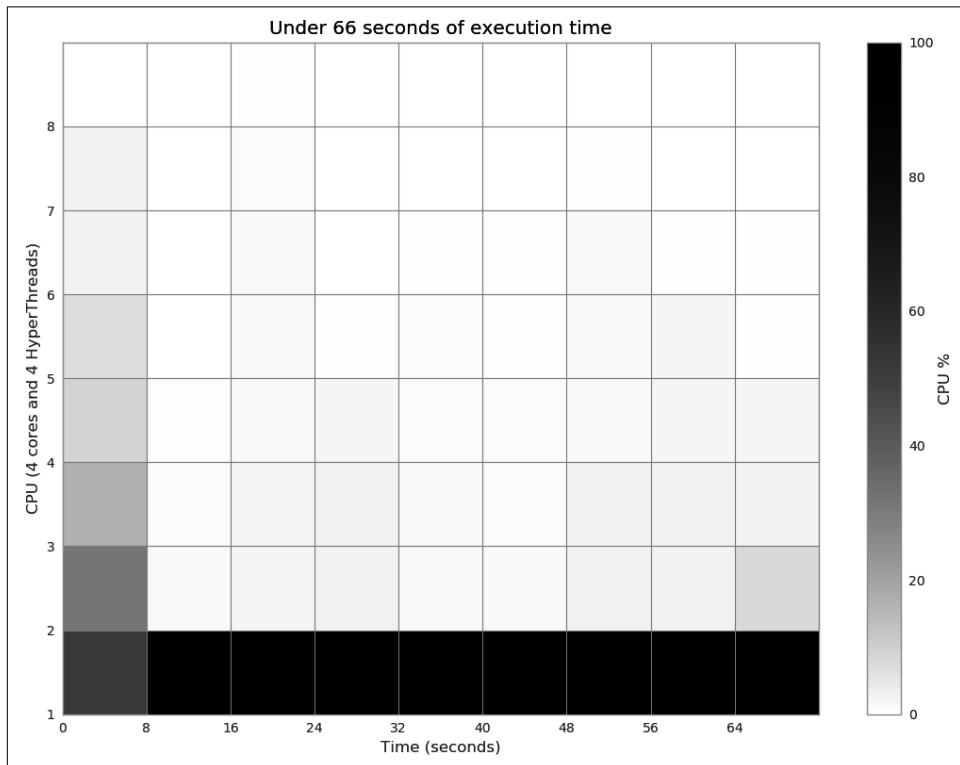


Figure 9-3. Estimating pi using Python objects and one process

Next we'll add a second process, effectively saying `Pool(processes=2)`. As you can see in [Figure 9-4](#), adding a second process roughly halves the execution time to 37 seconds, and two CPUs are fully occupied. This is the best result we can expect—we've efficiently used all the new computing resources, and we're not losing any speed to other overheads like communication, paging to disk, or contention with competing processes that want to use the same CPUs.

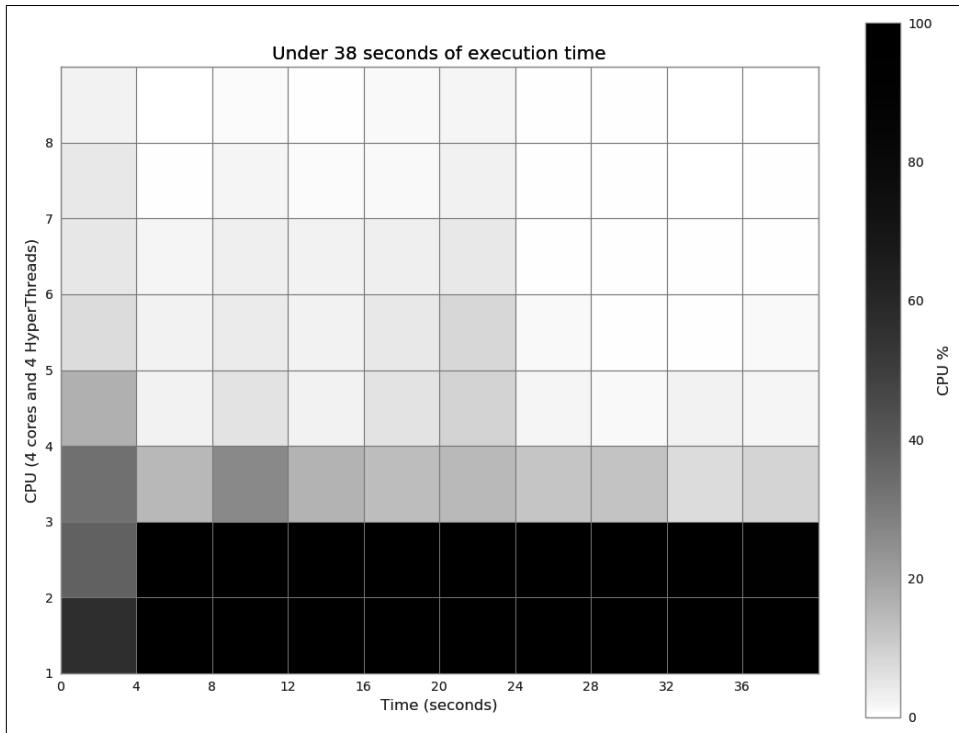


Figure 9-4. Estimating pi using Python objects and two processes

Figure 9-5 shows the results when using all four physical CPUs—now we are using all of the raw power of this laptop. Execution time is roughly a quarter that of the single-process version, at 19 seconds.

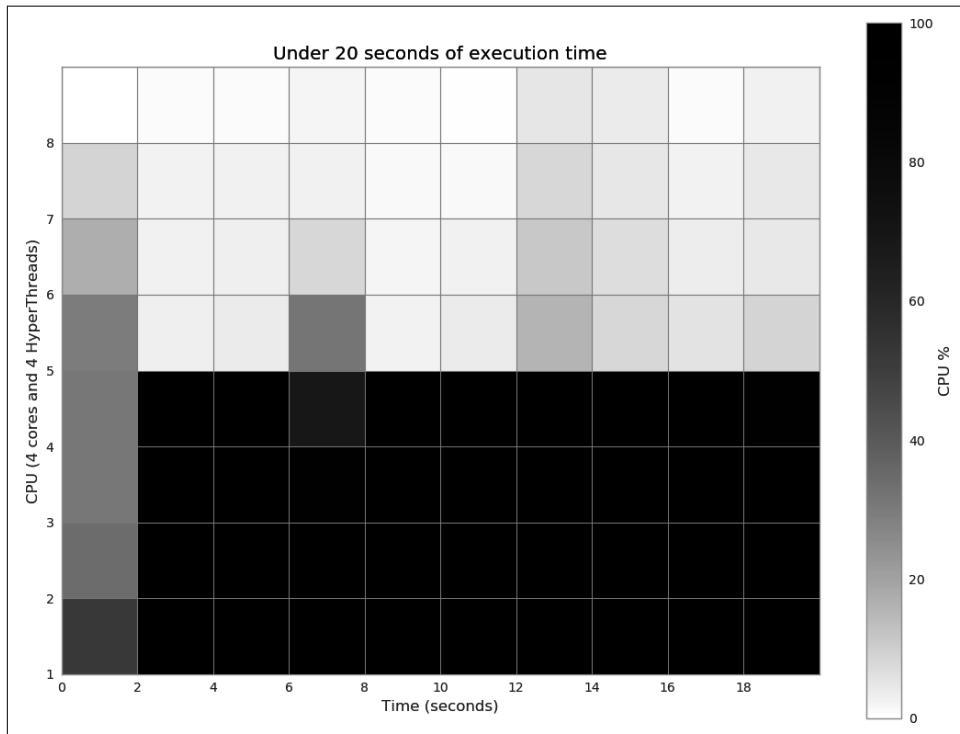


Figure 9-5. Estimating pi using Python objects and four processes

By switching to eight processes, as seen in [Figure 9-6](#), we cannot achieve more than a tiny speedup compared to the four-process version. That is because the four hyperthreads are able to squeeze only a little extra processing power out of the spare silicon on the CPUs, and the four CPUs are already maximally utilized.

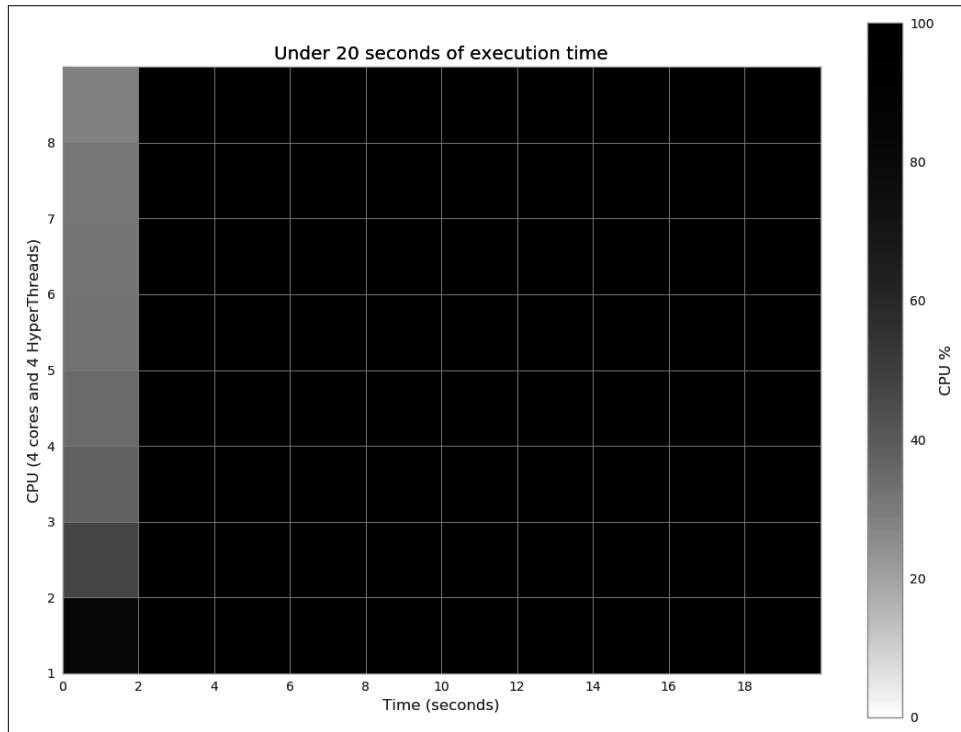


Figure 9-6. Estimating pi using Python objects and eight processes, with little additional gain

These diagrams show that we're efficiently using more of the available CPU resources at each step, and that the hyperthread resources are a poor addition. The biggest problem when using hyperthreads is that CPython is using a lot of RAM—hyperthreading is not cache friendly, so the spare resources on each chip are very poorly utilized. As we'll see in the next section, `numpy` makes better use of these resources.



In our experience, hyperthreading can give up to a 30% performance gain *if* there are enough spare computing resources. This works if, for example, you have a mix of floating-point and integer arithmetic rather than just the floating-point operations we have here. By mixing the resource requirements, the hyperthreads can schedule more of the CPU's silicon to be working concurrently. Generally, we see hyperthreads as an added bonus and not a resource to be optimized against, as adding more CPUs is probably more economical than tuning your code (which adds a support overhead).

Now we'll switch to using threads in one process, rather than multiple processes.

[Figure 9-7](#) shows the results of running the same code that we used in [Figure 9-5](#), but with threads in place of processes. Although a number of CPUs are being used, they each share the workload lightly. If each thread was running without the GIL, then we'd see 100% CPU utilization on the four CPUs. Instead, each CPU is partially utilized (because of the GIL).

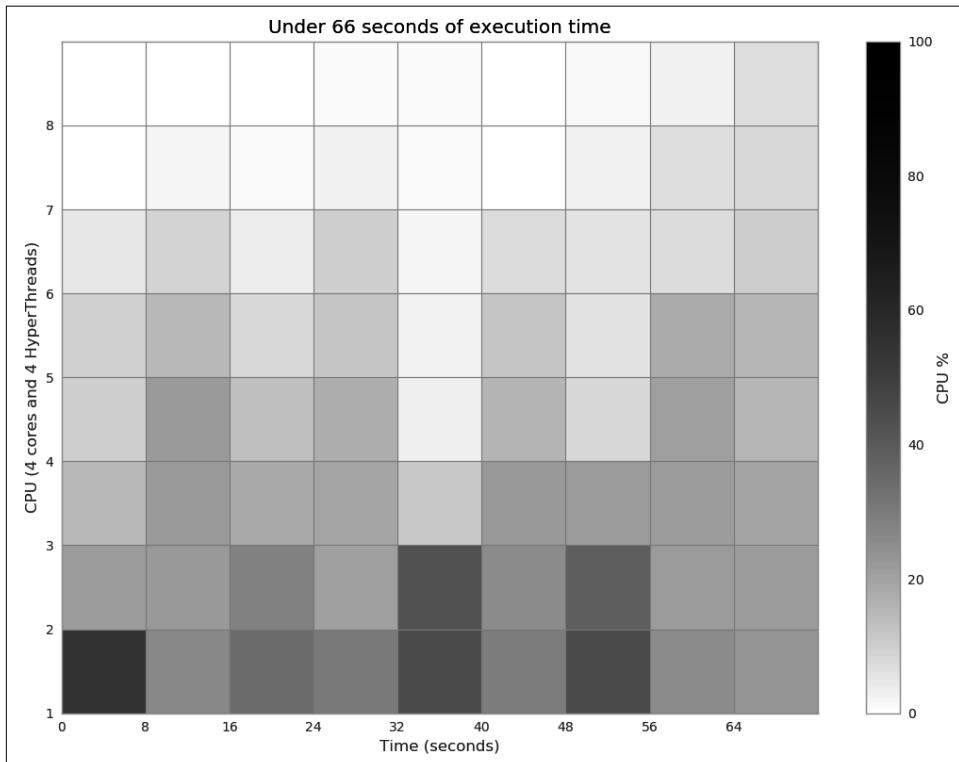


Figure 9-7. Estimating pi using Python objects and four threads

Replacing multiprocessing with Joblib

Joblib is an improvement on `multiprocessing` that enables lightweight pipelining with a focus on easy parallel computing and transparent disk-based caching of results. It focuses on NumPy arrays for scientific computing. It may offer a quick win for you if you're

- Using pure Python, with or without NumPy, to process a loop that could be embarrassingly parallel
- Calling expensive functions that have no side effects, where the output could be cached to disk between sessions
- Able to share NumPy data between processes but don't know how (and you haven't yet read “[Sharing numpy Data with multiprocessing](#)” on page 295)

Joblib builds on the Loky library (itself an improvement over Python's `concurrent.futures`) and uses `cloudpickle` to enable the pickling of functions defined in the interactive scope. This solves a couple of common issues that are encountered with the built-in `multiprocessing` library.

For parallel computing, we need the `Parallel` class and the `delayed` decorator. The `Parallel` class sets up a process pool, similar to the `multiprocessing pool` we used in the previous section. The `delayed` decorator wraps our target function so it can be applied to the instantiated `Parallel` object via an iterator.

The syntax is a little confusing to read—take a look at [Example 9-3](#). The call is written on one line; this includes our target function `estimate_nbr_points_in_quarter_circle` and the iterator (`delayed(...)(nbr_samples_per_worker)` for `sample_idx` in `range(nbr_parallel_blocks)`). Let's break this down.

Example 9-3. Using Joblib to parallelize pi estimation

```
...
from joblib import Parallel, delayed

if __name__ == "__main__":
    ...
    nbr_in_quarter_unit_circles = Parallel(n_jobs=nbr_parallel_blocks, verbose=1) \
        (delayed(estimate_nbr_points_in_quarter_circle)(nbr_samples_per_worker) \
         for sample_idx in range(nbr_parallel_blocks))
    ...

```

`Parallel` is a class; we can set parameters such as `n_jobs` to dictate how many processes will run, along with optional arguments like `verbose` for debugging information. Other arguments can set time-outs, change between threads or processes,

change the backends (which can help speed up certain edge cases), and configure memory mapping.

Parallel has a `__call__` callable method that takes an iterable. We supply the iterable in the following round brackets (`... for sample_idx in range(...)`). The callable iterates over each `delayed(estimate_nbr_points_in_quarter_circle)` function, batching the execution of these functions to their arguments (in this case, `nbr_samples_per_worker`). Ian has found it helpful to build up a parallelized call one step at a time, starting from a function with no arguments and building up arguments as needed. This makes diagnosing missteps much easier.

`nbr_in_quarter_unit_circles` will be a list containing the count of positive cases for each call as before. [Example 9-4](#) shows the console output for eight parallel blocks; each process ID (PID) is freshly created, and a summary is printed in a progress bar at the end of the output. In total this takes 19 seconds, the same amount of time as when we created our own Pool in the previous section.



Avoid passing large structures; passing large pickled objects to each process may be expensive. Ian had a case with a prebuilt cache of Pandas DataFrames in a dictionary object; the cost of serializing these via the `Pickle` module negated the gains from parallelization, and the serial version actually worked faster overall. The solution in this case was to build the DataFrame cache using Python's built-in `shelve` module, storing the dictionary to a file. A single DataFrame was loaded with `shelve` on each call to the target function; hardly anything had to be passed to the functions, and then the parallelized benefit of Joblib was clear.

Example 9-4. Output of Joblib calls

```
Making 12,500,000 samples per 8 worker
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10313
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10315
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10311
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10316
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10312
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10314
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10317
Executing estimate_nbr_points_in_quarter_circle with 12,500,000 on pid 10318
[Parallel(n_jobs=8)]: Done  2 out of  8 | elapsed:   18.9s remaining:   56.6s
[Parallel(n_jobs=8)]: Done  8 out of  8 | elapsed:   19.3s finished
Estimated pi 3.14157744
Delta: 19.32842755317688
```



To simplify debugging, we can set `n_jobs=1`, and the parallelized code is dropped. You don't have to modify your code any further, and you can drop a call to `breakpoint()` in your function to ease your debugging.

Intelligent caching of function call results

A useful feature in Joblib is the `Memory` cache; this is a decorator that caches function results based on the input arguments to a disk cache. This cache persists between Python sessions, so if you turn off your machine and then run the same code the next day, the cached results will be used.

For our `pi` estimation, this presents a small problem. We don't pass in unique arguments to `estimate_nbr_points_in_quarter_circle`; for each call we pass in `nbr_estimates`, so the call signature is the same, but we're after different results.

In this situation, once the first call has completed (taking around 19 seconds), any subsequent call with the same argument will get the cached result. This means that if we rerun our code a second time, it completes instantly, but it uses only one of the eight sample results as the result for each call—this obviously breaks our Monte Carlo sampling! If the last process to complete resulted in 9815738 points in the quarter circle, the cache for the function call would always answer this. Repeating the call eight times would generate [9815738, 9815738, 9815738, 9815738, 9815738, 9815738, 9815738, 9815738] rather than eight unique estimates.

The solution in [Example 9-5](#) is to pass in a second argument, `idx`, which takes on a value between 0 and `nbr_parallel_blocks-1`. This unique combination of arguments will let the cache store each positive count, so that on the second run we get the same result as on the first run, but without the wait.

This is configured using `Memory`, which takes a folder for persisting the function results. This persistence is kept between Python sessions; it is refreshed if you change the function that is being called, or if you empty the files in the cache folder.

Note that this refresh applies only to a change to the function that's been decorated (in this case, `estimate_nbr_points_in_quarter_circle_with_idx`), not to any sub-functions that are called from inside that function.

Example 9-5. Caching results with Joblib

```
...
from joblib import Memory

memory = Memory("./joblib_cache", verbose=0)

@memory.cache
```

```

def estimate_nbr_points_in_quarter_circle_with_idx(nbr_estimates, idx):
    print(f"Executing estimate_nbr_points_in_quarter_circle with \
          {nbr_estimates} on sample {idx} on pid {os.getpid()}")
    ...

if __name__ == "__main__":
    ...
    nbr_in_quarter_unit_circles = Parallel(n_jobs=nbr_parallel_blocks) \
        (delayed(estimate_nbr_points_in_quarter_circle_with_idx) \
         (nbr_samples_per_worker, idx) for idx in range(nbr_parallel_blocks))
    ...

```

In Example 9-6, we can see that while the first call costs 19 seconds, the second call takes only a fraction of a second and has the same estimated pi. In this run, the estimates were [9817605, 9821064, 9818420, 9817571, 9817688, 9819788, 9816377, 9816478].

Example 9-6. The zero-cost second call to the code thanks to cached results

```

$ python pi_lists_parallel_joblib_cache.py
Making 12,500,000 samples per 8 worker
Executing estimate_nbr_points_in_... with 12500000 on sample 0 on pid 10672
Executing estimate_nbr_points_in_... with 12500000 on sample 1 on pid 10676
Executing estimate_nbr_points_in_... with 12500000 on sample 2 on pid 10677
Executing estimate_nbr_points_in_... with 12500000 on sample 3 on pid 10678
Executing estimate_nbr_points_in_... with 12500000 on sample 4 on pid 10679
Executing estimate_nbr_points_in_... with 12500000 on sample 5 on pid 10674
Executing estimate_nbr_points_in_... with 12500000 on sample 6 on pid 10673
Executing estimate_nbr_points_in_... with 12500000 on sample 7 on pid 10675
Estimated pi 3.14179964
Delta: 19.28862953186035

$ python %run pi_lists_parallel_joblib_cache.py
Making 12,500,000 samples per 8 worker
Estimated pi 3.14179964
Delta: 0.02478170394897461

```

Joblib wraps up a lot of `multiprocessing` functionality with a simple (if slightly hard to read) interface. Ian has moved to using Joblib in favor of `multiprocessing`; he recommends you try it too.

Random Numbers in Parallel Systems

Generating good random number sequences is a hard problem, and it is easy to get it wrong if you try to do it yourself. Getting a good sequence quickly in parallel is even harder—suddenly you have to worry about whether you’ll get repeating or correlated sequences in the parallel processes.

We used Python’s built-in random number generator in [Example 9-1](#), and we’ll use the `numpy` random number generator in [Example 9-7](#) in the next section. In both cases, the random number generators are seeded in their forked process. For the Python `random` example, the seeding is handled internally by `multiprocessing`—if during a fork it sees that `random` is in the namespace, it will force a call to seed the generators in each of the new processes.



Set the `numpy` seed when parallelizing your function calls. In the forthcoming `numpy` example, we have to explicitly set the random number seed. If you forget to seed the random number sequence with `numpy`, each of your forked processes will generate an identical sequence of random numbers—it’ll appear to be working as you wanted it to, but behind the scenes each parallel process will evolve with identical results!

If you care about the quality of the random numbers used in the parallel processes, we urge you to research this topic. *Probably* the `numpy` and Python random number generators are good enough, but if significant outcomes depend on the quality of the random sequences (e.g., for medical or financial systems), then you must read up on this area.

In Python 3, the [Mersenne Twister algorithm](#) is used—it has a long period, so the sequence won’t repeat for a long time. It is heavily tested, as it is used in other languages, and it is thread-safe. It is probably not suitable for cryptographic purposes.

Using `numpy`

In this section, we switch to using `numpy`. Our dart-throwing problem is ideal for `numpy` vectorized operations—we generate the same estimate 25 times faster than the previous Python examples.

The main reason that `numpy` is faster than pure Python when solving the same problem is that `numpy` is creating and manipulating the same object types at a very low level in contiguous blocks of RAM, rather than creating many higher-level Python objects that each require individual management and addressing.

As `numpy` is far more cache friendly, we’ll also get a small speed boost when using the four hyperthreads. We didn’t get this in the pure Python version, as caches aren’t used efficiently by larger Python objects.

In [Figure 9-8](#), we see three scenarios:

- No use of `multiprocessing` (named “Serial”)
- Using threads

- Using processes

The serial and single-worker versions execute at the same speed—there's no overhead to using threads with numpy (and with only one worker, there's also no gain).

When using multiple processes, we see a classic 100% utilization of each additional CPU. The result mirrors the plots shown in Figures 9-3, 9-4, 9-5, and 9-6, but the code is much faster using numpy.

Interestingly, the threaded version runs *faster* with more threads. As discussed on the [SciPy wiki](#), by working outside the GIL, numpy can achieve some level of additional speedup around threads.

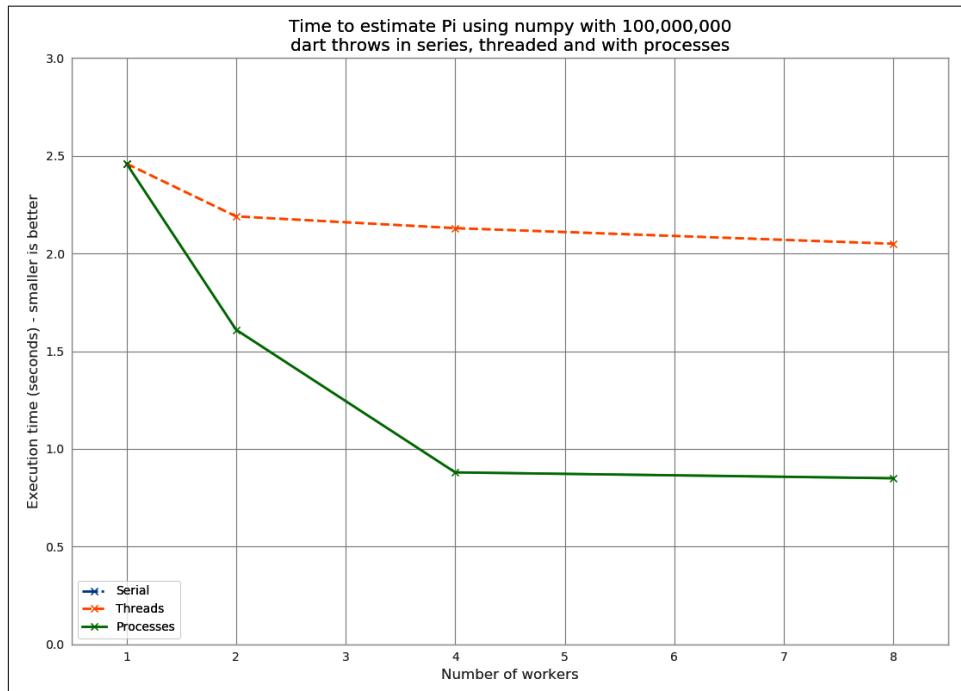


Figure 9-8. Working in series, with threads, and with processes using numpy

Using processes gives us a predictable speedup, just as it did in the pure Python example. A second CPU doubles the speed, and using four CPUs quadruples the speed.

[Example 9-7](#) shows the vectorized form of our code. Note that the random number generator is seeded when this function is called. For the threaded version, this isn't necessary, as each thread shares the same random number generator and they access it in series. For the process version, as each new process is a fork, all the forked

versions will share the *same state*. This means the random number calls in each will return the same sequence!



Remember to call `seed()` per process with `numpy` to ensure that each of the forked processes generates a unique sequence of random numbers, as a random source is used to set the seed for each call. Look back at “[Random Numbers in Parallel Systems](#)” on page 263 for some notes about the dangers of parallelized random number sequences.

Example 9-7. Estimating pi using numpy

```
def estimate_nbr_points_in_quarter_circle(nbr_samples):
    """Estimate pi using vectorized numpy arrays"""
    np.random.seed() # remember to set the seed per process
    xs = np.random.uniform(0, 1, nbr_samples)
    ys = np.random.uniform(0, 1, nbr_samples)
    estimate_inside_quarter_unit_circle = (xs * xs + ys * ys) <= 1
    nbr_trials_in_quarter_unit_circle = np.sum(estimate_inside_quarter_unit_circle)
    return nbr_trials_in_quarter_unit_circle
```

A short code analysis shows that the calls to `random` run a little slower on this machine when executed with multiple threads, and the call to `(xs * xs + ys * ys) <= 1` parallelizes well. Calls to the random number generator are GIL-bound, as the internal state variable is a Python object.

The process to understand this was basic but reliable:

1. Comment out all of the `numpy` lines, and run with *no* threads using the serial version. Run several times and record the execution times using `time.time()` in `__main__`.
2. Add a line back (we added `xs = np.random.uniform(...)` first) and run several times, again recording completion times.
3. Add the next line back (now adding `ys = ...`), run again, and record completion time.
4. Repeat, including the `nbr_trials_in_quarter_unit_circle = np.sum(...)` line.
5. Repeat this process again, but this time with four threads. Repeat line by line.
6. Compare the difference in runtime at each step for no threads and four threads.

Because we’re running code in parallel, it becomes harder to use tools like `line_profiler` or `cProfile`. Recording the raw runtimes and observing the differences in behavior with different configurations takes patience but gives solid evidence from which to draw conclusions.



If you want to understand the serial behavior of the `uniform` call, take a look at the `mtrand` code in the `numpy` source and follow the call to `def uniform` in `mtrand.pyx`. This is a useful exercise if you haven't looked at the `numpy` source code before.

The libraries used when building `numpy` are important for some of the parallelization opportunities. Depending on the underlying libraries used when building `numpy` (e.g., whether Intel's Math Kernel Library or OpenBLAS were included or not), you'll see different speedup behavior.

You can check your `numpy` configuration using `numpy.show_config()`. Stack Overflow has some [example timings](#) if you're curious about the possibilities. Only some `numpy` calls will benefit from parallelization by external libraries.

Finding Prime Numbers

Next, we'll look at testing for prime numbers over a large number range. This is a different problem from estimating pi, as the workload varies depending on your location in the number range, and each individual number's check has an unpredictable complexity. We can create a serial routine that checks for primality and then pass sets of possible factors to each process for checking. This problem is embarrassingly parallel, which means there is no state that needs to be shared.

The `multiprocessing` module makes it easy to control the workload, so we shall investigate how we can tune the work queue to use (and misuse!) our computing resources, and we will explore an easy way to use our resources slightly more efficiently. This means we'll be looking at *load balancing* to try to efficiently distribute our varying-complexity tasks to our fixed set of resources.

We'll use an algorithm that is slightly removed from the one earlier in the book (see "[Idealized Computing Versus the Python Virtual Machine](#)" on page 10); it exits early if we have an even number—see Example 9-8.

Example 9-8. Finding prime numbers using Python

```
def check_prime(n):
    if n % 2 == 0:
        return False
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False
    return True
```

How much variety in the workload do we see when testing for a prime with this approach? [Figure 9-9](#) shows the increasing time cost to check for primality as the possibly prime n increases from 10,000 to 1,000,000.

Most numbers are nonprime; they're drawn with a dot. Some can be cheap to check for, while others require the checking of many factors. Primes are drawn with an x and form the thick darker band; they're the most expensive to check for. The time cost of checking a number increases as n increases, as the range of possible factors to check increases with the square root of n . The sequence of primes is not predictable, so we can't determine the expected cost of a range of numbers (we could estimate it, but we can't be sure of its complexity).

For the figure, we test each n two hundred times and take the fastest result to remove jitter from the results. If we took only one result, we'd see wide variance in timing that would be caused by system load from other processes; by taking many readings and keeping the fastest, we get to see the expected best-case timing.

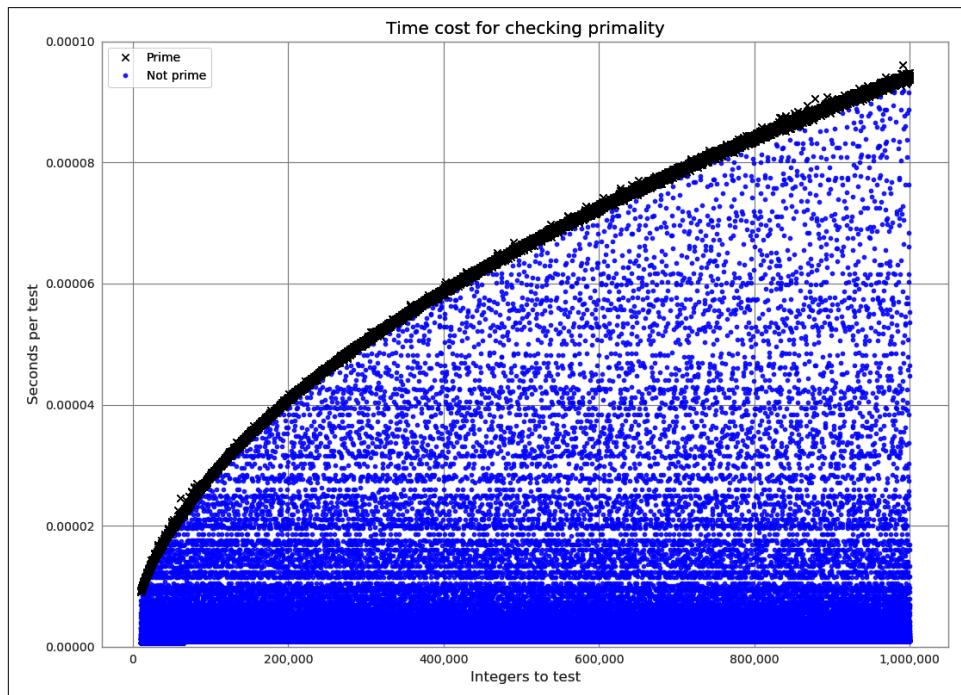


Figure 9-9. Time required to check primality as n increases

When we distribute work to a Pool of processes, we can specify how much work is passed to each worker. We could divide all of the work evenly and aim for one pass, or we could make many chunks of work and pass them out whenever a CPU is free. This is controlled using the `chunksize` parameter. Larger chunks of work mean less

communication overhead, while smaller chunks of work mean more control over how resources are allocated.

For our prime finder, a single piece of work is a number n that is checked by `check_prime`. A `chunksize` of 10 would mean that each process handles a list of 10 integers, one list at a time.

In [Figure 9-10](#), we can see the effect of varying the `chunksize` from 1 (every job is a single piece of work) to 64 (every job is a list of 64 numbers). Although having many tiny jobs gives us the greatest flexibility, it also imposes the greatest communication overhead. All four CPUs will be utilized efficiently, but the communication pipe will become a bottleneck as each job and result is passed through this single channel. If we double the `chunksize` to 2, our task gets solved twice as quickly, as we have less contention on the communication pipe. We might naively assume that by increasing the `chunksize`, we will continue to improve the execution time. However, as you can see in the figure, we will again come to a point of diminishing returns.

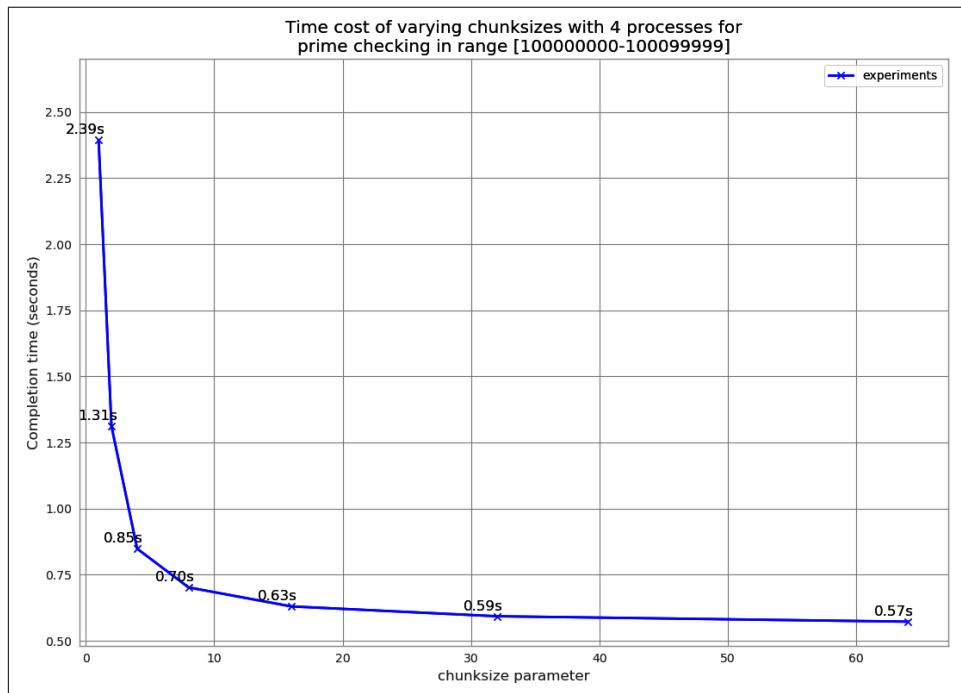


Figure 9-10. Choosing a sensible `chunksize` value

We can continue to increase the `chunksize` until we start to see a worsening of behavior. In [Figure 9-11](#), we expand the range of `chunksizes`, making them not just tiny but also huge. At the larger end of the scale, the worst result shown is 1.08 seconds, where we've asked for `chunksize` to be 50000—this means our 100,000 items

are divided into two work chunks, leaving two CPUs idle for that entire pass. With a chunksize of 10000 items, we are creating ten chunks of work; this means that four chunks of work will run twice in parallel, followed by the two remaining chunks. This leaves two CPUs idle in the third round of work, which is an inefficient usage of resources.

An optimal solution in this case is to divide the total number of jobs by the number of CPUs. This is the default behavior in `multiprocessing`, shown as the “default” blue dot in the figure.

As a general rule, the default behavior is sensible; tune it only if you expect to see a real gain, and definitely confirm your hypothesis against the default behavior.

Unlike the Monte Carlo pi problem, our prime testing calculation has varying complexity—sometimes a job exits quickly (an even number is detected the fastest), and sometimes the number is large and a prime (this takes a much longer time to check).

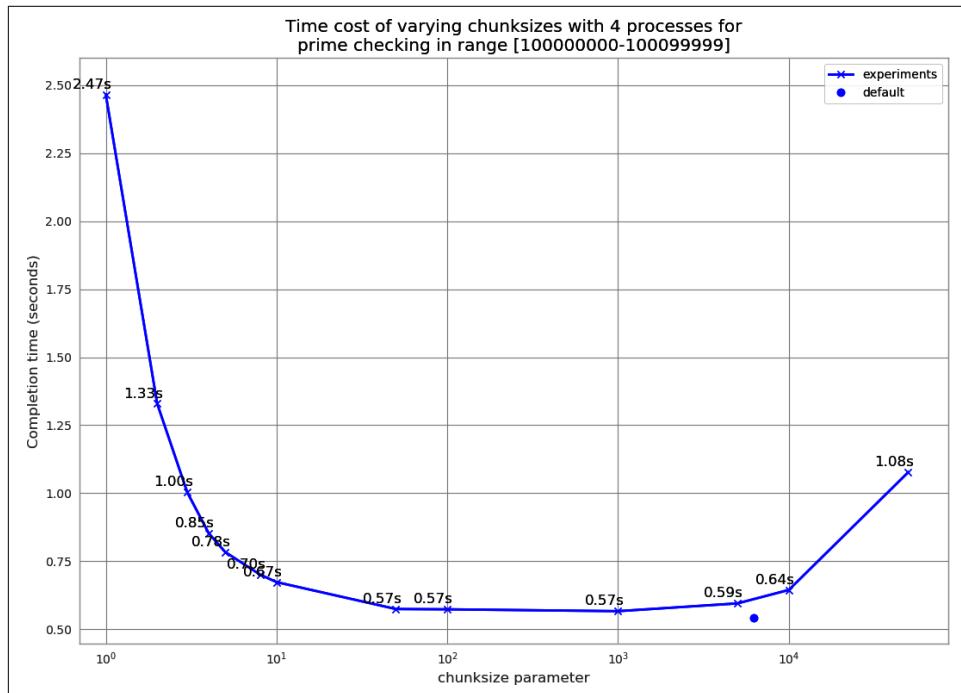


Figure 9-11. Choosing a sensible `chunksize` value (continued)

What happens if we randomize our job sequence? For this problem, we squeeze out a 2% performance gain, as you can see in [Figure 9-12](#). By randomizing, we reduce the likelihood of the final job in the sequence taking longer than the others, leaving all but one CPU active.

As our earlier example using a chunksize of 10000 demonstrated, misaligning the workload with the number of available resources leads to inefficiency. In that case, we created three rounds of work: the first two rounds used 100% of the resources, and the last round used only 50%.

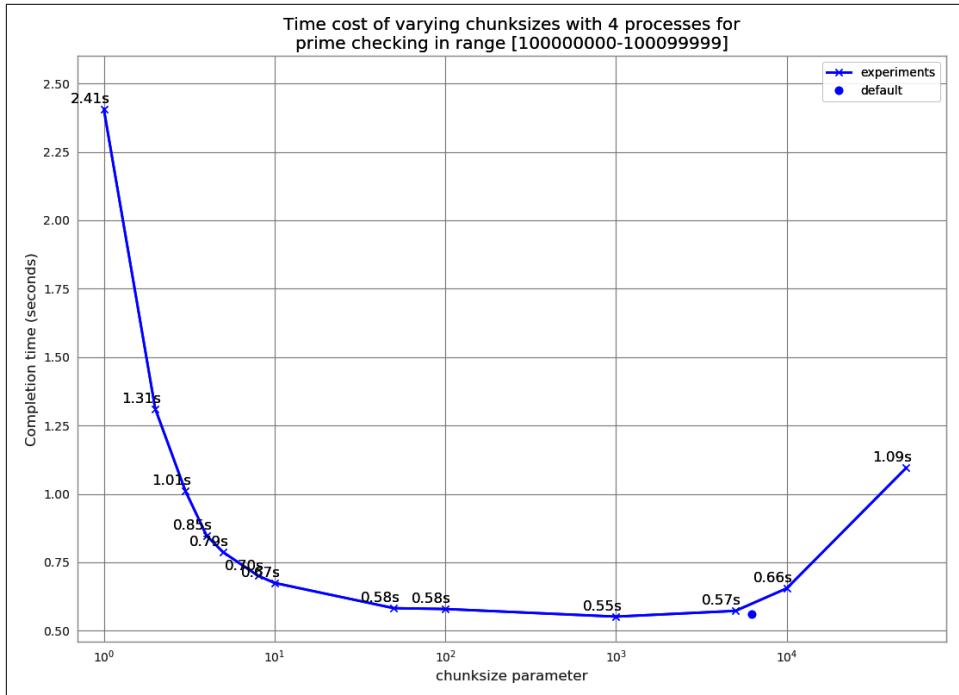


Figure 9-12. Randomizing the job sequence

Figure 9-13 shows the odd effect that occurs when we misalign the number of chunks of work against the number of processors. Mismatches will underutilize the available resources. The slowest overall runtime occurs when only one chunk of work is created: this leaves three unutilized. Two work chunks leave two CPUs unutilized, and so on; only when we have four work chunks are we using all of our resources. But if we add a fifth work chunk, we’re underutilizing our resources again—four CPUs will work on their chunks, and then one CPU will run to calculate the fifth chunk.

As we increase the number of chunks of work, we see that the inefficiencies decrease—the difference in runtime between 29 and 32 work chunks is approximately 0.03 seconds. The general rule is to make lots of small jobs for efficient resource utilization if your jobs have varying runtimes.

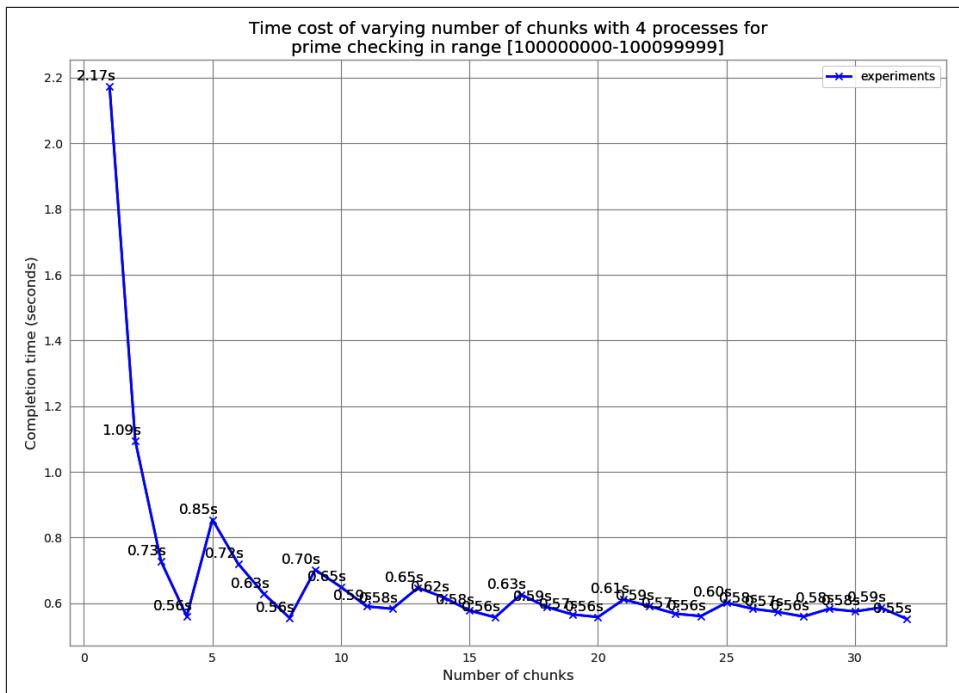


Figure 9-13. The danger of choosing an inappropriate number of chunks

Here are some strategies for efficiently using `multiprocessing` for embarrassingly parallel problems:

- Split your jobs into independent units of work.
- If your workers take varying amounts of time, consider randomizing the sequence of work (another example would be for processing variable-sized files).
- Sorting your work queue so that the slowest jobs go first may be an equally useful strategy.
- Use the default `chunksize` unless you have verified reasons for adjusting it.
- Align the number of jobs with the number of physical CPUs. (Again, the default `chunksize` takes care of this for you, although it will use any hyperthreads by default, which may not offer any additional gain.)

Note that by default `multiprocessing` will see hyperthreads as additional CPUs. This means that on Ian’s laptop, it will allocate eight processes when only four will really be running at 100% speed. The additional four processes could be taking up valuable RAM while barely offering any additional speed gain.

With a `Pool`, we can split up a chunk of predefined work up front among the available CPUs. This is less helpful if we have dynamic workloads, though, and particularly if we have workloads that arrive over time. For this sort of workload, we might want to use a `Queue`, introduced in the next section.

Queues of Work

`multiprocessing.Queue` objects give us nonpersistent queues that can send any pickleable Python objects between processes. They carry an overhead, as each object must be pickled to be sent and then unpickled in the consumer (along with some locking operations). In the following example, we'll see that this cost is not negligible. However, if your workers are processing larger jobs, the communication overhead is probably acceptable.

Working with the queues is fairly easy. In this example, we'll check for primes by consuming a list of candidate numbers and posting confirmed primes back to a `definite_primes_queue`. We'll run this with one, two, four, and eight processes and confirm that the latter three approaches all take longer than just running a single process that checks the same range.

A `Queue` gives us the ability to perform lots of interprocess communication using native Python objects. This can be useful if you're passing around objects with lots of state. Since the `Queue` lacks persistence, though, you probably don't want to use queues for jobs that might require robustness in the face of failure (e.g., if you lose power or a hard drive gets corrupted).

Example 9-9 shows the `check_prime` function. We're already familiar with the basic primality test. We run in an infinite loop, blocking (waiting until work is available) on `possible_primes_queue.get()` to consume an item from the queue. Only one process can get an item at a time, as the `Queue` object takes care of synchronizing the accesses. If there's no work in the queue, the `.get()` blocks until a task is available. When primes are found, they are put back on the `definite_primes_queue` for consumption by the parent process.

Example 9-9. Using two queues for interprocess communication (IPC)

```
FLAG_ALL_DONE = b"WORK_FINISHED"
FLAG_WORKER_FINISHED_PROCESSING = b"WORKER_FINISHED_PROCESSING"

def check_prime(possible_primes_queue, definite_primes_queue):
    while True:
        n = possible_primes_queue.get()
        if n == FLAG_ALL_DONE:
            # flag that our results have all been pushed to the results queue
            definite_primes_queue.put(FLAG_WORKER_FINISHED_PROCESSING)
```

```

        break
else:
    if n % 2 == 0:
        continue
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            break
    else:
        definite_primes_queue.put(n)

```

We define two flags: one is fed by the parent process as a poison pill to indicate that no more work is available, while the second is fed by the worker to confirm that it has seen the poison pill and has closed itself down. The first poison pill is also known as a *sentinel*, as it guarantees the termination of the processing loop.

When dealing with queues of work and remote workers, it can be helpful to use flags like these to record that the poison pills were sent and to check that responses were sent from the children in a sensible time window, indicating that they are shutting down. We don't handle that process here, but adding some timekeeping is a fairly simple addition to the code. The receipt of these flags can be logged or printed during debugging.

The Queue objects are created out of a Manager in [Example 9-10](#). We'll use the familiar process of building a list of Process objects that each contain a forked process. The two queues are sent as arguments, and multiprocessing handles their synchronization. Having started the new processes, we hand a list of jobs to the possible_primes_queue and end with one poison pill per process. The jobs will be consumed in FIFO order, leaving the poison pills for last. In check_prime we use a blocking .get(), as the new processes will have to wait for work to appear in the queue. Since we use flags, we could add some work, deal with the results, and then iterate by adding more work, and signal the end of life of the workers by adding the poison pills later.

Example 9-10. Building two queues for IPC

```

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Project description")
    parser.add_argument(
        "nbr_workers", type=int, help="Number of workers e.g. 1, 2, 4, 8"
    )
    args = parser.parse_args()
    primes = []

    manager = multiprocessing.Manager()
    possible_primes_queue = manager.Queue()
    definite_primes_queue = manager.Queue()

```

```

pool = Pool(processes=args.nbr_workers)
processes = []
for _ in range(args.nbr_workers):
    p = multiprocessing.Process(
        target=check_prime, args=(possible_primes_queue,
                                    definite_primes_queue)
    )
    processes.append(p)
    p.start()

t1 = time.time()
number_range = range(100_000_000, 101_000_000)

# add jobs to the inbound work queue
for possible_prime in number_range:
    possible_primes_queue.put(possible_prime)

# add poison pills to stop the remote workers
for n in range(args.nbr_workers):
    possible_primes_queue.put(FLAGS_ALL_DONE)

```

To consume the results, we start another infinite loop in [Example 9-11](#), using a blocking `.get()` on the `definite_primes_queue`. If the finished-processing flag is found, we take a count of the number of processes that have signaled their exit. If not, we have a new prime, and we add this to the `primes` list. We exit the infinite loop when all of our processes have signaled their exit.

Example 9-11. Using two queues for IPC

```

processors_indicating_they_have_finished = 0
while True:
    new_result = definite_primes_queue.get() # block while waiting for results
    if new_result == FLAG_WORKER_FINISHED_PROCESSING:
        processors_indicating_they_have_finished += 1
        if processors_indicating_they_have_finished == args.nbr_workers:
            break
    else:
        primes.append(new_result)
assert processors_indicating_they_have_finished == args.nbr_workers

print("Took:", time.time() - t1)
print(len(primes), primes[:10], primes[-10:])

```

There is quite an overhead to using a Queue, due to the pickling and synchronization. As you can see in [Figure 9-14](#), using a Queue-less single-process solution is significantly faster than using two or more processes. The reason in this case is because our workload is very light—the communication cost dominates the overall time for this task. With Queues, two processes complete this example a little faster than one process, while four and eight processes are both slower.

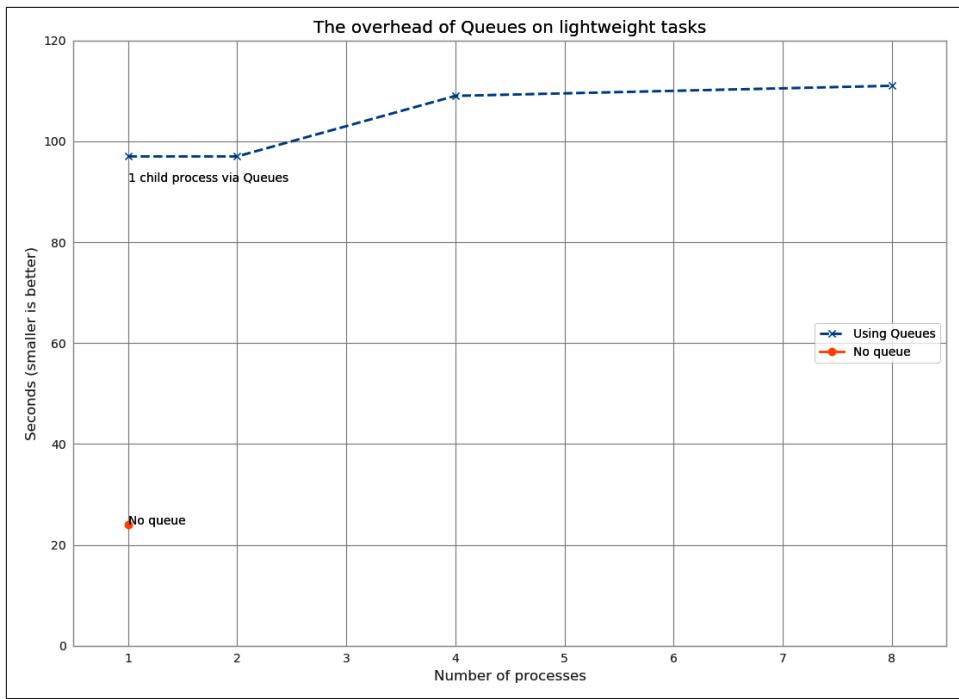


Figure 9-14. Cost of using Queue objects

If your task has a long completion time (at least a sizable fraction of a second) with a small amount of communication, a Queue approach might be the right answer. You will have to verify whether the communication cost makes this approach useful enough.

You might wonder what happens if we remove the redundant half of the job queue (all the even numbers—these are rejected very quickly in `check_prime`). Halving the size of the input queue halves our execution time in each case, but it still doesn't beat the single-process non-Queue example! This helps to illustrate that the communication cost is the dominating factor in this problem.

Asynchronously adding jobs to the Queue

By adding a Thread into the main process, we can feed jobs asynchronously into the `possible_primes_queue`. In [Example 9-12](#), we define a `feed_new_jobs` function: it performs the same job as the job setup routine that we had in `__main__` before, but it does it in a separate thread.

Example 9-12. Asynchronous job-feeding function

```
def feed_new_jobs(number_range, possible_primes_queue, nbr_poison_pills):
    for possible_prime in number_range:
        possible_primes_queue.put(possible_prime)
    # add poison pills to stop the remote workers
    for n in range(nbr_poison_pills):
        possible_primes_queue.put(FLAGS_ALL_DONE)
```

Now, in [Example 9-13](#), our `__main__` will set up the `Thread` using the `possible_primes_queue` and then move on to the result-collection phase *before* any work has been issued. The asynchronous job feeder could consume work from external sources (e.g., from a database or I/O-bound communication) while the `__main__` thread handles each processed result. This means that the input sequence and output sequence do not need to be created in advance; they can both be handled on the fly.

Example 9-13. Using a thread to set up an asynchronous job feeder

```
if __name__ == "__main__":
    primes = []
    manager = multiprocessing.Manager()
    possible_primes_queue = manager.Queue()

    ...

    import threading
    thrd = threading.Thread(target=feed_new_jobs,
                           args=(number_range,
                                  possible_primes_queue,
                                  NBR_PROCESSES))
    thrd.start()

    # deal with the results
```

If you want robust asynchronous systems, you should almost certainly look to using `asyncio` or an external library such as `tornado`. For a full discussion of these approaches, check out [Chapter 8](#). The examples we've looked at here will get you started, but pragmatically they are more useful for very simple systems and education than for production systems.

Be *very aware* that asynchronous systems require a special level of patience—you will end up tearing out your hair while you are debugging. We suggest the following:

- Applying the “Keep It Simple, Stupid” principle
- Avoiding asynchronous self-contained systems (like our example) if possible, as they will grow in complexity and quickly become hard to maintain

- Using mature libraries like `gevent` (described in the previous chapter) that give you tried-and-tested approaches to dealing with certain problem sets

Furthermore, we strongly suggest using an external queue system that gives you external visibility on the state of the queues (e.g., NSQ, discussed in “[NSQ for Robust Production Clustering](#)” on page 326; ZeroMQ; or Celery). This requires more thought but is likely to save you time because of increased debug efficiency and better system visibility for production systems.



Consider using a task graph for resilience. Data science tasks requiring long-running queues are frequently served well by specifying pipelines of work in acyclic graphs. Two strong libraries are [Airflow](#) and [Luigi](#). These are very frequently used in industrial settings and enable arbitrary task chaining, online monitoring, and flexible scaling.

Verifying Primes Using Interprocess Communication

Prime numbers are numbers that have no factor other than themselves and 1. It stands to reason that the most common factor is 2 (every even number cannot be a prime). After that, the low prime numbers (e.g., 3, 5, 7) become common factors of larger nonprimes (e.g., 9, 15, and 21, respectively).

Let’s say that we are given a large number and are asked to verify if it is prime. We will probably have a large space of factors to search. [Figure 9-15](#) shows the frequency of each factor for nonprimes up to 10,000,000. Low factors are far more likely to occur than high factors, but there’s no predictable pattern.

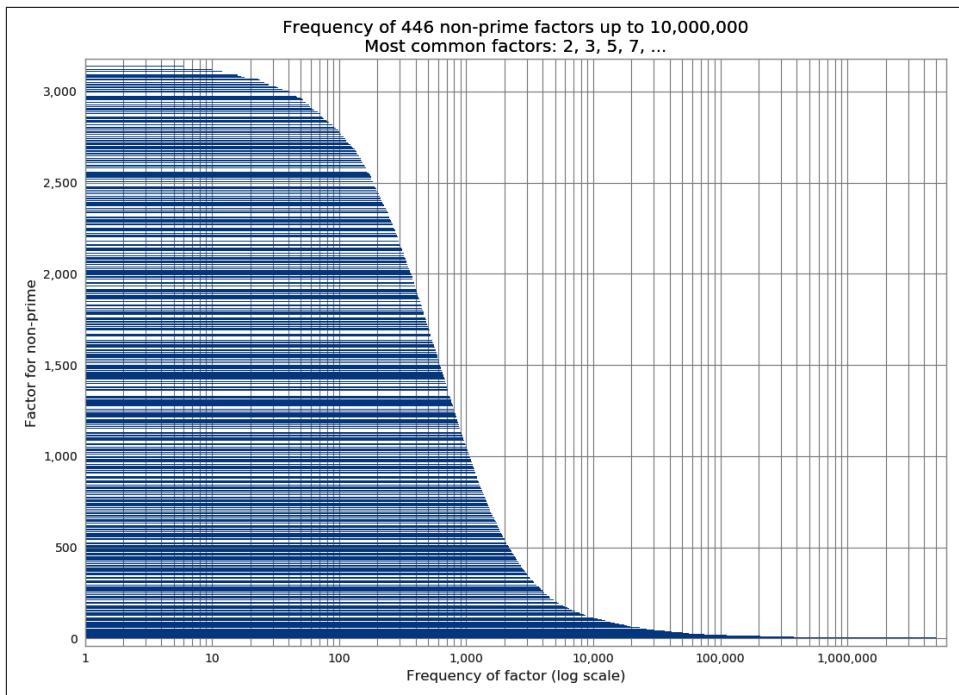


Figure 9-15. The frequency of factors of nonprimes

Let's define a new problem—suppose we have a *small* set of numbers, and our task is to efficiently use our CPU resources to figure out if each number is a prime, one number at a time. Possibly we'll have just one large number to test. It no longer makes sense to use one CPU to do the check; we want to coordinate the work across many CPUs.

For this section we'll look at some larger numbers, one with 15 digits and four with 18 digits:

- Small nonprime: 112,272,535,095,295
- Large nonprime 1: 100,109,100,129,100,369
- Large nonprime 2: 100,109,100,129,101,027
- Prime 1: 100,109,100,129,100,151
- Prime 2: 100,109,100,129,162,907

By using a smaller nonprime and some larger nonprimes, we get to verify that our chosen process not only is faster at checking for primes but also is not getting slower at checking nonprimes. We'll assume that we don't know the size or type of numbers that we're being given, so we want the fastest possible result for all our use cases.



If you own the previous edition of the book, you might be surprised to see that these runtimes with CPython 3.7 are *slightly slower* than the CPython 2.7 runtimes in the last edition, which ran on a slower laptop. The code here is one edge case where Python 3.x is currently slower than CPython 2.7. This code depends on integer operations; CPython 2.7 had system integers mixed with “long” integers (which can store arbitrarily sized numbers but at a cost in speed). CPython 3.x uses only “long” integers for all operations. This implementation is optimized but is still slower in some cases compared to the old (and more complicated) implementation.

We never have to worry about which “kind” of integer is being used, and in CPython 3.7 we take a small speed hit as a consequence. This is a microbenchmark that is incredibly unlikely to affect your own code, as CPython 3.x is faster than CPython 2.x in so many other ways. Our recommendation is not to worry about this, unless you depend on integer operations for most of your execution time—and in that case, we’d strongly suggest you look at PyPy, which doesn’t suffer from this slowdown.

Cooperation comes at a cost—the cost of synchronizing data and checking the shared data can be quite high. We’ll work through several approaches here that can be used in different ways for task coordination.

Note that we’re *not* covering the somewhat specialized message passing interface (MPI) here; we’re looking at batteries-included modules and Redis (which is very common). If you want to use MPI, we assume you already know what you’re doing. The [MPI4PY project](#) would be a good place to start. It is an ideal technology if you want to control latency when lots of processes are collaborating, whether you have one or many machines.

For the following runs, each test is performed 20 times, and the minimum time is taken to show the fastest speed that is possible for that method. In these examples we’re using various techniques to share a flag (often as 1 byte). We could use a basic object like a `Lock`, but then we’d be able to share only 1 bit of state. We’re choosing to show you how to share a primitive type so that more expressive state sharing is possible (even though we don’t need a more expressive state for this example).

We must emphasize that sharing state tends to make things *complicated*—you can easily end up in another hair-pulling state. Be careful and try to keep things as simple as they can be. It might be the case that less efficient resource usage is trumped by developer time spent on other challenges.

First we’ll discuss the results and then we’ll work through the code.

Figure 9-16 shows the first approaches to trying to use interprocess communication to test for primality faster. The benchmark is the serial version, which does not use any interprocess communication; each attempt to speed up our code must at least be faster than this.

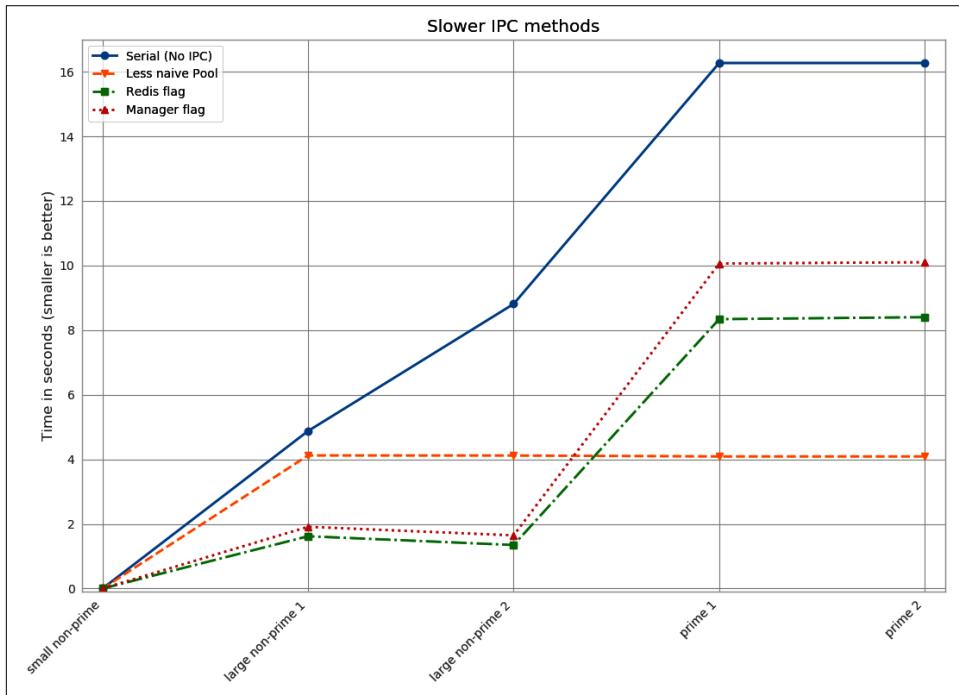


Figure 9-16. The slower ways to use IPC to validate primality

The Less Naive Pool version has a predictable (and good) speed. It is good enough to be rather hard to beat. Don’t overlook the obvious in your search for high-speed solutions—sometimes a dumb and good-enough solution is all you need.

The approach for the Less Naive Pool solution is to take our number under test, divide its possible-factor range evenly among the available CPUs, and then push the work out to each CPU. If any CPU finds a factor, it will exit early, but it won’t communicate this fact; the other CPUs will continue to work through their part of the range. This means for an 18-digit number (our four larger examples), the search time is the same whether it is prime or nonprime.

The Redis and Manager solutions are slower when it comes to testing a larger number of factors for primality because of the communication overhead. They use a shared flag to indicate that a factor has been found and the search should be called off.

Redis lets you share state not just with other Python processes but also with other tools and other machines, and even to expose that state over a web-browser interface (which might be useful for remote monitoring). The `Manager` is a part of `multiprocessing`; it provides a high-level synchronized set of Python objects (including primitives, the `list`, and the `dict`).

For the larger nonprime cases, although there is a cost to checking the shared flag, this is dwarfed by the savings in search time gained by signaling early that a factor has been found.

For the prime cases, though, there is no way to exit early, as no factor will be found, so the cost of checking the shared flag will become the dominating cost.



A little bit of thought is often enough. Here we explore various IPC-based solutions to making the prime-validation task faster. In terms of “minutes of typing” versus “gains made,” the first step—introducing naive parallel processing—gave us the largest win for the smallest effort. Subsequent gains took a lot of extra experimentation. Always think about the ultimate run-time, especially for ad hoc tasks. Sometimes it is just easier to let a loop run all weekend for a one-off task than to optimize the code so it runs quicker.

[Figure 9-17](#) shows that we can get a considerably faster result with a bit of effort. The Less Naive Pool result is still our benchmark, but the `RawValue` and `MMap` (memory map) results are much faster than the previous Redis and `Manager` results. The real magic comes from taking the fastest solution and performing some less-obvious code manipulations to make a near-optimal `MMap` solution—this final version is faster than the Less Naive Pool solution for nonprimes and almost as fast for primes.

In the following sections, we’ll work through various ways of using IPC in Python to solve our cooperative search problem. We hope you’ll see that IPC is fairly easy but generally comes with a cost.

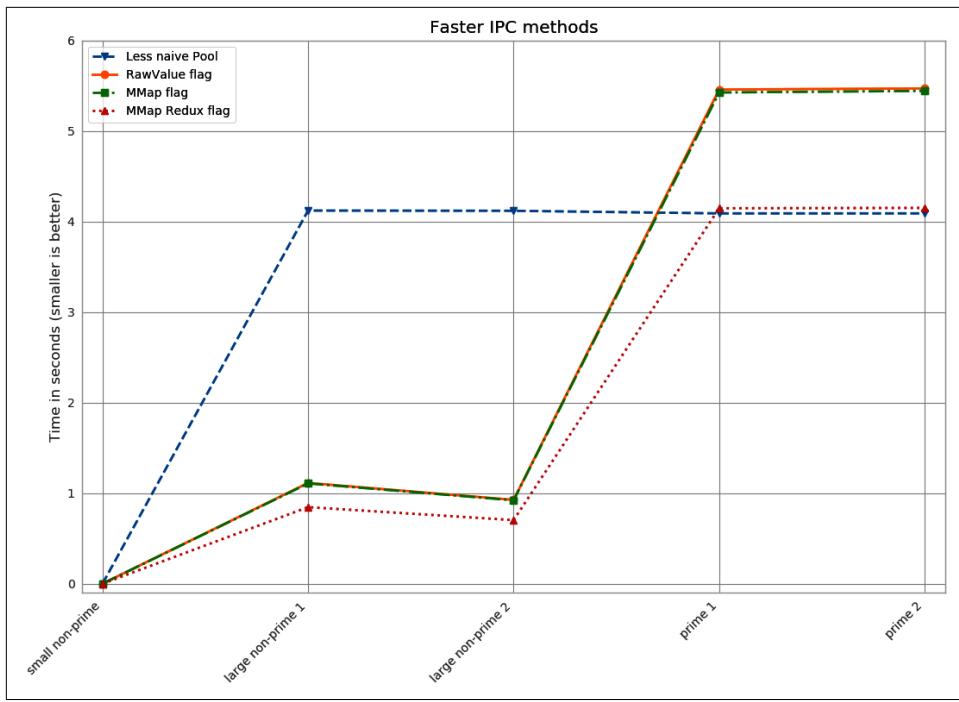


Figure 9-17. The faster ways to use IPC to validate primality

Serial Solution

We'll start with the same serial factor-checking code that we used before, shown again in [Example 9-14](#). As noted earlier, for any nonprime with a large factor, we could more efficiently search the space of factors in parallel. Still, a serial sweep will give us a sensible baseline to work from.

Example 9-14. Serial verification

```
def check_prime(n):
    if n % 2 == 0:
        return False
    from_i = 3
    to_i = math.sqrt(n) + 1
    for i in range(from_i, int(to_i), 2):
        if n % i == 0:
            return False
    return True
```

Naive Pool Solution

The Naive Pool solution works with a `multiprocessing.Pool`, similar to what we saw in “[Finding Prime Numbers](#)” on page 267 and “[Estimating Pi Using Processes and Threads](#)” on page 251 with four forked processes. We have a number to test for primality, and we divide the range of possible factors into four tuples of subranges and send these into the Pool.

In [Example 9-15](#), we use a new method, `create_range.create` (which we won’t show—it’s quite boring), that splits the work space into equal-sized regions. Each item in `ranges_to_check` is a pair of lower and upper bounds to search between. For the first 18-digit nonprime (100,109,100,129,100,369), with four processes we’ll have the factor ranges `ranges_to_check == [(3, 79_100_057), (79_100_057, 158_200_111), (158_200_111, 237_300_165), (237_300_165, 316_400_222)]` (where 316,400,222 is the square root of 100,109,100,129,100,369 plus 1). In `__main__` we first establish a Pool; `check_prime` then splits the `ranges_to_check` for each possibly prime number `n` via a `map`. If the result is `False`, we have found a factor and we do not have a prime.

Example 9-15. Naive Pool solution

```
def check_prime(n, pool, nbr_processes):
    from_i = 3
    to_i = int(math.sqrt(n)) + 1
    ranges_to_check = create_range.create(from_i, to_i, nbr_processes)
    ranges_to_check = zip(len(ranges_to_check) * [n], ranges_to_check)
    assert len(ranges_to_check) == nbr_processes
    results = pool.map(check_prime_in_range, ranges_to_check)
    if False in results:
        return False
    return True

if __name__ == "__main__":
    NBR_PROCESSES = 4
    pool = Pool(processes=NBR_PROCESSES)
    ...
```

We modify the previous `check_prime` in [Example 9-16](#) to take a lower and upper bound for the range to check. There’s no value in passing a complete list of possible factors to check, so we save time and memory by passing just two numbers that define our range.

Example 9-16. check_prime_in_range

```
def check_prime_in_range(n_from_i_to_i):
    (n, (from_i, to_i)) = n_from_i_to_i
```

```

if n % 2 == 0:
    return False
assert from_i % 2 != 0
for i in range(from_i, int(to_i), 2):
    if n % i == 0:
        return False
return True

```

For the “small nonprime” case, the verification time via the Pool is 0.1 seconds, a significantly longer time than the original 0.000002 seconds in the Serial solution. Despite this one worse result, the overall result is a speedup across the board. We could perhaps accept that one slower result isn’t a problem—but what if we might get lots of smaller nonprimes to check? It turns out we can avoid this slowdown; we’ll see that next with the Less Naive Pool solution.

A Less Naive Pool Solution

The previous solution was inefficient at validating the smaller nonprime. For any smaller (fewer than 18 digits) nonprime, it is likely to be slower than the serial method, because of the overhead of sending out partitioned work and not knowing if a very small factor (which is a more likely factor) will be found. If a small factor is found, the process will still have to wait for the other larger factor searches to complete.

We could start to signal between the processes that a small factor has been found, but since this happens so frequently, it will add a lot of communication overhead. The solution presented in [Example 9-17](#) is a more pragmatic approach—a serial check is performed quickly for likely small factors, and if none are found, then a parallel search is started. Combining a serial precheck before launching a relatively more expensive parallel operation is a common approach to avoiding some of the costs of parallel computing.

Example 9-17. Improving the Naive Pool solution for the small-nonprime case

```

def check_prime(n, pool, nbr_processes):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = 21
    if not check_prime_in_range((n, (from_i, to_i))):
        return False

    # continue to check for larger factors in parallel
    from_i = to_i
    to_i = int(math.sqrt(n)) + 1
    ranges_to_check = create_range.create(from_i, to_i, nbr_processes)
    ranges_to_check = zip(len(ranges_to_check) * [n], ranges_to_check)
    assert len(ranges_to_check) == nbr_processes

```

```
results = pool.map(check_prime_in_range, ranges_to_check)
if False in results:
    return False
return True
```

The speed of this solution is equal to or better than that of the original serial search for each of our test numbers. This is our new benchmark.

Importantly, this `Pool` approach gives us an optimal case for the prime-checking situation. If we have a prime, there's no way to exit early; we have to manually check all possible factors before we can exit.

There's no faster way to check though these factors: any approach that adds complexity will have more instructions, so the check-all-factors case will cause the most instructions to be executed. See the various `mmap` solutions covered in “[Using mmap as a Flag](#)” on page 291 for a discussion on how to get as close to this current result for primes as possible.

Using Manager.Value as a Flag

The `multiprocessing.Manager()` lets us share higher-level Python objects between processes as managed shared objects; the lower-level objects are wrapped in proxy objects. The wrapping and safety have a speed cost but also offer great flexibility. You can share both lower-level objects (e.g., integers and floats) and lists and dictionaries.

In [Example 9-18](#), we create a `Manager` and then create a 1-byte (character) `manager.Value(b"c", FLAG_CLEAR)` flag. You could create any of the `ctypes` primitives (which are the same as the `array.array` primitives) if you wanted to share strings or numbers.

Note that `FLAG_CLEAR` and `FLAG_SET` are assigned a byte (`b'0'` and `b'1'`, respectively). We chose to use the leading `b` to be very explicit (it might default to a Unicode or string object if left as an implicit string, depending on your environment and Python version).

Now we can flag across all of our processes that a factor has been found, so the search can be called off early. The difficulty is balancing the cost of reading the flag against the speed savings that is possible. Because the flag is synchronized, we don't want to check it too frequently—this adds more overhead.

Example 9-18. Passing a `Manager.Value` object as a flag

```
SERIAL_CHECK_CUTOFF = 21
CHECK_EVERY = 1000
FLAG_CLEAR = b'0'
FLAG_SET = b'1'
print("CHECK_EVERY", CHECK_EVERY)
```

```

if __name__ == "__main__":
    NBR_PROCESSES = 4
    manager = multiprocessing.Manager()
    value = manager.Value(b'c', FLAG_CLEAR) # 1-byte character
    ...

```

`check_prime_in_range` will now be aware of the shared flag, and the routine will be checking to see if a prime has been spotted by another process. Even though we've yet to begin the parallel search, we must clear the flag as shown in [Example 9-19](#) before we start the serial check. Having completed the serial check, if we haven't found a factor, we know that the flag must still be false.

Example 9-19. Clearing the flag with a Manager.Value

```

def check_prime(n, pool, nbr_processes, value):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = SERIAL_CHECK_CUTOFF
    value.value = FLAG_CLEAR
    if not check_prime_in_range((n, (from_i, to_i), value)):
        return False

    from_i = to_i
    ...

```

How frequently should we check the shared flag? Each check has a cost, both because we're adding more instructions to our tight inner loop and because checking requires a lock to be made on the shared variable, which adds more cost. The solution we've chosen is to check the flag every one thousand iterations. Every time we check, we look to see if `value.value` has been set to `FLAG_SET`, and if so, we exit the search. If in the search the process finds a factor, then it sets `value.value = FLAG_SET` and exits (see [Example 9-20](#)).

Example 9-20. Passing a Manager.Value object as a flag

```

def check_prime_in_range(n_from_i_to_i):
    (n, (from_i, to_i), value) = n_from_i_to_i
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_every = CHECK_EVERY
    for i in range(from_i, int(to_i), 2):
        check_every -= 1
        if not check_every:
            if value.value == FLAG_SET:
                return False
            check_every = CHECK_EVERY

```

```
if n % i == 0:  
    value.value = FLAG_SET  
    return False  
return True
```

The thousand-iteration check in this code is performed using a `check_every` local counter. It turns out that this approach, although readable, is suboptimal for speed. By the end of this section, we'll replace it with a less readable but significantly faster approach.

You might be curious about the total number of times we check for the shared flag. In the case of the two large primes, with four processes we check for the flag 316,405 times (we check it this many times in all of the following examples). Since each check has an overhead due to locking, this cost really adds up.

Using Redis as a Flag

Redis is a key/value in-memory storage engine. It provides its own locking and each operation is atomic, so we don't have to worry about using locks from inside Python (or from any other interfacing language).

By using Redis, we make the data storage language-agnostic—any language or tool with an interface to Redis can share data in a compatible way. You could share data between Python, Ruby, C++, and PHP equally easily. You can share data on the local machine or over a network; to share to other machines, all you need to do is change the Redis default of sharing only on `localhost`.

Redis lets you store the following:

- Lists of strings
- Sets of strings
- Sorted sets of strings
- Hashes of strings

Redis stores everything in RAM and snapshots to disk (optionally using journaling) and supports master/slave replication to a cluster of instances. One possibility with Redis is to use it to share a workload across a cluster, where other machines read and write state and Redis acts as a fast centralized data repository.

We can read and write a flag as a text string (all values in Redis are strings) in just the same way as we have been using Python flags previously. We create a `StrictRedis` interface as a global object, which talks to the external Redis server. We could create a new connection inside `check_prime_in_range`, but this is slower and can exhaust the limited number of Redis handles that are available.

We talk to the Redis server using a dictionary-like access. We can set a value using `rds[SOME_KEY] = SOME_VALUE` and read the string back using `rds[SOME_KEY]`.

Example 9-21 is very similar to the previous Manager example—we’re using Redis as a substitute for the local Manager. It comes with a similar access cost. You should note that Redis supports other (more complex) data structures; it is a powerful storage engine that we’re using just to share a flag for this example. We encourage you to familiarize yourself with its features.

Example 9-21. Using an external Redis server for our flag

```
FLAG_NAME = b'redis_primes_flag'
FLAG_CLEAR = b'0'
FLAG_SET = b'1'

rds = redis.StrictRedis()

def check_prime_in_range(n_from_i_to_i):
    (n, (from_i, to_i)) = n_from_i_to_i
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_every = CHECK_EVERY
    for i in range(from_i, int(to_i), 2):
        check_every -= 1
        if not check_every:
            flag = rds[FLAG_NAME]
            if flag == FLAG_SET:
                return False
            check_every = CHECK_EVERY

        if n % i == 0:
            rds[FLAG_NAME] = FLAG_SET
            return False
    return True

def check_prime(n, pool, nbr_processes):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = SERIAL_CHECK_CUTOFF
    rds[FLAG_NAME] = FLAG_CLEAR
    if not check_prime_in_range((n, (from_i, to_i))):
        return False

    ...
    if False in results:
        return False
    return True
```

To confirm that the data is stored outside these Python instances, we can invoke `redis-cli` at the command line, as in [Example 9-22](#), and get the value stored in the key `redis_primes_flag`. You'll note that the returned item is a string (not an integer). All values returned from Redis are strings, so if you want to manipulate them in Python, you'll have to convert them to an appropriate datatype first.

Example 9-22. redis-cli

```
$ redis-cli  
redis 127.0.0.1:6379> GET "redis_primes_flag"  
"0"
```

One powerful argument in favor of the use of Redis for data sharing is that it lives outside the Python world—non-Python developers on your team will understand it, and many tools exist for it. They'll be able to look at its state while reading (but not necessarily running and debugging) your code and follow what's happening. From a team-velocity perspective, this might be a big win for you, despite the communication overhead of using Redis. While Redis is an additional dependency on your project, you should note that it is a very commonly deployed tool, and one that is well debugged and well understood. Consider it a powerful tool to add to your armory.

Redis has many configuration options. By default it uses a TCP interface (that's what we're using), although the benchmark documentation notes that sockets might be much faster. It also states that while TCP/IP lets you share data over a network between different types of OS, other configuration options are likely to be faster (but also are likely to limit your communication options):

When the server and client benchmark programs run on the same box, both the TCP/IP loopback and unix domain sockets can be used. It depends on the platform, but unix domain sockets can achieve around 50% more throughput than the TCP/IP loopback (on Linux for instance). The default behavior of `redis-benchmark` is to use the TCP/IP loopback. The performance benefit of unix domain sockets compared to TCP/IP loopback tends to decrease when pipelining is heavily used (i.e., long pipelines).

—[Redis documentation](#)

Redis is widely used in industry and is mature and well trusted. If you're not familiar with the tool, we strongly suggest you take a look at it; it has a place in your high performance toolkit.

Using RawValue as a Flag

`multiprocessing.RawValue` is a thin wrapper around a `ctypes` block of bytes. It lacks synchronization primitives, so there's little to get in our way in our search for

the fastest way to set a flag between processes. It will be almost as fast as the following `mmap` example (it is slower only because a few more instructions get in the way).

Again, we could use any `ctypes` primitive; there's also a `RawArray` option for sharing an array of primitive objects (which will behave similarly to `array.array`). `RawValue` avoids any locking—it is faster to use, but you don't get atomic operations.

Generally, if you avoid the synchronization that Python provides during IPC, you'll come unstuck (once again, back to that pulling-your-hair-out situation). *However*, in this problem it doesn't matter if one or more processes set the flag at the same time—the flag gets switched in only one direction, and every other time it is read, it is just to learn if the search can be called off.

Because we never reset the state of the flag during the parallel search, we don't need synchronization. Be aware that this may not apply to your problem. If you avoid synchronization, please make sure you are doing it for the right reasons.

If you want to do things like update a shared counter, look at the documentation for the `Value` and use a context manager with `value.get_lock()`, as the implicit locking on a `Value` doesn't allow for atomic operations.

This example looks very similar to the previous `Manager` example. The only difference is that in [Example 9-23](#) we create the `RawValue` as a one-character (byte) flag.

Example 9-23. Creating and passing a RawValue

```
if __name__ == "__main__":
    NBR_PROCESSES = 4
    value = multiprocessing.RawValue('b', FLAG_CLEAR) # 1-byte character
    pool = Pool(processes=NBR_PROCESSES)
    ...
```

The flexibility to use managed and raw values is a benefit of the clean design for data sharing in `multiprocessing`.

Using `mmap` as a Flag

Finally, we get to the fastest way of sharing bytes. [Example 9-24](#) shows a memory-mapped (shared memory) solution using the `mmap` module. The bytes in a shared memory block are not synchronized, and they come with very little overhead. They act like a file—in this case, they are a block of memory with a file-like interface. We have to seek to a location and read or write sequentially. Typically, `mmap` is used to give a short (memory-mapped) view into a larger file, but in our case, rather than specifying a file number as the first argument, we instead pass `-1` to indicate that we want an anonymous block of memory. We could also specify whether we want read-only or write-only access (we want both, which is the default).

Example 9-24. Using a shared memory flag via mmap

```
sh_mem = mmap.mmap(-1, 1) # memory map 1 byte as a flag

def check_prime_in_range(n_from_i_to_i):
    (n, (from_i, to_i)) = n_from_i_to_i
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_every = CHECK_EVERY
    for i in range(from_i, int(to_i), 2):
        check_every -= 1
        if not check_every:
            sh_mem.seek(0)
            flag = sh_mem.read_byte()
            if flag == FLAG_SET:
                return False
            check_every = CHECK_EVERY

    if n % i == 0:
        sh_mem.seek(0)
        sh_mem.write_byte(FLAG_SET)
        return False
    return True

def check_prime(n, pool, nbr_processes):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = SERIAL_CHECK_CUTOFF
    sh_mem.seek(0)
    sh_mem.write_byte(FLAG_CLEAR)
    if not check_prime_in_range((n, (from_i, to_i))):
        return False

    ...
    if False in results:
        return False
    return True
```

`mmap` supports a number of methods that can be used to move around in the file that it represents (including `find`, `readline`, and `write`). We are using it in the most basic way—we `seek` to the start of the memory block before each read or write, and since we’re sharing just 1 byte, we use `read_byte` and `write_byte` to be explicit.

There is no Python overhead for locking and no interpretation of the data; we’re dealing with bytes directly with the operating system, so this is our fastest communication method.

Using mmap as a Flag Redux

While the previous `mmap` result was the best overall, we couldn't help but think that we should be able to get back to the Naive Pool result for the most expensive case of having primes. The goal is to accept that there is no early exit from the inner loop and to minimize the cost of anything extraneous.

This section presents a slightly more complex solution. The same changes can be made to the other flag-based approaches we've seen, although this `mmap` result will still be fastest.

In our previous examples, we've used `CHECK_EVERY`. This means we have the `check_next` local variable to track, decrement, and use in Boolean tests—and each operation adds a bit of extra time to every iteration. In the case of validating a large prime, this extra management overhead occurs over 300,000 times.

The first optimization, shown in [Example 9-25](#), is to realize that we can replace the decremented counter with a look-ahead value, and then we only have to do a Boolean comparison on the inner loop. This removes a decrement, which, because of Python's interpreted style, is quite slow. This optimization works in this test in CPython 3.7, but it is unlikely to offer any benefit in a smarter compiler (e.g., PyPy or Cython). This step saved 0.1 seconds when checking one of our large primes.

Example 9-25. Starting to optimize away our expensive logic

```
def check_prime_in_range(n_from_i_to_i):
    (n, (from_i, to_i)) = n_from_i_to_i
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_next = from_i + CHECK_EVERY
    for i in range(from_i, int(to_i), 2):
        if check_next == i:
            sh_mem.seek(0)
            flag = sh_mem.read_byte()
            if flag == FLAG_SET:
                return False
            check_next += CHECK_EVERY

        if n % i == 0:
            sh_mem.seek(0)
            sh_mem.write_byte(FLAG_SET)
            return False
    return True
```

We can also entirely replace the logic that the counter represents, as shown in [Example 9-26](#), by unrolling our loop into a two-stage process. First, the outer loop covers the expected range, but in steps, on `CHECK_EVERY`. Second, a new inner loop

replaces the `check_every` logic—it checks the local range of factors and then finishes. This is equivalent to the `if not check_every:` test. We follow this with the previous `sh_mem` logic to check the early-exit flag.

Example 9-26. Optimizing away our expensive logic

```
def check_prime_in_range(n_from_i_to_i):
    (n, (from_i, to_i)) = n_from_i_to_i
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    for outer_counter in range(from_i, int(to_i), CHECK_EVERY):
        upper_bound = min(int(to_i), outer_counter + CHECK_EVERY)
        for i in range(outer_counter, upper_bound, 2):
            if n % i == 0:
                sh_mem.seek(0)
                sh_mem.write_byte(FLAGS_SET)
                return False
        sh_mem.seek(0)
        flag = sh_mem.read_byte()
        if flag == FLAGS_SET:
            return False
    return True
```

The speed impact is dramatic. Our nonprime case improves even further, but more importantly, our prime-checking case is nearly as fast as the Less Naive Pool version (it is now just 0.1 seconds slower). Given that we’re doing a lot of extra work with interprocess communication, this is an interesting result. Do note, though, that it is specific to CPython and unlikely to offer any gains when run through a compiler.

In the last edition of the book we went even further with a final example that used loop unrolling and local references to global objects and eked out a further performance gain at the expense of readability. This example in Python 3 yields a minor slowdown, so we’ve removed it. We’re happy about this—fewer hoops needed jumping through to get the most performant example, and the preceding code is more likely to be supported correctly in a team than one that makes implementation-specific code changes.



These examples work just fine with PyPy, where they run around seven times faster than in CPython. Sometimes the better solution will be to investigate other runtimes rather than to go down rabbit holes with CPython.

Sharing numpy Data with multiprocessing

When working with large numpy arrays, you’re bound to wonder if you can share the data for read and write access, without a copy, between processes. It is possible, though a little fiddly. We’d like to acknowledge Stack Overflow user *pv* for the inspiration for this demo.²



Do not use this method to re-create the behaviors of BLAS, MKL, Accelerate, and ATLAS. These libraries all have multithreading support in their primitives, and they likely are better-debugged than any new routine that you create. They can require some configuration to enable multithreading support, but it would be wise to see if these libraries can give you free speedups before you invest time (and lose time to debugging!) writing your own.

Sharing a large matrix between processes has several benefits:

- Only one copy means no wasted RAM.
- No time is wasted copying large blocks of RAM.
- You gain the possibility of sharing partial results between the processes.

Thinking back to the pi estimation demo using numpy in “[Using numpy](#)” on page 264, we had the problem that the random number generation was a serial process. Here, we can imagine forking processes that share one large array, each one using a differently seeded random number generator to fill in a section of the array with random numbers, and therefore completing the generation of a large random block faster than is possible with a single process.

To verify this, we modified the forthcoming demo to create a large random matrix ($10,000 \times 320,000$ elements) as a serial process and by splitting the matrix into four segments where `random` is called in parallel (in both cases, one row at a time). The serial process took 53 seconds, and the parallel version took 29 seconds. Refer back to “[Random Numbers in Parallel Systems](#)” on page 263 to understand some of the dangers of parallelized random number generation.

For the rest of this section, we’ll use a simplified demo that illustrates the point while remaining easy to verify.

In [Figure 9-18](#), you can see the output from `htop` on Ian’s laptop. It shows four child processes of the parent (with PID 27628), where all five processes are sharing a single 10,000-by-320,000-element numpy array of doubles. One copy of this array costs

² See the [Stack Overflow topic](#).

25.6 GB, and the laptop has only 32 GB—you can see in `htop` by the process meters that the `Mem` reading shows a maximum of 31.1 GB RAM.

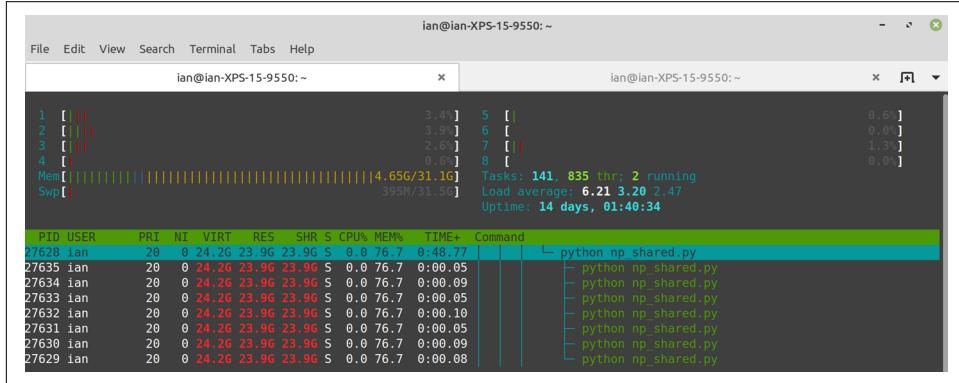


Figure 9-18. `htop` showing RAM and swap usage

To understand this demo, we'll first walk through the console output, and then we'll look at the code. In [Example 9-27](#), we start the parent process: it allocates a 25.6 GB double array of dimensions $10,000 \times 320,000$, filled with the value zero. The 10,000 rows will be passed out as indices to the worker function, and the worker will operate on each column of 320,000 items in turn. Having allocated the array, we fill it with the answer to life, the universe, and everything (42!). We can test in the worker function that we're receiving this modified array and not a filled-with-0s version to confirm that this code is behaving as expected.

Example 9-27. Setting up the shared array

```
$ python np_shared.py
Created shared array with 25,600,000,000 nbytes
Shared array id is 139636238840896 in PID 27628
Starting with an array of 0 values:
[[ 0.  0.  0. ...,  0.  0.  0.]
 ...
 [ 0.  0.  0. ...,  0.  0.  0.]]

Original array filled with value 42:
[[ 42.  42.  42. ...,  42.  42.  42.]
 ...
 [ 42.  42.  42. ...,  42.  42.  42.]]
Press a key to start workers using multiprocessing...
```

In [Example 9-28](#), we've started four processes working on this shared array. No copy of the array was made; each process is looking at the same large block of memory, and each process has a different set of indices to work from. Every few thousand lines, the worker outputs the current index and its PID, so we can observe its behav-

ior. The worker's job is trivial—it will check that the current element is still set to the default (so we know that no other process has modified it already), and then it will overwrite this value with the current PID. Once the workers have completed, we return to the parent process and print the array again. This time, we see that it is filled with PIDs rather than 42.

Example 9-28. Running `worker_fn` on the shared array

```
worker_fn: with idx 0
  id of local_npyarray_in_process is 139636238840896 in PID 27751
worker_fn: with idx 2000
  id of local_npyarray_in_process is 139636238840896 in PID 27754
worker_fn: with idx 1000
  id of local_npyarray_in_process is 139636238840896 in PID 27752
worker_fn: with idx 4000
  id of local_npyarray_in_process is 139636238840896 in PID 27753
...
worker_fn: with idx 8000
  id of local_npyarray_in_process is 139636238840896 in PID 27752
```

The default value has been overwritten with `worker_fn`'s result:
[[27751. 27751. 27751. ... 27751. 27751. 27751.]

...
[27751. 27751. 27751. ... 27751. 27751. 27751.]]

Finally, in [Example 9-29](#) we use a Counter to confirm the frequency of each PID in the array. As the work was evenly divided, we expect to see each of the four PIDs represented an equal number of times. In our 3,200,000,000-element array, we see four sets of 800,000,000 PIDs. The table output is presented using [PrettyTable](#).

Example 9-29. Verifying the result on the shared array

```
Verification - extracting unique values from 3,200,000,000 items
in the numpy array (this might be slow)...
Unique values in main_npyarray:
+-----+-----+
|   PID   |   Count   |
+-----+-----+
| 27751.0 | 800000000 |
| 27752.0 | 800000000 |
| 27753.0 | 800000000 |
| 27754.0 | 800000000 |
+-----+-----+
Press a key to exit...
```

Having completed, the program now exits, and the array is deleted.

We can take a peek inside each process under Linux by using `ps` and `pmap`. [Example 9-30](#) shows the result of calling `ps`. Breaking apart this command line:

- `ps` tells us about the process.
- `-A` lists all processes.
- `-o pid,size,vsize,cmd` outputs the PID, size information, and the command name.
- `grep` is used to filter all other results and leave only the lines for our demo.

The parent process (PID 27628) and its four forked children are shown in the output. The result is similar to what we saw in `htop`. We can use `pmap` to look at the memory map of each process, requesting extended output with `-x`. We `grep` for the pattern `s-` to list blocks of memory that are marked as being shared. In the parent process and the child processes, we see a 25,000,000 KB (25.6 GB) block that is shared between them.

Example 9-30. Using pmap and ps to investigate the operating system's view of the processes

```
$ ps -A -o pid,size,vsize,cmd | grep np_shared
27628 279676 25539428 python np_shared.py
27751 279148 25342688 python np_shared.py
27752 279148 25342688 python np_shared.py
27753 279148 25342688 python np_shared.py
27754 279148 25342688 python np_shared.py

ian@ian-Latitude-E6420 $ pmap -x 27628 | grep s-
Address          Kbytes      RSS   Dirty Mode  Mapping
00007ef9a2853000 25000000 25000000 2584636  rw-s-  pym-27628-npfjsxl6 (deleted)
...
ian@ian-Latitude-E6420 $ pmap -x 27751 | grep s-
Address          Kbytes      RSS   Dirty Mode  Mapping
00007ef9a2853000 25000000 6250104 1562508  rw-s-  pym-27628-npfjsxl6 (deleted)
...
```

We'll use a `multiprocessing.Array` to allocate a shared block of memory as a 1D array and then instantiate a `numpy` array from this object and reshape it to a 2D array. Now we have a `numpy`-wrapped block of memory that can be shared between processes and addressed as though it were a normal `numpy` array. `numpy` is not managing the RAM; `multiprocessing.Array` is managing it.

In [Example 9-31](#), you can see that each forked process has access to a global `main_nparray`. While the forked process has a copy of the `numpy` object, the underlying bytes that the object accesses are stored as shared memory. Our `worker_fn` will overwrite a chosen row (via `idx`) with the current process identifier.

Example 9-31. worker_fn for sharing numpy arrays using multiprocessing

```
import os
import multiprocessing
from collections import Counter
import ctypes
import numpy as np
from prettytable import PrettyTable

SIZE_A, SIZE_B = 10_000, 320_000 # 24GB

def worker_fn(idx):
    """Do some work on the shared np array on row idx"""
    # confirm that no other process has modified this value already
    assert main_nparray[idx, 0] == DEFAULT_VALUE
    # inside the subprocess print the PID and ID of the array
    # to check we don't have a copy
    if idx % 1000 == 0:
        print(" {}: with idx {}\\n id of local_nparray_in_process is {} in PID {}"\ \
              .format(worker_fn.__name__, idx, id(main_nparray), os.getpid()))
    # we can do any work on the array; here we set every item in this row to
    # have the value of the process ID for this process
    main_nparray[idx, :] = os.getpid()
```

In our `__main__` in Example 9-32, we'll work through three major stages:

1. Build a shared `multiprocessing.Array` and convert it into a `numpy` array.
2. Set a default value into the array, and spawn four processes to work on the array in parallel.
3. Verify the array's contents after the processes return.

Typically, you'd set up a `numpy` array and work on it in a single process, probably doing something like `arr = np.array((100, 5), dtype=np.float_)`. This is fine in a single process, but you can't share this data across processes for both reading and writing.

The trick is to make a shared block of bytes. One way is to create a `multiprocessing.Array`. By default the `Array` is wrapped in a lock to prevent concurrent edits, but we don't need this lock as we'll be careful about our access patterns. To communicate this clearly to other team members, it is worth being explicit and setting `lock=False`.

If you don't set `lock=False`, you'll have an object rather than a reference to the bytes, and you'll need to call `.get_obj()` to get to the bytes. By calling `.get_obj()`, you bypass the lock, so there's no value in not being explicit about this in the first place.

Next, we take this block of shareable bytes and wrap a `numpy` array around them using `frombuffer`. The `dtype` is optional, but since we're passing bytes around, it is always

sensible to be explicit. We reshape so we can address the bytes as a 2D array. By default the array values are set to 0. [Example 9-32](#) shows our `__main__` in full.

Example 9-32. `__main__` to set up numpy arrays for sharing

```
if __name__ == '__main__':
    DEFAULT_VALUE = 42
    NBR_OF_PROCESSES = 4

    # create a block of bytes, reshape into a local numpy array
    NBR_ITEMS_IN_ARRAY = SIZE_A * SIZE_B
    shared_array_base = multiprocessing.Array(ctypes.c_double,
                                              NBR_ITEMS_IN_ARRAY, lock=False)
    main_npararray = np.frombuffer(shared_array_base, dtype=ctypes.c_double)
    main_npararray = main_npararray.reshape(SIZE_A, SIZE_B)
    # assert no copy was made
    assert main_npararray.base.base is shared_array_base
    print("Created shared array with {}: {} nbytes".format(main_npararray.nbytes))
    print("Shared array id is {} in PID {}".format(id(main_npararray), os.getpid()))
    print("Starting with an array of 0 values:")
    print(main_npararray)
    print()
```

To confirm that our processes are operating on the same block of data that we started with, we set each item to a new `DEFAULT_VALUE` (we again use 42, the answer to life, the universe, and everything)—you’ll see that at the top of [Example 9-33](#). Next, we build a Pool of processes (four in this case) and then send batches of row indices via the call to `map`.

Example 9-33. `__main__` for sharing numpy arrays using `multiprocessing`

```
# Modify the data via our local numpy array
main_npararray.fill(DEFAULT_VALUE)
print("Original array filled with value {}".format(DEFAULT_VALUE))
print(main_npararray)

input("Press a key to start workers using multiprocessing...")
print()

# create a pool of processes that will share the memory block
# of the global numpy array, share the reference to the underlying
# block of data so we can build a numpy array wrapper in the new processes
pool = multiprocessing.Pool(processes=NBR_OF_PROCESSES)
# perform a map where each row index is passed as a parameter to the
# worker_fn
pool.map(worker_fn, range(SIZE_A))
```

Once we've completed the parallel processing, we return to the parent process to verify the result ([Example 9-34](#)). The verification step runs through a flattened view on the array (note that the view does *not* make a copy; it just creates a 1D iterable view on the 2D array), counting the frequency of each PID. Finally, we perform some `assert` checks to make sure we have the expected counts.

Example 9-34. `__main__` to verify the shared result

```
print("Verification - extracting unique values from {:,} items\n in the numpy \
      array (this might be slow)...".format(NBR_ITEMS_IN_ARRAY))
# main_nparray.flat iterates over the contents of the array, it doesn't
# make a copy
counter = Counter(main_nparray.flat)
print("Unique values in main_nparray:")
tbl = PrettyTable(["PID", "Count"])
for pid, count in list(counter.items()):
    tbl.add_row([pid, count])
print(tbl)

total_items_set_in_array = sum(counter.values())

# check that we have set every item in the array away from DEFAULT_VALUE
assert DEFAULT_VALUE not in list(counter.keys())
# check that we have accounted for every item in the array
assert total_items_set_in_array == NBR_ITEMS_IN_ARRAY
# check that we have NBR_OF_PROCESSES of unique keys to confirm that every
# process did some of the work
assert len(counter) == NBR_OF_PROCESSES

input("Press a key to exit...")
```

We've just created a 1D array of bytes, converted it into a 2D array, shared the array among four processes, and allowed them to process concurrently on the same block of memory. This recipe will help you parallelize over many cores. Be careful with concurrent access to the *same* data points, though—you'll have to use the locks in `multiprocessing` if you want to avoid synchronization problems, and this will slow down your code.

Synchronizing File and Variable Access

In the following examples, we'll look at multiple processes sharing and manipulating a state—in this case, four processes incrementing a shared counter a set number of times. Without a synchronization process, the counting is incorrect. If you're sharing data in a coherent way you'll always need a method to synchronize the reading and writing of data, or you'll end up with errors.

Typically, the synchronization methods are specific to the OS you're using, and they're often specific to the language you use. Here, we look at file-based synchronization using a Python library and sharing an integer object between Python processes.

File Locking

Reading and writing to a file will be the slowest example of data sharing in this section.

You can see our first work function in [Example 9-35](#). The function iterates over a local counter. In each iteration it opens a file and reads the existing value, increments it by one, and then writes the new value over the old one. On the first iteration the file will be empty or won't exist, so it will catch an exception and assume the value should be zero.



The examples given here are simplified—in practice it is safer to use a context manager to open a file using `with open(filename, "r") as f:`. If an exception is raised inside the context, the file `f` will correctly be closed.

Example 9-35. work function without a lock

```
def work(filename, max_count):
    for n in range(max_count):
        f = open(filename, "r")
        try:
            nbr = int(f.read())
        except ValueError as err:
            print("File is empty, starting to count from 0, error: " + str(err))
            nbr = 0
        f = open(filename, "w")
        f.write(str(nbr + 1) + '\n')
        f.close()
```

Let's run this example with one process. You can see the output in [Example 9-36](#). `work` is called one thousand times, and as expected it counts correctly without losing any data. On the first read, it sees an empty file. This raises the `invalid literal for int()` error for `int()` (as `int()` is called on an empty string). This error occurs only once; afterward, we always have a valid value to read and convert into an integer.

Example 9-36. Timing of file-based counting without a lock and with one process

```
$ python ex1_nolock1.py
Starting 1 process(es) to count to 1000
File is empty, starting to count from 0,
```

```
error: invalid literal for int() with base 10: ''
Expecting to see a count of 1000
count.txt contains:
1000
```

Now we'll run the same `work` function with four concurrent processes. We don't have any locking code, so we'll expect some odd results.



Before you look at the following code, what *two* types of error can you expect to see when two processes simultaneously read from or write to the same file? Think about the two main states of the code (the start of execution for each process and the normal running state of each process).

Take a look at [Example 9-37](#) to see the problems. First, when each process starts, the file is empty, so each tries to start counting from zero. Second, as one process writes, the other can read a partially written result that can't be parsed. This causes an exception, and a zero will be written back. This, in turn, causes our counter to keep getting reset! Can you see how \n and two values have been written by two concurrent processes to the same open file, causing an invalid entry to be read by a third process?

Example 9-37. Timing of file-based counting without a lock and with four processes

```
$ python ex1_nolock4.py
Starting 4 process(es) to count to 4000
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: ''
# many errors like these
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: ''
Expecting to see a count of 4000
count.txt contains:
112

$ python -m timeit -s "import ex1_nolock4" "ex1_nolock4.run_workers()"
2 loops, best of 5: 158 msec per loop
```

[Example 9-38](#) shows the `multiprocessing` code that calls `work` with four processes. Note that rather than using a `map`, we're building a list of `Process` objects. Although we don't use the functionality here, the `Process` object gives us the power to introspect the state of each `Process`. We encourage you to [read the documentation](#) to learn about why you might want to use a `Process`.

Example 9-38. run_workers setting up four processes

```
import multiprocessing
import os

...
MAX_COUNT_PER_PROCESS = 1000
FILENAME = "count.txt"
...

def run_workers():
    NBR_PROCESSES = 4
    total_expected_count = NBR_PROCESSES * MAX_COUNT_PER_PROCESS
    print("Starting {} process(es) to count to {}".format(NBR_PROCESSES,
                                                          total_expected_count))
    # reset counter
    f = open(FILENAME, "w")
    f.close()

    processes = []
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work, args=(FILENAME,
                                                       MAX_COUNT_PER_PROCESS))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    print("Expecting to see a count of {}".format(total_expected_count))
    print("{} contains:{}".format(FILENAME))
    os.system('more ' + FILENAME)

if __name__ == "__main__":
    run_workers()
```

Using the **fasteners module**, we can introduce a synchronization method so only one process gets to write at a time and the others each await their turn. The overall process therefore runs more slowly, but it doesn't make mistakes. You can see the correct output in [Example 9-39](#). Be aware that the locking mechanism is specific to Python, so other processes that are looking at this file will *not* care about the “locked” nature of this file.

Example 9-39. Timing of file-based counting with a lock and four processes

```
$ python ex1_lock.py
Starting 4 process(es) to count to 4000
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: ''
Expecting to see a count of 4000
```

```
count.txt contains:  
4000  
$ python -m timeit -s "import ex1_lock" "ex1_lock.run_workers()"  
10 loops, best of 3: 401 msec per loop
```

Using `fasteners` adds a single line of code in [Example 9-40](#) with the `@fasteners.interprocess_locked` decorator; the filename can be anything, but using a similar name as the file you want to lock probably makes debugging from the command line easier. Note that we haven't had to change the inner function; the decorator gets the lock on each call, and it will wait until it can get the lock before the call into `work` proceeds.

Example 9-40. work function with a lock

```
@fasteners.interprocess_locked('/tmp/tmp_lock')  
def work(filename, max_count):  
    for n in range(max_count):  
        f = open(filename, "r")  
        try:  
            nbr = int(f.read())  
        except ValueError as err:  
            print("File is empty, starting to count from 0, error: " + str(err))  
            nbr = 0  
        f = open(filename, "w")  
        f.write(str(nbr + 1) + '\n')  
        f.close()
```

Locking a Value

The `multiprocessing` module offers several options for sharing Python objects between processes. We can share primitive objects with a low communication overhead, and we can also share higher-level Python objects (e.g., dictionaries and lists) using a `Manager` (but note that the synchronization cost will significantly slow down the data sharing).

Here, we'll use a `multiprocessing.Value` object to share an integer between processes. While a `Value` has a lock, the lock doesn't do quite what you might expect—it prevents simultaneous reads or writes but does *not* provide an atomic increment. [Example 9-41](#) illustrates this. You can see that we end up with an incorrect count; this is similar to the file-based unsynchronized example we looked at earlier.

Example 9-41. No locking leads to an incorrect count

```
$ python ex2_nolock.py  
Expecting to see a count of 4000  
We have counted to 2340
```

```
$ python -m timeit -s "import ex2_nolock" "ex2_nolock.run_workers()"  
20 loops, best of 5: 9.97 msec per loop
```

No corruption occurs to the data, but we do miss some of the updates. This approach might be suitable if you're writing to a `Value` from one process and consuming (but not modifying) that `Value` in other processes.

The code to share the `Value` is shown in [Example 9-42](#). We have to specify a datatype and an initialization value—using `Value("i", 0)`, we request a signed integer with a default value of 0. This is passed as a regular argument to our `Process` object, which takes care of sharing the same block of bytes between processes behind the scenes. To access the primitive object held by our `Value`, we use `.value`. Note that we're asking for an in-place addition—we'd expect this to be an atomic operation, but that's not supported by `Value`, so our final count is lower than expected.

Example 9-42. The counting code without a Lock

```
import multiprocessing  
  
def work(value, max_count):  
    for n in range(max_count):  
        value.value += 1  
  
def run_workers():  
    ...  
    value = multiprocessing.Value('i', 0)  
    for process_nbr in range(NBR_PROCESSES):  
        p = multiprocessing.Process(target=work, args=(value, MAX_COUNT_PER_PROCESS))  
        p.start()  
        processes.append(p)  
    ...
```

You can see the correctly synchronized count in [Example 9-43](#) using a `multiprocessing.Lock`.

Example 9-43. Using a Lock to synchronize writes to a Value

```
# lock on the update, but this isn't atomic  
$ python ex2_lock.py  
Expecting to see a count of 4000  
We have counted to 4000  
$ python -m timeit -s "import ex2_lock" "ex2_lock.run_workers()"  
20 loops, best of 5: 19.3 msec per loop
```

In [Example 9-44](#), we've used a context manager (with `Lock`) to acquire the lock.

Example 9-44. Acquiring a Lock using a context manager

```
import multiprocessing

def work(value, max_count, lock):
    for n in range(max_count):
        with lock:
            value.value += 1

def run_workers():
    ...
    processes = []
    lock = multiprocessing.Lock()
    value = multiprocessing.Value('i', 0)
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work,
                                    args=(value, MAX_COUNT_PER_PROCESS, lock))
        p.start()
        processes.append(p)
    ...

If we avoid the context manager and directly wrap our increment with acquire and release, we can go a little faster, but the code is less readable compared to using the context manager. We suggest sticking to the context manager to improve readability. The snippet in Example 9-45 shows how to acquire and release the Lock object.
```

Example 9-45. Inline locking rather than using a context manager

```
lock.acquire()
value.value += 1
lock.release()
```

Since a Lock doesn't give us the level of granularity that we're after, the basic locking that it provides wastes a bit of time unnecessarily. We can replace the Value with a RawValue, as in Example 9-46, and achieve an incremental speedup. If you're interested in seeing the bytecode behind this change, read [Eli Bendersky's blog post](#) on the subject.

Example 9-46. Console output showing the faster RawValue and Lock approach

```
# RawValue has no lock on it
$ python ex2_lock_rawvalue.py
Expecting to see a count of 4000
We have counted to 4000
$ python -m timeit -s "import ex2_lock_rawvalue" "ex2_lock_rawvalue.run_workers()"
50 loops, best of 5: 9.49 msec per loop
```

To use a `RawValue`, just swap it for a `Value`, as shown in [Example 9-47](#).

Example 9-47. Example of using a <code>RawValue</code> integer

```
...
def run_workers():
    ...
    lock = multiprocessing.Lock()
    value = multiprocessing.RawValue('i', 0)
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work,
                                    args=(value, MAX_COUNT_PER_PROCESS, lock))
        p.start()
        processes.append(p)
```

We could also use a `RawArray` in place of a `multiprocessing.Array` if we were sharing an array of primitive objects.

We've looked at various ways of dividing up work on a single machine between multiple processes, along with sharing a flag and synchronizing data sharing between these processes. Remember, though, that sharing data can lead to headaches—try to avoid it if possible. Making a machine deal with all the edge cases of state sharing is hard; the first time you have to debug the interactions of multiple processes, you'll realize why the accepted wisdom is to avoid this situation if possible.

Do consider writing code that runs a bit slower but is more likely to be understood by your team. Using an external tool like Redis to share state leads to a system that can be inspected at runtime by people *other* than the developers—this is a powerful way to enable your team to keep on top of what's happening in your parallel systems.

Definitely bear in mind that tweaked performant Python code is less likely to be understood by more junior members of your team—they'll either be scared of it or break it. Avoid this problem (and accept a sacrifice in speed) to keep team velocity high.

Wrap-Up

We've covered a lot in this chapter. First, we looked at two embarrassingly parallel problems, one with predictable complexity and the other with nonpredictable complexity. We'll use these examples again on multiple machines when we discuss clustering in [Chapter 10](#).

Next, we looked at Queue support in `multiprocessing` and its overheads. In general, we recommend using an external queue library so that the state of the queue is more transparent. Preferably, you should use an easy-to-read job format so that it is easy to debug, rather than pickled data.

The IPC discussion should have impressed upon you how difficult it is to use IPC efficiently, and that it can make sense just to use a naive parallel solution (without IPC). Buying a faster computer with more cores might be a far more pragmatic solution than trying to use IPC to exploit an existing machine.

Sharing `numpy` matrices in parallel without making copies is important for only a small set of problems, but when it counts, it'll really count. It takes a few extra lines of code and requires some sanity checking to make sure that you're really not copying the data between processes.

Finally, we looked at using file and memory locks to avoid corrupting data—this is a source of subtle and hard-to-track errors, and this section showed you some robust and lightweight solutions.

In the next chapter we'll look at clustering using Python. With a cluster, we can move beyond single-machine parallelism and utilize the CPUs on a group of machines. This introduces a new world of debugging pain—not only can your code have errors, but the other machines can also have errors (either from bad configuration or from failing hardware). We'll show how to parallelize the pi estimation demo using the Parallel Python module and how to run research code inside IPython using an IPython cluster.

Clusters and Job Queues

Questions You'll Be Able to Answer After This Chapter

- Why are clusters useful?
- What are the costs of clustering?
- How can I convert a multiprocessing solution into a clustered solution?
- How does an IPython cluster work?
- How can I parallelize Pandas using Dask and Swifter?
- How does NSQ help with making robust production systems?

A *cluster* is commonly recognized to be a collection of computers working together to solve a common task. It could be viewed from the outside as a larger single system.

In the 1990s, the notion of using a cluster of commodity PCs on a local area network for clustered processing—known as a [Beowulf cluster](#)—became popular. Google later gave the practice a boost by using clusters of commodity PCs in its own data centers, particularly for running MapReduce tasks. At the other end of the scale, the [TOP500 project](#) ranks the most powerful computer systems each year; these typically have a clustered design, and the fastest machines all use Linux.

Amazon Web Services (AWS) is commonly used both for engineering production clusters in the cloud and for building on-demand clusters for short-lived tasks like machine learning. With AWS you can rent tiny to huge machines with 10s of CPUs and up to 768 GB of RAM for \$1 to \$15 an hour. Multiple GPUs can be rented at extra cost. Look at “[Using IPython Parallel to Support Research](#)” on page 319 and the

ElastiCluster package if you'd like to explore AWS or other providers for ad hoc clusters on compute-heavy or RAM-heavy tasks.

Different computing tasks require different configurations, sizes, and capabilities in a cluster. We'll define some common scenarios in this chapter.

Before you move to a clustered solution, do make sure that you have done the following:

- Profiled your system so you understand the bottlenecks
- Exploited compiler solutions like Numba and Cython
- Exploited multiple cores on a single machine (possibly a big machine with many cores) with Joblib or `multiprocessing`
- Exploited techniques for using less RAM

Keeping your system to one machine will make your life easier (even if the “one machine” is a really beefy computer with lots of RAM and many CPUs). Move to a cluster if you really need a *lot* of CPUs or the ability to process data from disks in parallel, or if you have production needs like high resiliency and rapid speed of response. Most research scenarios do not need resilience or scalability and are limited to few people, so the simplest solution is often the most sensible.

A benefit of staying on one *large* machine is that a tool like Dask can quickly parallelize your Pandas or plain Python code with no networking complications. Dask can also control a cluster of machines to parallelize Pandas, NumPy, and pure Python problems. Swifter automatically parallelizes some multicore single-machine cases by piggybacking on Dask. We introduce both Dask and Swifter later in this chapter.

Benefits of Clustering

The most obvious benefit of a cluster is that you can easily scale computing requirements—if you need to process more data or to get an answer faster, you just add more machines (or *nodes*).

By adding machines, you can also improve reliability. Each machine's components have a certain likelihood of failing, but with a good design, the failure of a number of components will not stop the operation of the cluster.

Clusters are also used to create systems that scale dynamically. A common use case is to cluster a set of servers that process web requests or associated data (e.g., resizing user photos, transcoding video, or transcribing speech) and to activate more servers as demand increases at certain times of the day.

Dynamic scaling is a very cost-effective way of dealing with nonuniform usage patterns, as long as the machine activation time is fast enough to deal with the speed of changing demand.



Consider the effort versus the reward of building a cluster. Whilst the parallelization gains of a cluster can feel attractive, do consider the costs associated with constructing and maintaining a cluster. They fit well for long-running processes in a production environment or for well-defined and oft-repeated R&D tasks. They are less attractive for variable and short-lived R&D tasks.

A subtler benefit of clustering is that clusters can be separated geographically but still centrally controlled. If one geographic area suffers an outage (due to a flood or power loss, for example), the other cluster can continue to work, perhaps with more processing units being added to handle the demand. Clusters also allow you to run heterogeneous software environments (e.g., different versions of operating systems and processing software), which *might* improve the robustness of the overall system—note, though, that this is definitely an expert-level topic!

Drawbacks of Clustering

Moving to a clustered solution requires a change in thinking. This is an evolution of the change in thinking required when you move from serial to parallel code, as we introduced back in [Chapter 9](#). Suddenly you have to consider what happens when you have more than one machine—you have latency between machines, you need to know if your other machines are working, and you need to keep all the machines running the same version of your software. System administration is probably your biggest challenge.

In addition, you normally have to think hard about the algorithms you are implementing and what happens once you have all these additional moving parts that may need to stay in sync. This additional planning can impose a heavy mental tax; it is likely to distract you from your core task, and once a system grows large enough, you'll probably need to add a dedicated engineer to your team.



We've tried to focus on using one machine efficiently in this book because we believe that life is easier if you're dealing with only one computer rather than a collection (though we confess it can be *way* more fun to play with a cluster—until it breaks). If you can scale vertically (by buying more RAM or more CPUs), it is worth investigating this approach in favor of clustering. Of course, your processing needs may exceed what's possible with vertical scaling, or the robustness of a cluster may be more important than having a single machine. If you're a single person working on this task, though, bear in mind also that running a cluster will suck up some of your time.

When designing a clustered solution, you'll need to remember that each machine's configuration might be different (each machine will have a different load and different local data). How will you get all the right data onto the machine that's processing your job? Does the latency involved in moving the job and the data amount to a problem? Do your jobs need to communicate partial results to one another? What happens if a process fails or a machine dies or some hardware wipes itself when several jobs are running? Failures can be introduced if you don't consider these questions.

You should also consider that failures *can be acceptable*. For example, you probably don't need 99.999% reliability when you're running a content-based web service—if on occasion a job fails (e.g., a picture doesn't get resized quickly enough) and the user is required to reload a page, that's something that everyone is already used to. It might not be the solution you want to give to the user, but accepting a little bit of failure typically reduces your engineering and management costs by a worthwhile margin. On the flip side, if a high-frequency trading system experiences failures, the cost of bad stock market trades could be considerable!

Maintaining a fixed infrastructure can become expensive. Machines are relatively cheap to purchase, but they have an awful habit of going wrong—automatic software upgrades can glitch, network cards fail, disks have write errors, power supplies can give spiky power that disrupts data, cosmic rays can flip a bit in a RAM module. The more computers you have, the more time will be lost to dealing with these issues. Sooner or later you'll want to bring in a system engineer who can deal with these problems, so add another \$100,000 to the budget. Using a cloud-based cluster can mitigate a lot of these problems (it costs more, but you don't have to deal with the hardware maintenance), and some cloud providers also offer a **spot-priced market** for cheap but temporary computing resources.

An insidious problem with a cluster that grows organically over time is that it's possible no one has documented how to restart it safely if everything gets turned off. If you don't have a documented restart plan, you should assume you'll have to write one at the worst possible time (one of your authors has been involved in debugging this sort

of problem on Christmas Eve—this is not the Christmas present you want!). At this point you’ll also learn just how long it can take each part of a system to get up to speed—it might take minutes for each part of a cluster to boot and to start to process jobs, so if you have 10 parts that operate in succession, it might take an hour to get the whole system running from cold. The consequence is that you might have an hour’s worth of backlogged data. Do you then have the necessary capacity to deal with this backlog in a timely fashion?

Slack behavior can be a cause of expensive mistakes, and complex and hard-to-anticipate behavior can cause unexpected and expensive outcomes. Let’s look at two high-profile cluster failures and see what lessons we can learn.

\$462 Million Wall Street Loss Through Poor Cluster Upgrade Strategy

In 2012, the high-frequency trading firm [Knight Capital lost \\$462 million](#) after a bug was introduced during a software upgrade in a cluster. The software made orders for more shares than customers had requested.

In the trading software, an older flag was repurposed for a new function. The upgrade was rolled out to seven of the eight live machines, but the eighth machine used older code to handle the flag, which resulted in the wrong trades being made. The Securities and Exchange Commission (SEC) noted that Knight Capital didn’t have a second technician review the upgrade and in fact had no established process for reviewing such an upgrade.

The underlying mistake seems to have had two causes. The first was that the software development process hadn’t removed an obsolete feature, so the stale code stayed around. The second was that no manual review process was in place to confirm that the upgrade was completed successfully.

Technical debt adds a cost that eventually has to be paid—preferably by taking time when not under pressure to remove the debt. Always use unit tests, both when building and when refactoring code. The lack of a written checklist to run through during system upgrades, along with a second pair of eyes, could cost you an expensive failure. There’s a reason that airplane pilots have to work through a takeoff checklist: it means that nobody ever skips the important steps, no matter how many times they might have done them before!

Skype’s 24-Hour Global Outage

Skype suffered a [24-hour planetwide failure](#) in 2010. Behind the scenes, Skype is supported by a peer-to-peer network. An overload in one part of the system (used to process offline instant messages) caused delayed responses from Windows clients; some versions of the Windows client didn’t properly handle the delayed responses

and crashed. In all, approximately 40% of the live clients crashed, including 25% of the public supernodes. Supernodes are critical to routing data in the network.

With 25% of the routing offline (it came back on, but slowly), the network overall was under great strain. The crashed Windows client nodes were also restarting and attempting to rejoin the network, adding a new volume of traffic on the already overloaded system. The supernodes have a back-off procedure if they experience too much load, so they started to shut down in response to the waves of traffic.

Skype became largely unavailable for 24 hours. The recovery process involved first setting up hundreds of new “mega-supernodes” configured to deal with the increased traffic, and then following up with thousands more. Over the coming days, the network recovered.

This incident caused a lot of embarrassment for Skype; clearly, it also changed its focus to damage limitation for several tense days. Customers were forced to look for alternative solutions for voice calls, which was likely a marketing boon for competitors.

Given the complexity of the network and the escalation of failures that occurred, this failure likely would have been hard both to predict and to plan for. The reason that *all* of the nodes on the network didn’t fail was due to different versions of the software and different platforms—there’s a reliability benefit to having a heterogeneous network rather than a homogeneous system.

Common Cluster Designs

It is common to start with a local ad hoc cluster of reasonably equivalent machines. You might wonder if you can add old computers to an ad hoc network, but typically older CPUs eat a lot of power and run very slowly, so they don’t contribute nearly as much as you might hope compared to one new, high-specification machine. An in-office cluster requires someone who can maintain it. A cluster on [Amazon’s EC2](#) or [Microsoft’s Azure](#), or one run by an academic institution, offloads the hardware support to the provider’s team.

If you have well-understood processing requirements, it might make sense to design a custom cluster—perhaps one that uses an InfiniBand high-speed interconnect in place of gigabit Ethernet, or one that uses a particular configuration of RAID drives that support your read, write, or resiliency requirements. You might want to combine CPUs and GPUs on some machines, or just default to CPUs.

You might want a massively decentralized processing cluster, like the ones used by projects such as *SETI@home* and *Folding@home* through [the Berkeley Open Infrastructure for Network Computing \(BOINC\) system](#). They share a centralized coordi-

nation system, but the computing nodes join and leave the project in an ad hoc fashion.

On top of the hardware design, you can run different software architectures. Queues of work are the most common and easiest to understand. Typically, jobs are put onto a queue and consumed by a processor. The result of the processing might go onto another queue for further processing, or it might be used as a final result (e.g., being added into a database). Message-passing systems are slightly different—messages get put onto a message bus and are then consumed by other machines. The messages might time out and get deleted, and they might be consumed by multiple machines. In a more complex system, processes talk to each other using interprocess communication—this can be considered an expert-level configuration, as there are lots of ways that you can set it up badly, which will result in you losing your sanity. Go down the IPC route only if you really know that you need it.

How to Start a Clustered Solution

The easiest way to start a clustered system is to begin with one machine that will run both the job server and a job processor (just one job processor for one CPU). If your tasks are CPU-bound, run one job processor per CPU; if your tasks are I/O-bound, run several per CPU. If they're RAM-bound, be careful that you don't run out of RAM. Get your single-machine solution working with one processor and then add more. Make your code fail in unpredictable ways (e.g., do a `1/0` in your code, use `kill -9 <pid>` on your worker, pull the power plug from the socket so the whole machine dies) to check if your system is robust.

Obviously, you'll want to do heavier testing than this—a unit test suite full of coding errors and artificial exceptions is good. Ian likes to throw in unexpected events, like having a processor run a set of jobs while an external process is systematically killing important processes and confirming that these all get restarted cleanly by whatever monitoring process is being used.

Once you have one running job processor, add a second. Check that you're not using too much RAM. Do you process jobs twice as fast as before?

Now introduce a second machine, with just one job processor on that new machine and no job processors on the coordinating machine. Does it process jobs as fast as when you had the processor on the coordinating machine? If not, why not? Is latency a problem? Do you have different configurations? Maybe you have different machine hardware, like CPUs, RAM, and cache sizes?

Now add another nine computers and test to see if you're processing jobs 10 times faster than before. If not, why not? Are network collisions now occurring that slow down your overall processing rate?

To reliably start the cluster's components when the machine boots, we tend to use either a `cron` job, [Circus](#), or [supervisord](#). Circus and `supervisord` are both Python-based and have been around for years. `cron` is old but very reliable if you're just starting scripts like a monitoring process that can start subprocesses as required.

Once you have a reliable cluster, you might want to introduce a random-killer tool like Netflix's [Chaos Monkey](#), which deliberately kills parts of your system to test them for resiliency. Your processes and your hardware will die eventually, and it doesn't hurt to know that you're likely to survive at least the errors you predict might happen.

Ways to Avoid Pain When Using Clusters

In one particularly painful experience Ian encountered, a series of queues in a clustered system ground to a halt. Later queues were not being consumed, so they filled up. Some of the machines ran out of RAM, so their processes died. Earlier queues were being processed but couldn't pass their results to the next queue, so they crashed. In the end the first queue was being filled but not consumed, so it crashed. After that, we were paying for data from a supplier that ultimately was discarded. You must sketch out some notes to consider the various ways your cluster will die and what will happen when (not *if*) it does. Will you lose data (and is this a problem)? Will you have a large backlog that's too painful to process?

Having a system that's easy to debug *probably* beats having a faster system. Engineering time and the cost of downtime are *probably* your largest expenses (this isn't true if you're running a missile defense program, but it is probably true for a start-up). Rather than shaving a few bytes by using a low-level compressed binary protocol, consider using human-readable text in JSON when passing messages. It does add an overhead for sending the messages and decoding them, but when you're left with a partial database after a core computer has caught fire, you'll be glad that you can read the important messages quickly as you work to bring the system back online.

Make sure it is cheap in time and money to deploy updates to the system—both operating system updates and new versions of your software. Every time anything changes in the cluster, you risk the system responding in odd ways if it is in a schizophrenic state. Make sure you use a deployment system like [Fabric](#), [Salt](#), [Chef](#), or [Puppet](#), or a system image like a Debian `.deb`, a RedHat `.rpm`, or an [Amazon Machine Image](#). Being able to robustly deploy an update that upgrades an entire cluster (with a report on any problems found) massively reduces stress during difficult times.

Positive reporting is useful. Every day, send an email to someone detailing the performance of the cluster. If that email doesn't turn up, that's a useful clue that something's happened. You'll probably want other early warning systems that'll notify you faster too; [Pingdom](#) and [Server Density](#) are particularly useful here. A

“dead man’s switch” that reacts to the absence of an event (e.g., [Dead Man’s Switch](#)) is another useful backup.

Reporting to the team on the health of the cluster is very useful. This might be an admin page inside a web application, or a separate report. [Ganglia](#) is great for this. Ian has seen a *Star Trek* LCARS-like interface running on a spare PC in an office that plays the “red alert” sound when problems are detected—that’s particularly effective at getting the attention of an entire office. We’ve even seen Arduinos driving analog instruments like old-fashioned boiler pressure gauges (they make a nice sound when the needle moves!) showing system load. This kind of reporting is important so that everyone understands the difference between “normal” and “this might ruin our Friday night!”

Two Clustering Solutions

In this section we introduce IPython Parallel and NSQ.

IPython clusters are easy to use on one machine with multiple cores. Since many researchers use IPython as their shell or work through Jupyter Notebooks, it is natural to also use it for parallel job control. Building a cluster requires a little bit of system administration knowledge. A huge win with IPython Parallel is that you can use remote clusters (Amazon’s AWS and EC2, for example) just as easily as local clusters.

NSQ is a production-ready queuing system. It has persistence (so if machines die, jobs can be picked up again by another machine) and strong mechanisms for scalability. With this greater power comes a slightly greater need for system administration and engineering skills. However, NSQ shines in its simplicity and ease of use. While many queuing systems exist (such as the popular [Kafka](#)), none have such as low a barrier for entry as NSQ.

Using IPython Parallel to Support Research

The IPython clustering support comes via the [IPython Parallel](#) project. IPython becomes an interface to local and remote processing engines where data can be pushed among the engines and jobs can be pushed to remote machines. Remote debugging is possible, and the message passing interface (MPI) is optionally supported. This same ZeroMQ communication mechanism powers the Jupyter Notebook interface.

This is great for a research setting—you can push jobs to machines in a local cluster, interact and debug if there’s a problem, push data to machines, and collect results back, all interactively. Note also that PyPy runs IPython and IPython Parallel. The combination might be very powerful (if you don’t use `numpy`).

Behind the scenes, ZeroMQ is used as the messaging middleware—be aware that ZeroMQ provides no security by design. If you’re building a cluster on a local network, you can avoid SSH authentication. If you need security, SSH is fully supported, but it makes configuration a little more involved—start on a local trusted network and build out as you learn how each component works.

The project is split into four components. An *engine* is an extension of the IPython kernel; it is a synchronous Python interpreter that runs your code. You’ll run a set of engines to enable parallel computing. A *controller* provides an interface to the engines; it is responsible for work distribution and supplies a *direct* interface and a *load-balanced* interface that provides a work scheduler. A *hub* keeps track of engines, schedulers, and clients. *Schedulers* hide the synchronous nature of the engines and provide an asynchronous interface.

On the laptop, we start four engines using `ipcluster start -n 4`. In [Example 10-1](#), we start IPython and check that a local `Client` can see our four local engines. We can address all four engines using `c[:]`, and we apply a function to each engine—`apply_sync` takes a callable, so we supply a zero-argument `lambda` that will return a string. Each of our four local engines will run one of these functions, returning the same result.

Example 10-1. Testing that we can see the local engines in IPython

```
In [1]: import ipyparallel as ipp  
In [2]: c = ipp.Client()  
In [3]: print(c.ids)  
[0, 1, 2, 3]  
  
In [4]: c[:].apply_sync(lambda: "Hello High Performance Pythonistas!")  
Out[4]:  
['Hello High Performance Pythonistas!',  
 'Hello High Performance Pythonistas!',  
 'Hello High Performance Pythonistas!',  
 'Hello High Performance Pythonistas!']
```

The engines we’ve constructed are now in an empty state. If we import modules locally, they won’t be imported into the remote engines.

A clean way to import both locally and remotely is to use the `sync_imports` context manager. In [Example 10-2](#), we’ll `import os` on both the local IPython and the four connected engines and then call `apply_sync` again on the four engines to fetch their PIDs.

If we didn't do the remote imports, we'd get a `NameError`, as the remote engines wouldn't know about the `os` module. We can also use `execute` to run any Python command remotely on the engines.

Example 10-2. Importing modules into our remote engines

```
In [5]: dview=c[:] # this is a direct view (not a load-balanced view)

In [6]: with dview.sync_imports():
....:     import os
....:

importing os on engine(s)

In [7]: dview.apply_sync(lambda:os.getpid())
Out[7]: [16158, 16159, 16160, 16163]

In [8]: dview.execute("import sys") # another way to execute commands remotely
```

You'll want to push data to the engines. The `push` command shown in [Example 10-3](#) lets you send a dictionary of items that are added to the global namespace of each engine. There's a corresponding `pull` to retrieve items: you give it keys, and it'll return the corresponding values from each of the engines.

Example 10-3. Pushing shared data to the engines

```
In [9]: dview.push({'shared_data':[50, 100]})

In [10]: dview.apply_sync(lambda:len(shared_data))
Out[10]: [2, 2, 2, 2]
```

In [Example 10-4](#), we use these four engines to estimate pi. This time we use the `@require` decorator to import the `random` module in the engines. We use a direct view to send our work out to the engines; this blocks until all the results come back. Then we estimate pi as we did in [Example 9-1](#).

Example 10-4. Estimating pi using our local cluster

```
import time
import ipyparallel as ipp
from ipyparallel import require

@require('random')
def estimate_nbr_points_in_quarter_circle(nbr_estimates):
    ...
    return nbr_trials_in_quarter_unit_circle

if __name__ == "__main__":
```

```

c = ippp.Client()
nbr_engines = len(c.ids)
print("We're using {} engines".format(nbr_engines))
nbr_samples_in_total = 1e8
nbr_parallel_blocks = 4

dview = c[:]

nbr_samples_per_worker = nbr_samples_in_total / nbr_parallel_blocks
t1 = time.time()
nbr_in_quarter_unit_circles = \
    dview.apply_sync(estimate_nbr_points_in_quarter_circle,
                    nbr_samples_per_worker)
print("Estimates made:", nbr_in_quarter_unit_circles)

nbr_jobs = len(nbr_in_quarter_unit_circles)
pi_estimate = sum(nbr_in_quarter_unit_circles) * 4 / nbr_samples_in_total
print("Estimated pi", pi_estimate)
print("Delta:", time.time() - t1)

```

In Example 10-5 we run this on our four local engines. As in Figure 9-5, this takes approximately 20 seconds on the laptop.

Example 10-5. Estimating pi using our local cluster in IPython

```

In [1]: %run pi_ipython_cluster.py
We're using 4 engines
Estimates made: [19636752, 19634225, 19635101, 19638841]
Estimated pi 3.14179676
Delta: 20.68650197982788

```

IPython Parallel offers much more than what's shown here. Asynchronous jobs and mappings over larger input ranges are, of course, possible. MPI is supported, which can provide efficient data sharing. The Joblib library introduced in “Replacing multiprocessing with Joblib” on page 260 can use IPython Parallel as a backend along with Dask (which we introduce in “Parallel Pandas with Dask” on page 322).

One particularly powerful feature of IPython Parallel is that it allows you to use larger clustering environments, including supercomputers and cloud services like Amazon’s EC2. The [ElastiCluster project](#) has support for common parallel environments such as IPython and for deployment targets, including AWS, Azure, and OpenStack.

Parallel Pandas with Dask

Dask aims to provide a suite of parallelization solutions that scales from a single core on a laptop to multicore machines to thousands of cores in a cluster. Think of it as “Apache Spark lite.” If you don’t need all of Apache Spark’s functionality (which includes replicated writes and multimachine failover) and you don’t want to support

a second computation and storage environment, then Dask may provide the parallelized and bigger-than-RAM solution you’re after.

A task graph is constructed for the lazy evaluation of a number of computation scenarios, including pure Python, scientific Python, and machine learning with small, medium, and big datasets:

Bag

`bag` enables parallelized computation on unstructured and semistructured data, including text files, JSON or user-defined objects. `map`, `filter`, and `groupby` are supported on generic Python objects, including lists and sets.

Array

`array` enables distributed and larger-than-RAM `numpy` operations. Many common operations are supported, including some linear algebra functions. Operations that are inefficient across cores (sorting, for example, and many linear algebra operations) are not supported. Threads are used, as NumPy has good thread support, so data doesn’t have to be copied during parallelized operations.

Distributed DataFrame

`dataframe` enables distributed and larger-than-RAM Pandas operations; behind the scenes, Pandas is used to represent partial DataFrames that have been partitioned using their index. Operations are lazily computed using `.compute()` and otherwise look very similar to their Pandas counterparts. Supported functions include `groupby-aggregate`, `groupby-apply`, `value_counts`, `drop_duplicates`, and `merge`. By default, threads are used, but as Pandas is more GIL-bound than NumPy, you may want to look at the Process or Distributed scheduler options.

Delayed

`delayed` extends the idea we introduced with Joblib in “[Replacing multiprocessing with Joblib](#)” on page 260 to parallelize *chains* of arbitrary Python functions in a lazy fashion. A `visualize()` function will draw the task graph to assist in diagnosing issues.

Futures

The `Client` interface enables immediate execution and evolution of tasks, unlike `delayed`, which is lazy and doesn’t allow operations like adding or destroying tasks. The `Future` interface includes `Queue` and `Lock` to support task collaboration.

Dask-ML

A scikit-learn-like interface is provided for scalable machine learning. Dask-ML provides cluster support to some scikit-learn algorithms, and it reimplements some algorithms (e.g., the `linear_model` set) using Dask to enable learning on big data. It closes some of the gap to the Apache Spark distributed machine

learning toolkit. It also provides support for XGBoost and TensorFlow to be used in a Dask cluster.

For Pandas users, Dask can help in two use cases: larger-than-RAM datasets and a desire for multicore parallelization.

If your dataset is larger than Pandas can fit into RAM, Dask can split the dataset by rows into a set of partitioned DataFrames called a *Distributed DataFrame*. These DataFrames are split by their index; a subset of operations can be performed across each partition. As an example, if you have a set of multi-GB CSV files and want to calculate `value_counts` across all the files, Dask will perform partial `value_counts` on each DataFrame (one per file) and then combine the results into a single set of counts.

A second use case is to take advantage of the multiple cores on your laptop (and just as easily across a cluster); we'll examine this use case here. Recall that in [Example 6-24](#), we calculated the slope of the line across rows of values in a DataFrame with various approaches. Let's use the two fastest approaches and parallelize them with Dask.



You can use Dask (and Swifter, discussed in the next section) to parallelize any side-effect-free function that you'd usually use in an `apply` call. Ian has done this for numeric calculations and for calculating text metrics on multiple columns of text in a large DataFrame.

With Dask, we have to specify the number of *partitions* to make from our DataFrame; a good rule of thumb is to use at least as many partitions as cores so that each core can be used. In [Example 10-6](#), we ask for eight partitions. We use `dd.from_pandas` to convert our regular Pandas DataFrame into a Dask Distributed DataFrame split into eight equal-sized sections.

We call our familiar `ddf.apply` on the Distributed DataFrame, specifying our function `ols_lsqr` and the optional expected return type via the `meta` argument. Dask requires us to specify when we should apply the computation with the `compute()` call; here, we specify the use of `processes` rather than the default `threads` to spread our work over multiple cores, avoiding Python's GIL.

Example 10-6. Calculating line slopes with multiple cores using Dask

```
import dask.dataframe as dd

N_PARTITIONS = 8
ddf = dd.from_pandas(df, npartitions=N_PARTITIONS, sort=False)
SCHEDULER = "processes"
```

```
results = ddf.apply(ols_lstsq, axis=1, meta=(None, 'float64',)). \
    compute(scheduler=SCHEDULER)
```

Running `ols_lstsq_raw` in [Example 10-7](#) with the same eight partitions (on four cores with four hyperthreads) in Ian’s laptop, we go from the previous single-threaded `apply` result of 6.8 seconds to 1.5 seconds—almost a 5× speedup.

Example 10-7. Calculating line slopes with multiple cores using Dask

```
results = ddf.apply(ols_lstsq_raw, axis=1, meta=(None, 'float64',), raw=True). \
    compute(scheduler=SCHEDULER)
```

Running `ols_lstsq_raw` with the same eight partitions takes us from the previous single-threaded `apply` result of 5.3 seconds with `raw=True` to 1.2 seconds—also almost a 5× speedup.

If we also use the compiled Numba function from “[Numba to Compile NumPy for Pandas](#)” on page 182 with `raw=True`, our runtime drops from 0.58 seconds to 0.3 seconds—a further 2× speedup. Functions compiled with Numba using NumPy arrays on Pandas DataFrames work well with Dask for very little effort.

Parallelized apply with Swifter on Dask

`Swifter` builds on Dask to provide three parallelized options with very simple calls—`apply`, `resample`, and `rolling`. Behind the scenes, it takes a subsample of your DataFrame and attempts to vectorize your function call. If that works, `Swifter` will apply it; if it works but it is slow, `Swifter` will run it on multiple cores using Dask.

Since `Swifter` uses heuristics to determine how to run your code, it could run slower than if you didn’t use it at all—but the “cost” of trying it is one line of effort. It is well worth evaluating.

`Swifter` makes its own decisions about how many cores to use with Dask and how many rows to sample for its evaluation; as a result, in [Example 10-8](#) we see the call to `df.swifter...apply()` looks just like a regular call to `df.apply`. In this case, we’ve disabled the progress bar; the progress bar works fine in a Jupyter Notebook using the excellent `tqdm` library.

Example 10-8. Calculating line slopes with multiple cores using Dask

```
import swifter

results = df.swifter.progress_bar(False).apply(ols_lstsq_raw, axis=1, raw=True)
```

Swifter with `ols_lstsq_raw` and no partitioning choices takes our previous single-threaded result of 5.3 seconds down to 1.6 seconds. For this particular function and dataset, this is not as fast as the slightly longer Dask solution we've just looked at, but it does offer a 3× speedup for only one line of code. For different functions and datasets, you'll see different results; it is definitely worth an experiment to see whether you can achieve a very easy win.

Vaex for bigger-than-RAM DataFrames

[Vaex](#) is an intriguing new library that provides a Pandas DataFrame-like structure that has built-in support for larger-than-RAM computations. It neatly combines the features of Pandas and Dask into a single package.

Vaex uses lazy computation to compute column results just-in-time; it'll compute on only the subset of rows required by the user. If, for example, you ask for a sum on a billion rows between two columns and you ask for only a *sample* of those rows as the result, Vaex will touch only the data for that sample and will not compute the sum for all of the nonsampled rows. For interactive work and visualization-driven investigation, this can be very efficient.

Pandas's support of strings comes from CPython; it is GIL-bound, and the string objects are larger objects that are scattered in memory and do not support vectorized operations. Vaex uses its own custom string library, which enables significantly faster string-based operations with a similar Pandas-like interface.

If you're working on string-heavy DataFrames or larger-than-RAM datasets, Vaex is an obvious choice for evaluation. If you commonly work on subsets of a DataFrame, the implicit lazy evaluation may make your workflow simpler than adding Dask to Pandas DataFrames.

NSQ for Robust Production Clustering

In a production environment, you will need a solution that is more robust than the other solutions we've talked about so far. This is because during the everyday operation of your cluster, nodes may become unavailable, code may crash, networks may go down, or one of the other thousands of problems that can happen may happen. The problem is that all the previous systems have had one computer where commands are issued, and a limited and static number of computers that read the commands and execute them. We would instead like a system that can have multiple actors communicating via a message bus—this would allow us to have an arbitrary and constantly changing number of message creators and consumers.

One simple solution to these problems is [NSQ](#), a highly performant distributed messaging platform. While it is written in GO, it is completely data format and language agnostic. As a result, there are libraries in many languages, and the basic

interface into NSQ is a REST API that requires only the ability to make HTTP calls. Furthermore, we can send messages in any format we want: JSON, Pickle, msgpack, and so on. Most importantly, however, it provides fundamental guarantees regarding message delivery, and it does all of this using two simple design patterns: queues and pub/subs.



We picked NSQ to discuss because it is simple to use and generally performant. Most importantly for our purposes, it clearly highlights the considerations you must make when thinking about queuing and message passing in a cluster. However, other solutions such as ZeroMQ, Amazon's SQS, Celery, or even Redis may be better suited for your application.

Queues

A *queue* is a type of buffer for messages. Whenever you want to send a message to another part of your processing pipeline, you send it to the queue, and it'll wait there until a worker is available. A queue is most useful in distributed processing when an imbalance exists between production and consumption. When this imbalance occurs, we can simply scale horizontally by adding more data consumers until the message production rate and the consumption rate are equal. In addition, if the computers responsible for consuming messages go down, the messages are not lost but are simply queued until a consumer is available, thus giving us message delivery guarantees.

For example, let's say we would like to process new recommendations for a user every time that user rates a new item on our site. If we didn't have a queue, the "rate" action would directly call the "recalculate-recommendations" action, regardless of how busy the servers dealing with recommendations were. If all of a sudden thousands of users decided to rate something, our recommendation servers could get so swamped with requests that they could start timing out, dropping messages, and generally becoming unresponsive!

On the other hand, with a queue, the recommendation servers ask for more tasks when they are ready. A new "rate" action would put a new task on the queue, and when a recommendation server becomes ready to do more work, it would grab the task from the queue and process it. In this setup, if more users than normal start rating items, our queue would fill up and act as a buffer for the recommendation servers—their workload would be unaffected, and they could still process messages until the queue was empty.

One potential problem with this is that if a queue becomes completely overwhelmed with work, it will be storing quite a lot of messages. NSQ solves this by having multiple storage backends—when there aren't many messages, they are stored in memory, and as more messages start coming in, the messages get put onto disk.



Generally, when working with queued systems, it is a good idea to try to have the downstream systems (i.e., the recommendation systems in the preceding example) be at 60% capacity with a normal workload. This is a good compromise between allocating too many resources for a problem and giving your servers enough extra power for when the amount of work increases beyond normal levels.

Pub/sub

A *pub/sub* (short for *publisher/subscriber*), on the other hand, describes who gets what messages. A data publisher can push data out of a particular topic, and data subscribers can subscribe to different feeds of data. Whenever the publisher puts out a piece of information, it gets sent to all the subscribers—each gets an identical copy of the original information. You can think of this like a newspaper: many people can subscribe to a particular newspaper, and whenever a new edition of the newspaper comes out, every subscriber gets an identical copy of it. In addition, the producer of the newspaper doesn’t need to know all the people its papers are being sent to. As a result, publishers and subscribers are decoupled from each other, which allows our system to be more robust as our network changes while still in production.

In addition, NSQ adds the notion of a *data consumer*; that is, multiple processes can be connected to the same data subscription. Whenever a new piece of data comes out, every subscriber gets a copy of the data; however, only one consumer of each subscription sees that data. In the newspaper analogy, you can think of this as having multiple people in the same household who read the newspaper. The publisher will deliver one paper to the house, since that house has only one subscription, and whoever in the house gets to it first gets to read that data. Each subscriber’s consumers do the same processing to a message when they see it; however, they can potentially be on multiple computers and thus add more processing power to the entire pool.

We can see a depiction of this pub/sub/consumer paradigm in [Figure 10-1](#). If a new message gets published on the “clicks” topic, all the subscribers (or, in NSQ parlance, *channels*—i.e., “metrics,” “spam_analysis,” and “archive”) will get a copy. Each subscriber is composed of one or more consumers, which represent actual processes that react to the messages. In the case of the “metrics” subscriber, only one consumer will see the new message. The next message will go to another consumer, and so on.

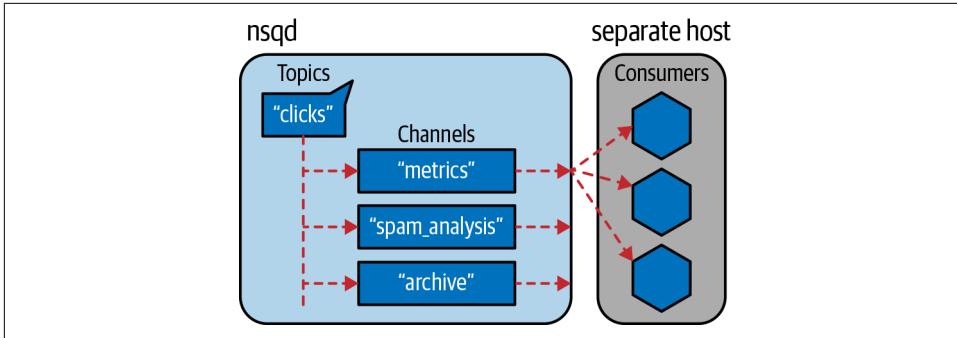


Figure 10-1. NSQ’s pub/sub-like topology

The benefit of spreading the messages among a potentially large pool of consumers is essentially automatic load balancing. If a message takes quite a long time to process, that consumer will not signal to NSQ that it is ready for more messages until it’s done, and thus the other consumers will get the majority of future messages (until that original consumer is ready to process again). In addition, it allows existing consumers to disconnect (whether by choice or because of failure) and new consumers to connect to the cluster while still maintaining processing power within a particular subscription group. For example, if we find that “metrics” takes quite a while to process and often is not keeping up with demand, we can simply add more processes to the consumer pool for that subscription group, giving us more processing power. On the other hand, if we see that most of our processes are idle (i.e., not getting any messages), we can easily remove consumers from this subscription pool.

It is also important to note that anything can publish data. A consumer doesn’t simply need to be a consumer—it can consume data from one topic and then publish it to another topic. In fact, this chain is an important workflow when it comes to this paradigm for distributed computing. Consumers will read from a topic of data, transform the data in some way, and then publish the data onto a new topic that other consumers can further transform. In this way, different topics represent different data, subscription groups represent different transformations on the data, and consumers are the actual workers who transform individual messages.

Furthermore, this system provides an incredible redundancy. There can be many nsqd processes that each consumer connects to, and there can be many consumers connected to a particular subscription. This makes it so that no single point of failure exists, and your system will be robust even if several machines disappear. We can see in [Figure 10-2](#) that even if one of the computers in the diagram goes down, the system is still able to deliver and process messages. In addition, since NSQ saves pending messages to disk when shutting down, unless the hardware loss is catastrophic, your data will most likely still be intact and be delivered. Last, if a consumer is shut down before responding to a particular message, NSQ will resend that message to another

consumer. This means that even as consumers get shut down, we know that all the messages in a topic will be responded to at least once.¹

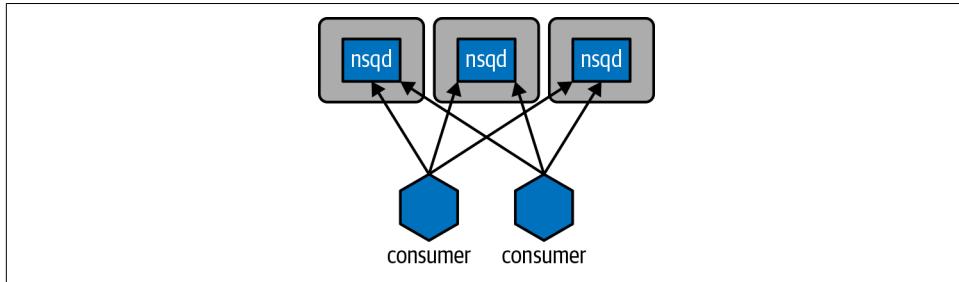


Figure 10-2. NSQ connection topology

Distributed Prime Calculation

Code that uses NSQ is generally asynchronous (see [Chapter 8](#) for a full explanation), although it doesn't necessarily have to be.² In the following example, we will create a pool of workers that read from a topic called *numbers* where the messages are simply JSON blobs with numbers in them. The consumers will read this topic, find out if the numbers are primes, and then write to another topic, depending on whether the numbers were prime. This will give us two new topics, *prime* and *non_prime*, that other consumers can connect to in order to do more calculations.³



pynsq (last release Nov 11, 2018) depends on a very outdated release of tornado (4.5.3, from Jan 6, 2018). This is a good example use case for Docker (discussed in “[Docker](#) on page 335”).

As we've said before, there are many benefits to doing CPU-bound work like this. First, we have all the guarantees of robustness, which may or may not be useful for this project. More importantly, however, we get automatic load balancing. That means that if one consumer gets a number that takes a particularly long time to process, the other consumers will pick up the slack.

¹ This can be quite advantageous when we're working in AWS, where we can have our nsqd processes running on a reserved instance and our consumers working on a cluster of spot instances.

² This asynchronicity comes from NSQ's protocol for sending messages to consumers being push-based. This makes it so our code can have an asynchronous read from our connection to NSQ happen in the background and wake up when a message is found.

³ This sort of chaining of data analysis is called *pipelining* and can be an effective way to perform multiple types of analysis on the same data efficiently.

We create a consumer by creating an `nsq.Reader` object with the topic and subscription group specified (as can be seen at the end of [Example 10-9](#)). We also must specify the location of the running `nsqd` instance (or the `nsql lookupd` instance, which we will not get into in this section). In addition, we specify a *handler*, which is simply a function that gets called for each message from the topic. To create a producer, we create an `nsq.Writer` object and specify the location of one or more `nsqd` instances to write to. This gives us the ability to write to `nsq`, simply by specifying the topic name and the message.⁴

Example 10-9. Distributed prime calculation with NSQ

```
import json
from functools import partial
from math import sqrt

import nsq

def is_prime(number):
    if number % 2 == 0:
        return False
    for i in range(3, int(sqrt(number)) + 1, 2):
        if number % i == 0:
            return False
    return True

def write_message(topic, data, writer):
    response = writer.pub(topic, data)
    if isinstance(response, nsq.Error):
        print("Error with Message: {}: {}".format(data, response))
        return write_message(data, writer)
    else:
        print("Published Message: ", data)

def calculate_prime(message, writer):
    data = json.loads(message.body)

    prime = is_prime(data["number"])
    data["prime"] = prime
    if prime:
        topic = "prime"
    else:
        topic = "non_prime"

    output_message = json.dumps(data).encode("utf8")
    write_message(topic, output_message, writer)
```

⁴ You can also easily publish a message manually with an HTTP call; however, this `nsq.Writer` object simplifies much of the error handling.

```

message.finish() ❶

if __name__ == "__main__":
    writer = nsq.Writer(["127.0.0.1:4150"])
    handler = partial(calculate_prime, writer=writer)
    reader = nsq.Reader(
        message_handler=handler,
        nsqd_tcp_addresses=["127.0.0.1:4150"],
        topic="numbers",
        channel="worker_group_a",
    )
    nsq.run()

```

- ❶ We must signal to NSQ when we are done with a message. This will make sure the message is not redelivered to another reader in case of failure.



We can handle messages asynchronously by enabling `message.enable_async()` in the message handler after the message is received. However, note that NSQ uses the older callback mechanisms with tornado's IOLoop (discussed in “[tornado](#)” on page 226).

To set up the NSQ ecosystem, start an instance of `nsqd` on our local machine:⁵

```
$ nsqd
[nsqd] 2020/01/25 13:36:39.333097 INFO: nsqd v1.2.0 (built w/go1.12.9)
[nsqd] 2020/01/25 13:36:39.333141 INFO: ID: 235
[nsqd] 2020/01/25 13:36:39.333352 INFO: NSQ: persisting topic/channel metadata
                                to nsqd.dat
[nsqd] 2020/01/25 13:36:39.340583 INFO: TCP: listening on [::]:4150
[nsqd] 2020/01/25 13:36:39.340630 INFO: HTTP: listening on [::]:4151
```

Now we can start as many instances of our Python code ([Example 10-9](#)) as we want. In fact, we can have these instances running on other computers as long as the reference to the `nsqd_tcp_address` in the instantiation of the `nsq.Reader` is still valid. These consumers will connect to `nsqd` and wait for messages to be published on the *numbers* topic.

Data can be published to the *numbers* topic in many ways. We will use command-line tools to do this, since knowing how to poke and prod a system goes a long way in understanding how to properly deal with it. We can simply use the HTTP interface to publish messages to the topic:

⁵ For this example, we installed NSQ straight onto our system by unpacking the provided binaries into our PATH environment variable. Alternatively, you can use Docker, discussed in “[Docker](#)” on page 335, to easily run the latest versions.

```
$ for i in `seq 10000`  
> do  
>   echo {"number": $i} | curl -d@- "http://127.0.0.1:4151/pub?topic=numbers"  
> done
```

As this command starts running, we are publishing messages with different numbers in them to the *numbers* topic. At the same time, all of our producers will start outputting status messages indicating that they have seen and processed messages. In addition, these numbers are being published to either the *prime* or the *non_prime* topic. This allows us to have other data consumers that connect to either of these topics to get a filtered subset of our original data. For example, an application that requires only the prime numbers can simply connect to the *prime* topic and constantly have new primes for its calculation. We can see the status of our calculation by using the `stats` HTTP endpoint for nsqd:

```
$ curl "http://127.0.0.1:4151/stats"  
nsqd v1.2.0 (built w/go1.12.9)  
start_time 2020-01-25T14:16:35Z  
uptime 26.087839544s  
  
Health: OK  
  
Memory:  
  heap_objects          25973  
  heap_idle_bytes       61399040  
  heap_in_use_bytes     4661248  
  heap_released_bytes   0  
  gc_pause_usec_100     43  
  gc_pause_usec_99      43  
  gc_pause_usec_95      43  
  next_gc_bytes         4194304  
  gc_total_runs          6  
  
Topics:  
  [non_prime] depth: 902    be-depth: 0      msgs: 902      e2e%:  
  [numbers] depth: 0        be-depth: 0      msgs: 3009      e2e%:  
    [worker_group_a] depth: 1926  be-depth: 0      inflt: 1  
      def: 0    re-q: 0      timeout: 0  
      msgs: 3009      e2e%:  
    [V2 electron] state: 3 inflt: 1    rdy: 1      fin: 1082  
      re-q: 0    msgs: 1083      connected: 15s  
  
  [prime] depth: 180    be-depth: 0      msgs: 180      e2e%:  
  
Producers:  
  [V2 electron] msgs: 1082      connected: 15s  
  [prime] msgs: 180  
  [non_prime] msgs: 902
```

We can see here that the *numbers* topic has one subscription group, *worker_group_a*, with one consumer. In addition, the subscription group has a large depth of 1,926 messages, which means that we are putting messages into NSQ faster than we can process them. This would be an indication to add more consumers so that we have more processing power to get through more messages. Furthermore, we can see that this particular consumer has been connected for 15 seconds, has processed 1,083 messages, and currently has 1 message in flight. This status endpoint gives quite a good deal of information for debugging your NSQ setup! Last, we see the *prime* and *non_prime* topics, which have no subscribers or consumers. This means that the messages will be stored until a subscriber comes requesting the data.



In production systems, you can use the even more powerful tool `nsqadmin`, which provides a web interface with very detailed overviews of all topics/subscribers and consumers. In addition, it allows you to easily pause and delete subscribers and topics.

To actually see the messages, we would create a new consumer for the *prime* (or *non_prime*) topic that simply archives the results to a file or database. Alternatively, we can use the `nsq_tail` tool to take a peek at the data and see what it contains:

```
$ nsq_tail --topic prime -n 5 --nsqd-tcp-address=127.0.0.1:4150
2020/01/25 14:34:17 Adding consumer for topic: prime
2020/01/25 14:34:17 INF      1 [prime/tail574169#ephemeral] (127.0.0.1:4150)
                           connecting to nsqd
{"number": 1, "prime": true}
{"number": 3, "prime": true}
 {"number": 5, "prime": true}
 {"number": 7, "prime": true}
 {"number": 11, "prime": true}
```

Other Clustering Tools to Look At

Job processing systems using queues have existed since the start of the computer science industry, back when computers were very slow and lots of jobs needed to be processed. As a result, there are *many* libraries for queues, and many of these can be used in a cluster configuration. We strongly suggest that you pick a mature library with an active community behind it and supporting the same feature set that you'll need without too many additional features.

The more features a library has, the more ways you'll find to misconfigure it and waste time on debugging. Simplicity is *generally* the right aim when dealing with clustered solutions. Here are a few of the more commonly used clustering solutions:

- **ZeroMQ** is a low-level and performant messaging library that enables you to send messages between nodes. It natively supports pub/sub paradigms and can

also communicate over multiple types of transports (TCP, UDP, WebSocket, etc). It is quite low level and doesn't provide many useful abstractions, which can make its use a bit difficult. That being said, it's in use in Jupyter, Auth0, Spotify, and many more places!

- [Celery](#) (BSD license) is a widely used asynchronous task queue using a distributed messaging architecture, written in Python. It supports Python, PyPy, and Jython. It typically uses RabbitMQ as the message broker, but it also supports Redis, MongoDB, and other storage systems. It is often used in web development projects. Andrew Godwin discusses Celery in [“Task Queues at Lanyrd.com \(2014\)” on page 430](#).
- [Airflow](#) and [Luigi](#) use directed acyclic graphs to chain dependent jobs into sequences that run reliably, with monitoring and reporting services. They're widely used in industry for data science tasks, and we recommend reviewing these before you embark on a custom solution.
- [Amazon's Simple Queue Service \(SQS\)](#) is a job processing system integrated into AWS. Job consumers and producers can live inside AWS or can be external, so SQS is easy to start with and supports easy migration into the cloud. Library support exists for many languages.

Docker

[Docker](#) is a tool of general importance in the Python ecosystem. However, the problems it solves are particularly important when dealing with a large team or a cluster. In particular, Docker helps to create reproducible environments in which to run your code, share/control runtime environments, easily share runnable code between team members, and deploy code to a cluster of nodes based on resource needs.

Docker's Performance

One common misconception about Docker is that it substantially slows down the runtime performance of the applications it is running. While this can be true in some cases, it generally is not. Furthermore, most of the performance degradations can almost always be removed with some easy configuration changes.

In terms of CPU and memory access, Docker (and all other container-based solutions) will *not* lead to any performance degradations. This is because Docker simply creates a special namespace within the host operating system where the code can run normally, albeit with separate constraints from other running programs. Essentially, Docker code accesses the CPU and memory in the same way that every other

program on the computer does; however, it can have a separate set of configuration values to fine-tune resource limits.⁶

This is because Docker is an instance of OS-level virtualization, as opposed to hardware virtualization such as VMware or VirtualBox. With hardware virtualization, software runs on “fake” hardware that introduces overhead accessing all resources. On the other hand, OS virtualization uses the native hardware but runs on a “fake” operating system. Thanks to the `cgroups` Linux feature, this “fake” operating system can be tightly coupled to the running operating system, which gives the possibility of running with almost no overhead.



`cgroups` is specifically a feature in the Linux kernel. As a result, performance implications discussed here are restricted to Linux systems. In fact, to run Docker on macOS or Windows, we first must run the Linux kernel in a hardware-virtualized environment. Docker Machine, the application helping streamline this process, uses VirtualBox to accomplish this. As a result, you will see performance overhead from the hardware-virtualized portion of the process. This overhead will be greatly reduced when running on a Linux system, where hardware virtualization is not needed.

As an example, let’s create a simple Docker container to run the 2D diffusion code from [Example 6-17](#). As a baseline, we can run the code on the host system’s Python to get a benchmark:

```
$ python diffusion_numpy_memory2.py
Runtime for 100 iterations with grid size (256, 256): 1.4418s
```

To create our Docker container, we have to make a directory that contains the Python file `diffusion_numpy_memory2.py`, a `pip` requirements file for dependencies, and a `Dockerfile`, as shown in [Example 10-10](#).

Example 10-10. Simple Docker container

```
$ ls
diffusion_numpy_memory2.py
Dockerfile
requirements.txt

$ cat requirements.txt
numpy>=1.18.0
```

⁶ This fine-tuning can, for example, be used to adjust the amount of memory a process has access to, or which CPUs or even how much of the CPU it can use.

```
$ cat Dockerfile
FROM python:3.7

WORKDIR /usr/src/app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
CMD python ./diffusion_numpy_memory2.py
```

The *Dockerfile* starts by stating what container we'd like to use as our base. These base containers can be a wide selection of Linux-based operating systems or a higher-level service. The Python Foundation provides [official containers for all major Python versions](#), which makes selecting the Python version you'd like to use incredibly simple. Next, we define the location of our working directory (the selection of `/usr/src/app` is arbitrary), copy our requirements file into it, and begin setting up our environment as we normally would on our local machine, using `RUN` commands.

One major difference between setting up your development environment normally and on Docker are the `COPY` commands. They copy files from the local directory into the container. For example, the `requirements.txt` file is copied into the container so that it is there for the `pip install` command. Finally, at the end of the *Dockerfile*, we copy all the files from the current directory into the container and tell Docker to run `python ./diffusion_numpy_memory2.py` when the container starts.



In the *Dockerfile* in [Example 10-10](#), beginners often wonder why we first copy only the requirements file and then later copy the entire directory into the container. When building a container, Docker tries hard to cache each step in the build process. To determine whether the cache is still valid, the contents of the files being copied back and forth are checked. By first copying only the requirements file and *then* moving the rest of the directory, we will have to run `pip install` only if the requirements file changes. If only the Python source has changed, a new build will use cached build steps and skip straight to the second `COPY` command.

Now we are ready to build and run the container, which can be named and tagged. Container names generally take the format `<username>/<project-name>`,⁷ while the optional tag generally is either descriptive of the current version of the code or simply the tag `latest` (this is the default and will be applied automatically if no tag is specified). To help with versioning, it is general convention to always tag the most recent

⁷ The `username` portion of the container name is useful when also pushing built containers to a repository.

build with `latest` (which will get overwritten when a new build is made) as well as a descriptive tag so that we can easily find this version again in the future:

```
$ docker build -t high_performance/diffusion2d:numpy-memory2 \
    -t high_performance/diffusion2d:latest .
Sending build context to Docker daemon 5.632kB
Step 1/6 : FROM python:3.7
--> 3624d01978a1
Step 2/6 : WORKDIR /usr/src/app
--> Running in 04efc02f2ddf
Removing intermediate container 04efc02f2ddf
--> 9110a0496749
Step 3/6 : COPY requirements.txt ./
--> 45f9ecf91f74
Step 4/6 : RUN pip install --no-cache-dir -r requirements.txt
--> Running in 8505623a9fa6
Collecting numpy>=1.18.0 (from -r requirements.txt (line 1))
  Downloading https://.../numpy-1.18.0-cp37-cp37m-manylinux1_x86_64.whl (20.1MB)
Installing collected packages: numpy
Successfully installed numpy-1.18.0
You are using pip version 18.1, however version 19.3.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Removing intermediate container 8505623a9fa6
--> 5abc2df1116f
Step 5/6 : COPY . .
--> 52727a6e9715
Step 6/6 : CMD python ./diffusion_numpy_memory2.py
--> Running in c1e885b926b3
Removing intermediate container c1e885b926b3
--> 892a33754f1d
Successfully built 892a33754f1d
Successfully tagged high_performance/diffusion2d:numpy-memory2
Successfully tagged high_performance/diffusion2d:latest

$ docker run high_performance/diffusion2d:numpy-memory2
Runtime for 100 iterations with grid size (256, 256): 1.4493s
```

We can see that at its core, Docker is not slower than running on the host machine in any meaningful way when the task relies mainly on CPU/memory. However, as with anything, there is no free lunch, and at times Docker performance suffers. While a full discussion of optimizing Docker containers is outside the scope of this book, we offer the following list of considerations for when you are creating Docker containers for high performance code:

- Be wary of copying too much data into a Docker container or even having too much data in the same directory as a Docker build. If the `build context`, as advertised by the first line of the `docker build` command, is too large, performance can suffer (remedy this with a `.dockerignore` file).

- Docker uses various filesystem tricks to layer filesystems on top of each other. This helps the caching of builds but can be slower than dealing with the host filesystem. Use host-level mounts when you need to access data quickly, and consider using `volumes` set as read-only to choose a volume driver that is fitting for your infrastructure.
- Docker creates a virtual network for all your containers to live behind. This can be great for keeping most of your services hidden behind a gateway, but it also adds slight network overhead. For most use cases, this overhead is negligible, but it can be mitigated by changing the network driver.
- GPUs and other host-level devices can be accessed using special runtime drivers for Docker. For example, `nvidia-docker` allows Docker environments to easily use connected NVIDIA GPUs. In general, devices can be made available with the `--device` runtime flag.

As always, it is important to profile your Docker containers to know what the issues are and whether there are some easy wins in terms of efficiency. The `docker stats` command provides a good high-level view to help understand the current runtime performance of your containers.

Advantages of Docker

So far it has seemed that Docker is simply adding a whole new host of issues to contend with in terms of performance. However, the gains to reproducibility and reliability of runtime environments far surpass any extra complexity.

Locally, having access to all our previously run Docker containers allows us to quickly rerun and retest previous versions of our code without having to worry about changes to the runtime environment, such as dependencies and system packages ([Example 10-11](#) shows a list of containers we can run with a simple `docker_run` command). This makes it incredibly easy to constantly be testing for performance regressions that otherwise would be difficult to reproduce.

Example 10-11. Docker tags to keep track of previous runtime environments

```
$ docker images -a
REPOSITORY          TAG      IMAGE ID
highperformance/diffusion2d    latest   ceabe8b555ab
highperformance/diffusion2d    numpy-memory2  ceabe8b555ab
highperformance/diffusion2d    numpy-memory1  66523a1a107d
highperformance/diffusion2d    python-memory  46381a8db9bd
highperformance/diffusion2d    python        4cac9773ca5e
```

Many more benefits come with the use of a [container registry](#), which allows the storage and sharing of Docker images with the simple `docker pull` and `docker push`

commands, in a similar way to `git`. This lets us put all our containers in a publicly available location, allowing team members to pull in changes or new versions and letting them immediately run the code.



This book is a great example of the benefits of sharing a Docker container for standardizing a runtime environment. To convert this book from `asciidoc`, the markup language it was written in, into PDF, a Docker container was shared between us so we could reliably and reproducibly build book artifacts. This standardization saved us countless hours that were spent in the first edition as one of us would have a build issue that the other couldn't reproduce or help debug.

Running `docker pull highperformance/diffusion2d:latest` is much easier than having to clone a repository and doing all the associated setup that may be necessary to run a project. This is particularly true for research code, which may have some very fragile system dependencies. Having all of this inside an easily pullable Docker container means all of these setup steps can be skipped and the code can be run easily. As a result, code can be shared more easily, and a coding team can work more effectively together.

Finally, in conjunction with **kubernetes** and other similar technologies, Dockerizing your code helps with actually running it with the resources it needs. Kubernetes allows you to create a cluster of nodes, each labeled with resources it may have, and to orchestrate running containers on the nodes. It will take care of making sure the correct number of instances are being run, and thanks to the Docker virtualization, the code will be run in the same environment that you saved it to. One of the biggest pains of working with a cluster is making sure that the cluster nodes have the correct runtime environment as your workstation, and using Docker virtualization completely resolves this.⁸

Wrap-Up

So far in the book, we've looked at profiling to understand slow parts of your code, compiling and using `numpy` to make your code run faster, and various approaches to multiple processes and computers. In addition, we surveyed container virtualization to manage code environments and help in cluster deployment. In the penultimate chapter, we'll look at ways of using less RAM through different data structures and probabilistic approaches. These lessons could help you keep all your data on one machine, avoiding the need to run a cluster.

⁸ A great tutorial to get started with Docker and Kubernetes can be found at <https://oreil.ly/l9jXD>.

Using Less RAM

Questions You'll Be Able to Answer After This Chapter

- Why should I use less RAM?
- Why are `numpy` and `array` better for storing lots of numbers?
- How can lots of text be efficiently stored in RAM?
- How could I count (approximately!) to 10^{76} using just 1 byte?
- What are Bloom filters, and why might I need them?

We rarely think about how much RAM we're using until we run out of it. If you run out while scaling your code, it can become a sudden blocker. Fitting more into a machine's RAM means fewer machines to manage, and it gives you a route to planning capacity for larger projects. Knowing why RAM gets eaten up and considering more efficient ways to use this scarce resource will help you deal with scaling issues. We'll use the Memory Profiler and IPython Memory Usage tools to measure the actual RAM usage, along with some tools that introspect objects to try to guess how much RAM they're using.

Another route to saving RAM is to use containers that utilize features in your data for compression. In this chapter, we'll look at a trie (ordered tree data structures) and a directed acyclic word graph (DAWG) that can compress a 1.2 GB set of strings down to just 30 MB with little change in performance. A third approach is to trade storage for accuracy. For this we'll look at approximate counting and approximate set membership, which use dramatically less RAM than their exact counterparts.

A consideration with RAM usage is the notion that “data has mass.” The more there is of it, the slower it moves around. If you can be parsimonious in your use of RAM, your data will probably get consumed faster, as it’ll move around buses faster and more of it will fit into constrained caches. If you need to store it in offline storage (e.g., a hard drive or a remote data cluster), it’ll move far more slowly to your machine. Try to choose appropriate data structures so all your data can fit onto one machine. We’ll use NumExpr to efficiently calculate with NumPy and Pandas with fewer data movements than the more direct method, which will save us time and make certain larger calculations feasible in a fixed amount of RAM.

Counting the amount of RAM used by Python objects is surprisingly tricky. We don’t necessarily know how an object is represented behind the scenes, and if we ask the operating system for a count of bytes used, it will tell us about the total amount allocated to the process. In both cases, we can’t see exactly how each individual Python object adds to the total.

As some objects and libraries don’t report their full internal allocation of bytes (or they wrap external libraries that do not report their allocation at all), this has to be a case of best-guessing. The approaches explored in this chapter can help us to decide on the best way to represent our data so we use less RAM overall.

We’ll also look at several lossy methods for storing strings in scikit-learn and counts in data structures. This works a little like a JPEG compressed image—we lose some information (and we can’t undo the operation to recover it), and we gain a lot of compression as a result. By using hashes on strings, we compress the time and memory usage for a natural language processing task in scikit-learn, and we can count huge numbers of events with only a small amount of RAM.

Objects for Primitives Are Expensive

It’s common to work with containers like the `list`, storing hundreds or thousands of items. As soon as you store a large number, RAM usage becomes an issue.

A `list` with 100 million items consumes approximately 760 MB of RAM, *if the items are the same object*. If we store 100 million *different* items (e.g., unique integers), we can expect to use gigabytes of RAM! Each unique object has a memory cost.

In [Example 11-1](#), we store many `0` integers in a `list`. If you stored 100 million references to any object (regardless of how large one instance of that object was), you’d still expect to see a memory cost of roughly 760 MB, as the `list` is storing references to (not copies of) the object. Refer back to [“Using memory_profiler to Diagnose Memory Usage” on page 46](#) for a reminder of how to use `memory_profiler`; here, we load it as a new magic function in IPython using `%load_ext memory_profiler`.

Example 11-1. Measuring memory usage of 100 million of the same integer in a list

```
In [1]: %load_ext memory_profiler # load the %memit magic function
In [2]: %memit [0] * int(1e8)
peak memory: 806.33 MiB, increment: 762.77 MiB
```

For our next example, we'll start with a fresh shell. As the results of the first call to `memit` in [Example 11-2](#) reveal, a fresh IPython shell consumes approximately 40 MB of RAM. Next, we can create a temporary list of 100 million *unique* numbers. In total, this consumes approximately 3.8 GB.



Memory can be cached in the running process, so it is always safer to exit and restart the Python shell when using `memit` for profiling.

After the `memit` command finishes, the temporary list is deallocated. The final call to `memit` shows that the memory usage drops to its previous level.

Example 11-2. Measuring memory usage of 100 million different integers in a list

```
# we use a new IPython shell so we have a clean memory
In [1]: %load_ext memory_profiler
In [2]: %memit # show how much RAM this process is consuming right now
peak memory: 43.39 MiB, increment: 0.11 MiB
In [3]: %memit [n for n in range(int(1e8))]
peak memory: 3850.29 MiB, increment: 3806.59 MiB
In [4]: %memit
peak memory: 44.79 MiB, increment: 0.00 MiB
```

A subsequent `memit` in [Example 11-3](#) to create a second 100-million-item list consumes approximately 3.8 GB again.

Example 11-3. Measuring memory usage again for 100 million different integers in a list

```
In [5]: %memit [n for n in range(int(1e8))]
peak memory: 3855.78 MiB, increment: 3810.96 MiB
```

Next, we'll see that we can use the `array` module to store 100 million integers far more cheaply.

The array Module Stores Many Primitive Objects Cheaply

The `array` module efficiently stores primitive types like integers, floats, and characters, but *not* complex numbers or classes. It creates a contiguous block of RAM to hold the underlying data.

In [Example 11-4](#), we allocate 100 million integers (8 bytes each) into a contiguous chunk of memory. In total, approximately 760 MB is consumed by the process. The difference between this approach and the previous list-of-unique-integers approach is $3100\text{MB} - 760\text{MB} == 2.3\text{GB}$. This is a huge savings in RAM.

Example 11-4. Building an array of 100 million integers with 760 MB of RAM

```
In [1]: %load_ext memory_profiler
In [2]: import array
In [3]: %memit array.array('l', range(int(1e8)))
peak memory: 837.88 MiB, increment: 761.39 MiB
In [4]: arr = array.array('l')
In [5]: arr.itemsize
Out[5]: 8
```

Note that the unique numbers in the `array` are *not* Python objects; they are bytes in the `array`. If we were to dereference any of them, a new Python `int` object would be constructed. If you're going to compute on them, no overall savings will occur, but if instead you're going to pass the array to an external process or use only some of the data, you should see a good savings in RAM compared to using a `list` of integers.



If you're working with a large array or matrix of numbers with Cython and you don't want an external dependency on `numpy`, be aware that you can store your data in an `array` and pass it into Cython for processing without any additional memory overhead.

The `array` module works with a limited set of datatypes with varying precisions (see [Example 11-5](#)). Choose the smallest precision that you need, so that you allocate just as much RAM as needed and no more. Be aware that the byte size is platform-dependent—the sizes here refer to a 32-bit platform (it states *minimum* size), whereas we're running the examples on a 64-bit laptop.

Example 11-5. The basic types provided by the array module

```
In [5]: array.array? # IPython magic, similar to help(array)
Init signature: array.array(self, /, *args, **kwargs)
Docstring:
array(typecode [, initializer]) -> array
```

Return a new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, string, or iterable over elements of the appropriate type.

Arrays represent basic values and behave very much like lists, except the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

Type code	C Type	Minimum size in bytes
'b'	signed integer	1
'B'	unsigned integer	1
'u'	Unicode character	2 (see note)
'h'	signed integer	2
'H'	unsigned integer	2
'i'	signed integer	2
'I'	unsigned integer	2
'l'	signed integer	4
'L'	unsigned integer	4
'q'	signed integer	8 (see note)
'Q'	unsigned integer	8 (see note)
'f'	floating point	4
'd'	floating point	8

NumPy has arrays that can hold a wider range of datatypes—you have more control over the number of bytes per item, and you can use complex numbers and datetime objects. A complex128 object takes 16 bytes per item: each item is a pair of 8-byte floating-point numbers. You can't store complex objects in a Python array, but they come for free with numpy. If you'd like a refresher on numpy, look back to [Chapter 6](#).

In [Example 11-6](#), you can see an additional feature of numpy arrays; you can query for the number of items, the size of each primitive, and the combined total storage of the underlying block of RAM. Note that this doesn't include the overhead of the Python object (typically, this is tiny in comparison to the data you store in the arrays).



Be wary of lazy allocation with zeros. In the following example the call to zeros costs “zero” RAM, whilst the call to ones costs 1.5 GB. Both calls will ultimately cost 1.5 GB, but the call to zeros will allocate the RAM only after it is used, so the cost is seen later.

Example 11-6. Storing more complex types in a numpy array

```
In [1]: %load_ext memory_profiler
In [2]: import numpy as np
# NOTE that zeros have lazy allocation so misreport the memory used!
In [3]: %memit arr=np.zeros(int(1e8), np.complex128)
peak memory: 58.37 MiB, increment: 0.00 MiB
In [4]: %memit arr=np.ones(int(1e8), np.complex128)
peak memory: 1584.41 MiB, increment: 1525.89 MiB
In [5]: f"{arr.size:,}"
Out[5]: '100,000,000'
In [6]: f"{arr.nbytes:,}"
Out[6]: '1,600,000,000'
In [7]: arr.nbytes/arr.size
Out[7]: 16.0
In [8]: arr.itemsize
Out[8]: 16
```

Using a regular `list` to store many numbers is much less efficient in RAM than using an `array` object. More memory allocations have to occur, which each take time; calculations also occur on larger objects, which will be less cache friendly, and more RAM is used overall, so less RAM is available to other programs.

However, if you do any work on the contents of the `array` in Python numex, the primitives are likely to be converted into temporary objects, negating their benefit. Using them as a data store when communicating with other processes is a great use case for the `array`.

`numpy` arrays are almost certainly a better choice if you are doing anything heavily numeric, as you get more datatype options and many specialized and fast functions. You might choose to avoid `numpy` if you want fewer dependencies for your project, though Cython works equally well with `array` and `numpy` arrays; Numba works with `numpy` arrays only.

Python provides a few other tools to understand memory usage, as we'll see in the following section.

Using Less RAM in NumPy with NumExpr

Large vectorized expressions in NumPy (which also happen behind the scenes in Pandas) can create intermediate large arrays during complex operations. These occur invisibly and may draw attention to themselves only when an out-of-memory error occurs. These calculations can also be slow, as large vectors will not be cache friendly—a cache can be megabytes or smaller, and large vectors of hundreds of megabytes or gigabytes of data will stop the cache from being used effectively. NumExpr is a tool that both speeds up and reduces the size of intermediate operations; we introduced it in “[numexpr: Making In-Place Operations Faster and Easier](#)” on page 140.

We've also introduced the Memory Profiler before, in “[Using memory_profiler to Diagnose Memory Usage](#)” on page 46. Here, we build on it with the [IPython Memory Usage tool](#), which reports line-by-line memory changes inside the IPython shell or in a Jupyter Notebook. Let's look at how these can be used to check that NumExpr is generating a result more efficiently.



Remember to install the optional NumExpr when using Pandas. If NumExpr is installed in Pandas, calls to `eval` will run more quickly—but note that Pandas does not tell you if you *haven't* installed NumExpr.

We'll use the cross entropy formula to calculate the error for a machine learning classification challenge. *Cross entropy* (or *Log Loss*) is a common metric for classification challenges; it penalizes large errors significantly more than small errors. Each row in a machine learning problem needs to be scored during the training and prediction phases:

$$-\log P(yt \mid yp) = - (yt \log(yp) + (1 - yt) \log(1 - yp))$$

We'll use random numbers in the range [0, 1] here to simulate the results of a machine learning system from a package like scikit-learn or TensorFlow. [Figure 11-1](#) shows the natural logarithm for the range [0, 1] on the right, and on the left it shows the result of calculating the cross entropy for any probability if the target is either 0 or 1.

If the target `yt` is 1, the first half of the formula is active and the second half goes to zero. If the target is 0, the second half of the formula is active and the first part goes to zero. This result is calculated for every row of data that needs to be scored and often for many iterations of the machine learning algorithm.

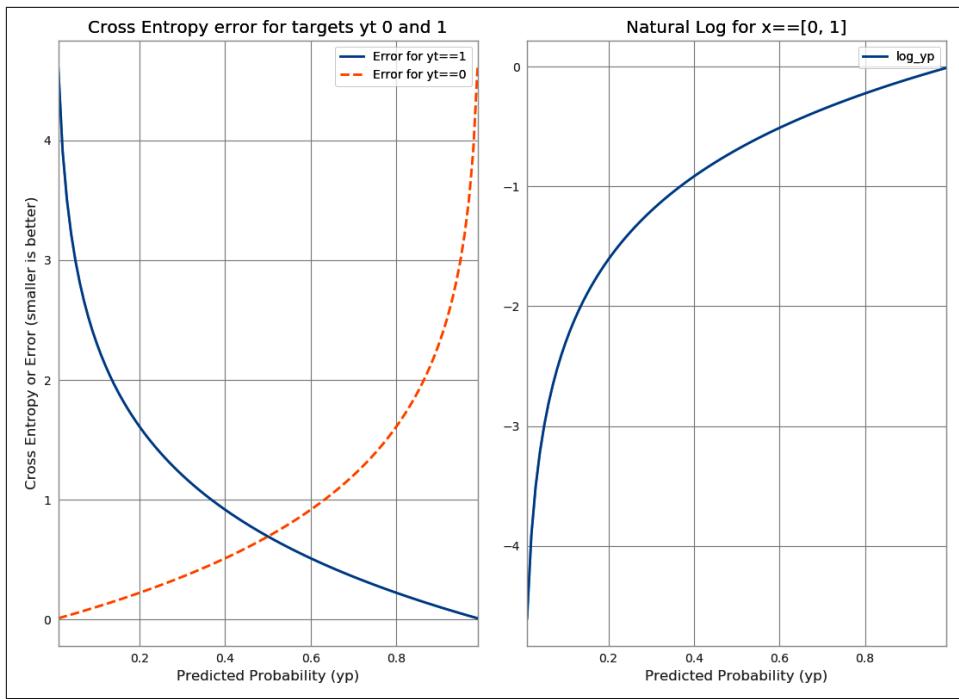


Figure 11-1. Cross entropy for yt (the “truth”) with values 0 and 1

In [Example 11-7](#), we generate 200 million random numbers in the range $[0, 1]$ as yp . yt is the desired truth—in this case, an array of 1s. In a real application, we’d see yp generated by a machine learning algorithm, and yt would be the ground truth mixing 0s and 1s for the target we’d be learning provided by the machine learning researcher.

Example 11-7. The hidden cost of temporaries with large NumPy arrays

```
In [1]: import ipython_memory_usage.ipython_memory_usage as imu; import numpy as np
In [2]: %ipython_memory_usage_start
Out[2]: 'memory profile enabled'
In [3]: nbr_items = 200_000_000
In [4]: yp = np.random.uniform(low=0.0000001, size=nbr_items)
In [4] used 1526.0508 MiB RAM in 2.18s, peaked 0.00 MiB above current,
      total RAM usage 1610.05 MiB
In [5]: yt = np.ones(shape=nbr_items)
In [5] used 1525.8516 MiB RAM in 0.44s, peaked 0.00 MiB above current,
      total RAM usage 3135.90 MiB

In [6]: answer = -(yt * np.log(yp) + ((1-yt) * (np.log(1-yp))))
In [6] used 1525.8594 MiB RAM in 18.63s, peaked 4565.70 MiB above current,
      total RAM usage 4661.76 MiB
```

```
In [7]: del answer
In [7] used -1525.8242 MiB RAM in 0.11s, peaked 0.00 MiB above current,
      total RAM usage 3135.93 MiB
```

Both `yp` and `yt` take 1.5 GB each, bringing the total RAM usage to just over 3.1 GB. The `answer` vector has the same dimension as the inputs and thus adds a further 1.5 GB. Note that the calculation peaks at 4.5 GB over the current RAM usage, so while we end with a 4.6 GB result, we had over 9 GB allocated during the calculation. The cross entropy calculation creates several temporaries (notably `1 - yt`, `np.log(1 - yp)`, and their multiplication). If you had an 8 GB machine, you'd have failed to calculate this result because of memory exhaustion.

In [Example 11-8](#), we see the same expression placed as a string inside `numexpr.evaluate`. It peaks at 0 GB above the current usage—it doesn't need any additional RAM in this case. Significantly, it also calculates much more quickly: the previous direct vector calculation in In[6] took 18 seconds, while here with NumExpr the same calculation takes 2.6 seconds.

NumExpr breaks the long vectors into shorter, cache-friendly chunks and processes each in series, so local chunks of results are calculated in a cache-friendly way. This explains both the requirement for no extra RAM and the increased speed.

Example 11-8. NumExpr breaks the vectorized calculations into cache-efficient chunks

```
In [8]: import numexpr
In [8] used 0.0430 MiB RAM in 0.12s, peaked 0.00 MiB above current,
      total RAM usage 3135.95 MiB
In [9]: answer = numexpr.evaluate("-(yt * log(yp) + ((1-yt) * (log(1-yp))))")
In [9] used 1525.8281 MiB RAM in 2.67s, peaked 0.00 MiB above current,
      total RAM usage 4661.78 MiB
```

We can see a similar benefit in Pandas in [Example 11-9](#). We construct a DataFrame with the same items as in the preceding example and invoke NumExpr by using `df.eval`. The Pandas machinery has to unpack the DataFrame for NumExpr, and more RAM is used overall; behind the scenes, NumExpr is still calculating the result in a cache-friendly manner. Note that here NumExpr was installed in addition to Pandas.

Example 11-9. Pandas eval uses NumExpr if it is available

```
In [2]: df = pd.DataFrame({'yp': np.random.uniform(low=0.0000001, size=nbr_items),
                        'yt': np.ones(nbr_items)})
In [3]: answer_eval = df.eval("-(yt * log(yp) + ((1-yt) * (log(1-yp))))")
In [3] used 3052.1953 MiB RAM in 5.26s, peaked 3045.77 MiB above current,
      total RAM usage 6185.45 MiB
```

Contrast the preceding example with [Example 11-10](#), where NumExpr has *not* been installed. The call to `df.eval` falls back on the Python interpreter—the same result is calculated but with a 34-second execution time (compared to 5.2 seconds before) and a much larger peak memory usage. You can test whether NumExpr is installed with `import numexpr`—if this fails, you’ll want to install it.

Example 11-10. Beware that Pandas without NumExpr makes a slow and costly call to eval!

```
In [2]: df = pd.DataFrame({'yp': np.random.uniform(low=0.000001, size=nbr_items),
                           'yt': np.ones(nbr_items)})
In [3]: answer_eval = df.eval("-(yt * log(yp) + ((1-yt) * (log(1-yp))))")
In [3] used 3052.5625 MiB RAM in 34.88s, peaked 7620.15 MiB above current,
      total RAM usage 6185.24 MiB
```

Complex vector operations on large arrays will run faster if you can use NumExpr. Pandas will not warn you that NumExpr hasn’t been installed, so we recommend that you add it as part of your setup if you use `eval`. The IPython Memory Usage tool will help you to diagnose where your RAM is being spent if you have large arrays to process; this can help you fit more into RAM on your current machine so you don’t have to start dividing your data and introducing greater engineering effort.

Understanding the RAM Used in a Collection

You may wonder if you can ask Python about the RAM that’s used by each object. Python’s `sys.getsizeof(obj)` call will tell us *something* about the memory used by an object (most but not all objects provide this). If you haven’t seen it before, be warned that it won’t give you the answer you’d expect for a container!

Let’s start by looking at some primitive types. An `int` in Python is a variable-sized object of arbitrary size, well above the range of an 8-byte C integer. The basic object costs 24 bytes in Python 3.7 when initialized with 0. More bytes are added as you count to larger numbers:

```
In [1]: sys.getsizeof(0)
Out[1]: 24
In [2]: sys.getsizeof(1)
Out[2]: 28
In [3]: sys.getsizeof((2**30)-1)
Out[3]: 28
In [4]: sys.getsizeof((2**30))
Out[4]: 32
```

Behind the scenes, sets of 4 bytes are added each time the size of the number you’re counting steps above the previous limit. This affects only the memory usage; you don’t see any difference externally.

We can do the same check for byte strings. An empty byte sequence costs 33 bytes, and each additional character adds 1 byte to the cost:

```
In [5]: sys.getsizeof(b"")
Out[5]: 33
In [6]: sys.getsizeof(b"a")
Out[6]: 34
In [7]: sys.getsizeof(b"ab")
Out[7]: 35
In [8]: sys.getsizeof(b"abc")
Out[8]: 36
```

When we use a list, we see different behavior. `getsizeof` isn't counting the cost of the contents of the list—just the cost of the list itself. An empty list costs 64 bytes, and each item in the list takes another 8 bytes on a 64-bit laptop:

```
# goes up in 8-byte steps rather than the 24+ we might expect!
In [9]: sys.getsizeof([])
Out[9]: 64
In [10]: sys.getsizeof([1])
Out[10]: 72
In [11]: sys.getsizeof([1, 2])
Out[11]: 80
```

This is more obvious if we use byte strings—we'd expect to see much larger costs than `getsizeof` is reporting:

```
In [12]: sys.getsizeof([b""])
Out[12]: 72
In [13]: sys.getsizeof([b"abcdefghijklm"])
Out[13]: 72
In [14]: sys.getsizeof([b"a", b"b"])
Out[14]: 80
```

`getsizeof` reports only some of the cost, and often just for the parent object. As noted previously, it also isn't always implemented, so it can have limited usefulness.

A better tool is `asizeof` in [pympler](#). This will walk a container's hierarchy and make a best guess about the size of each object it finds, adding the sizes to a total. Be warned that it is quite slow.

In addition to relying on guesses and assumptions, `asizeof` also cannot count memory allocated behind the scenes (such as a module that wraps a C library may not report the bytes allocated in the C library). It is best to use this as a guide. We prefer to use `memit`, as it gives us an accurate count of memory usage on the machine in question.

We can check the estimate it makes for a large list—here we'll use 10 million integers:

```
In [1]: from pympler.asizeof import asizeof
In [2]: asizeof([x for x in range(int(1e7))])
Out[2]: 401528048
In [3]: %memit [x for x in range(int(1e7))]
peak memory: 401.91 MiB, increment: 326.77 MiB
```

We can validate this estimate by using `memit` to see how the process grew. Both reports are approximate—`memit` takes snapshots of the RAM usage reported by the operating system while the statement is executing, and `asizeof` asks the objects about their size (which may not be reported correctly). We can conclude that 10 million integers in a list cost between 320 and 400 MB of RAM.

Generally, the `asizeof` process is slower than using `memit`, but `asizeof` can be useful when you're analyzing small objects. `memit` is probably more useful for real-world applications, as the actual memory usage of the process is measured rather than inferred.

Bytes Versus Unicode

One of the (many!) advantages of Python 3.x over Python 2.x is the switch to Unicode-by-default. Previously, we had a mix of single-byte strings and multibyte Unicode objects, which could cause a headache during data import and export. In Python 3.x, all strings are Unicode by default, and if you want to deal in bytes, you'll explicitly create a byte sequence.

Unicode objects have more efficient RAM usage in Python 3.7 than in Python 2.x. In [Example 11-11](#), we can see a 100-million-character sequence being built as a collection of bytes and as a Unicode object. The Unicode variant for common characters (here we're assuming UTF 8 as the system's default encoding) costs the same—a single-byte implementation is used for these common characters.

Example 11-11. Unicode objects can be as cheap as bytes in Python 3.x

```
In [1]: %load_ext memory_profiler
In [2]: type(b"b")
Out[2]: bytes
In [3]: %memit b"a" * int(1e8)
peak memory: 121.55 MiB, increment: 78.17 MiB
In [4]: type("u")
Out[4]: str
In [5]: %memit "u" * int(1e8)
peak memory: 122.43 MiB, increment: 78.49 MiB
In [6]: %memit "\u0393" * int(1e8)
peak memory: 316.40 MiB, increment: 176.17 MiB
```

The sigma character (Σ) is more expensive—it is represented in UTF 8 as 2 bytes. We gained the flexible Unicode representation from Python 3.3 thanks to [PEP 393](#). It works by observing the range of characters in the string and using a smaller number of bytes to represent the lower-order characters, if possible.

The UTF-8 encoding of a Unicode object uses 1 byte per ASCII character and more bytes for less frequently seen characters. If you’re not sure about Unicode encodings versus Unicode objects, go and watch [Net Batchelder’s “Pragmatic Unicode, or, How Do I Stop the Pain?”](#).

Efficiently Storing Lots of Text in RAM

A common problem with text is that it occupies a lot of RAM—but if we want to test if we have seen strings before or count their frequency, having them in RAM is more convenient than paging them to and from a disk. Storing the strings naively is expensive, but tries and directed acyclic word graphs (DAWGs) can be used to compress their representation and still allow fast operations.

These more advanced algorithms can save you a significant amount of RAM, which means that you might not need to expand to more servers. For production systems, the savings can be huge. In this section we’ll look at compressing a set of strings costing 1.2 GB down to 30 MB using a trie, with only a small change in performance.

For this example, we’ll use a text set built from a partial dump of Wikipedia. This set contains 11 million unique tokens from a portion of the English Wikipedia and takes up 120 MB on disk.

The tokens are split on whitespace from their original articles; they have variable length and contain Unicode characters and numbers. They look like this:

```
faddishness
'melanesians'
Kharálampos
PizzaInACup™
url="http://en.wikipedia.org/wiki?curid=363886"
VIIIa),
Superbagñères.
```

We’ll use this text sample to test how quickly we can build a data structure holding one instance of each unique word, and then we’ll see how quickly we can query for a known word (we’ll use the uncommon “Zwiebel,” from the painter Alfred Zwiebel). This lets us ask, “Have we seen Zwiebel before?” Token lookup is a common problem, and being able to do it quickly is important.



When you try these containers on your own problems, be aware that you will probably see different behaviors. Each container builds its internal structures in different ways; passing in different types of token is likely to affect the build time of the structure, and different lengths of token will affect the query time. Always test in a methodical way.

Trying These Approaches on 11 Million Tokens

Figure 11-2 shows the 11-million-token text file (120 MB raw data) stored using a number of containers that we'll discuss in this section. The x-axis shows RAM usage for each container, the y-axis tracks the query time, and the size of each point relates to the time taken to build the structure (larger means it took longer).

As we can see in this diagram, the `set` and `list` examples use a lot of RAM; the `list` example is both large *and* slow! The Marisa trie example is the most RAM-efficient for this dataset, while the DAWG runs twice as fast for a relatively small increase in RAM usage.

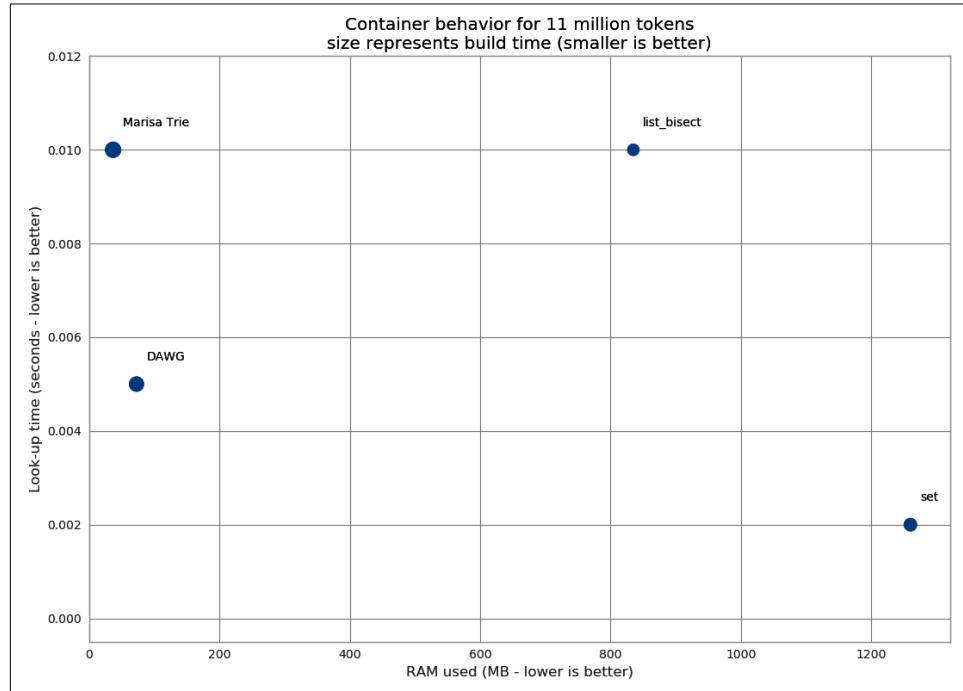


Figure 11-2. DAWG and tries versus built-in containers

The figure doesn't show the lookup time for the naive `list` without sort approach, which we'll introduce shortly, as it takes far too long. Do be aware that you must test

your problem with a variety of containers—each offers different trade-offs, such as construction time and API flexibility.

Next, we'll build up a process to test the behavior of each container.

list

Let's start with the simplest approach. We'll load our tokens into a `list` and then query it using an $O(n)$ linear search. You can't do this on the large example that we've already mentioned—the search takes far too long—so we'll demonstrate the technique with a much smaller (500,000 tokens) example.

In each of the following examples, we use a generator, `text_example.readers`, that extracts one Unicode token at a time from the input file. This means that the read process uses only a tiny amount of RAM:

```
print("RAM at start {:.0f}MiB".format(memory_profiler.memory_usage()[0]))
t1 = time.time()
words = [w for w in text_example.readers]
print("Loading {} words".format(len(words)))
t2 = time.time()
print("RAM after creating list {:.0f}MiB, took {:.0f}s" \
      .format(memory_profiler.memory_usage()[0], t2 - t1))
```

We're interested in how quickly we can query this `list`. Ideally, we want to find a container that will store our text and allow us to query it and modify it without penalty. To query it, we look for a known word a number of times by using `timeit`:

```
assert 'Zwiebel' in words
time_cost = sum(timeit.repeat(stmt='Zwiebel' in words',
                             setup="from __main__ import words",
                             number=1,
                             repeat=10000))
print("Summed time to look up word {:.4f}s".format(time_cost))
```

Our test script reports that approximately 34 MB was used to store the original 5 MB file as a list, and that the aggregate lookup time was 53 seconds:

```
$ python text_example_list.py
RAM at start 36.6MiB
Loading 499056 words
RAM after creating list 70.9MiB, took 1.0s
Summed time to look up word 53.5657s
```

Storing text in an unsorted `list` is obviously a poor idea; the $O(n)$ lookup time is expensive, as is the memory usage. This is the worst of all worlds! If we tried this method on the following larger dataset, we'd expect an aggregate lookup time of 25 minutes rather than a fraction of a second for the methods we discuss.

We can improve the lookup time by sorting the `list` and using a binary search via the `bisect module`; this gives us a sensible lower bound for future queries. In

Example 11-12, we time how long it takes to sort the list. Here, we switch to the larger 11-million-token set.

Example 11-12. Timing the sort operation to prepare for using bisect

```
print("RAM at start {:.0f}MiB".format(memory_profiler.memory_usage()[0]))
t1 = time.time()
words = [w for w in text_example.readers]
print("Loading {} words".format(len(words)))
t2 = time.time()
print("RAM after creating list {:.0f}MiB, took {:.0f}s" \
    .format(memory_profiler.memory_usage()[0], t2 - t1))
print("The list contains {} words".format(len(words)))
words.sort()
t3 = time.time()
print("Sorting list took {:.0f}s".format(t3 - t2))
```

Next, we do the same lookup as before, but with the addition of the `index` method, which uses `bisect`:

```
import bisect
...
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect.bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
...
time_cost = sum(timeit.repeat(stmt="index(words, 'Zwiebel')",
                             setup="from __main__ import words, index",
                             number=1,
                             repeat=10000))
```

In **Example 11-13**, we see that the RAM usage is much larger than before, as we're loading significantly more data. The sort takes a further 0.6 seconds, and the cumulative lookup time is 0.01 seconds.

Example 11-13. Timings for using bisect on a sorted list

```
$ python text_example_list_bisect.py
RAM at start 36.6MiB
Loading 11595290 words
RAM after creating list 871.9MiB, took 20.6s
The list contains 11595290 words
Sorting list took 0.6s
Summed time to look up word 0.0109s
```

We now have a sensible baseline for timing string lookups: RAM usage must get better than 871 MB, and the total lookup time should be better than 0.01 seconds.

set

Using the built-in `set` might seem to be the most obvious way to tackle our task. In [Example 11-14](#), the `set` stores each string in a hashed structure (see [Chapter 4](#) if you need a refresher). It is quick to check for membership, but each string must be stored separately, which is expensive on RAM.

Example 11-14. Using a set to store the data

```
words_set = set(text_example.readers)
```

As we can see in [Example 11-15](#), the `set` uses more RAM than the `list` by a further 250 MB; however, it gives us a very fast lookup time without requiring an additional `index` function or an intermediate sorting operation.

Example 11-15. Running the set example

```
$ python text_example_set.py
RAM at start 36.6MiB
RAM after creating set 1295.3MiB, took 24.0s
The set contains 11595290 words
Summed time to look up word 0.0023s
```

If RAM isn't at a premium, this might be the most sensible first approach.

We have now lost the *ordering* of the original data, though. If that's important to you, note that you could store the strings as keys in a dictionary, with each value being an index connected to the original read order. This way, you could ask the dictionary if the key is present and for its index.

More efficient tree structures

Let's introduce a set of algorithms that use RAM more efficiently to represent our strings.

[Figure 11-3](#) from [Wikimedia Commons](#) shows the difference in representation of four words—"tap", "taps", "top", and "tops"—between a trie and a DAWG.¹ With a `list` or a `set`, each of these words would be stored as a separate string. Both the DAWG and the trie share parts of the strings, so that less RAM is used.

¹ This example is taken from the Wikipedia article on the [deterministic acyclic finite state automaton](#) (DAFSA). DAFSA is another name for DAWG. The accompanying image is from Wikimedia Commons.

The main difference between these is that a trie shares just common prefixes, while a DAWG shares common prefixes and suffixes. In languages (like English) that have many common word prefixes and suffixes, this can save a lot of repetition.

Exact memory behavior will depend on your data's structure. Typically, a DAWG cannot assign a value to a key because of the multiple paths from the start to the end of the string, but the version shown here can accept a value mapping. Tries can also accept a value mapping. Some structures have to be constructed in a pass at the start, and others can be updated at any time.

A big strength of some of these structures is that they provide a *common prefix search*; that is, you can ask for all words that share the prefix you provide. With our list of four words, the result when searching for "ta" would be "tap" and "taps." Furthermore, since these are discovered through the graph structure, the retrieval of these results is very fast. If you're working with DNA, for example, compressing millions of short strings by using a trie can be an efficient way to reduce RAM usage.

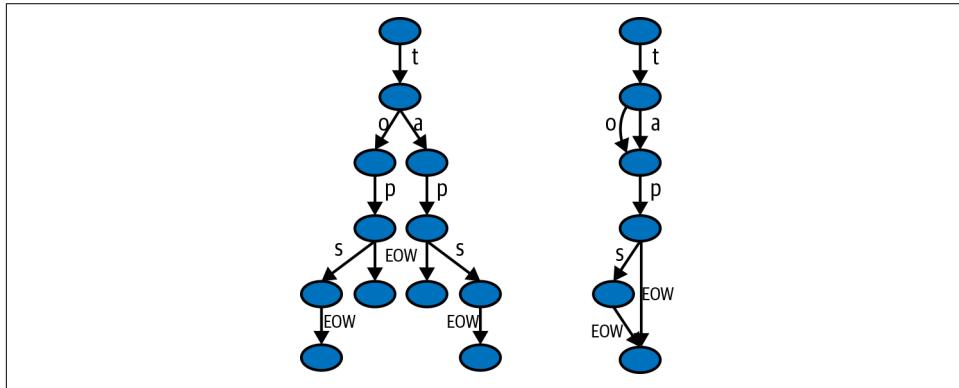


Figure 11-3. Trie and DAWG structures (image by Chkno [CC BY-SA 3.0])

In the following sections, we take a closer look at DAWGs, tries, and their usage.

Directed acyclic word graph

The [directed acyclic word graph](#) (MIT license) attempts to efficiently represent strings that share common prefixes and suffixes.

Note that at the time of writing, an [open Pull Request on GitHub](#) has to be applied to make this DAWG work with Python 3.7.

In [Example 11-16](#), you see the very simple setup for a DAWG. For this implementation, the DAWG cannot be modified after construction; it reads an iterator to construct itself once. The lack of post-construction updates might be a deal breaker for your use case. If so, you might need to look into using a trie instead. The DAWG

does support rich queries, including prefix lookups; it also allows persistence and supports storing integer indices as values along with byte and record values.

Example 11-16. Using a DAWG to store the data

```
import dawg
...
words_dawg = dawg.DAWG(text_example.readers)
```

As you can see in [Example 11-17](#), for the same set of strings it uses significantly less RAM than the earlier `set` example during the *construction* phase. More similar input text will cause stronger compression.

Example 11-17. Running the DAWG example

```
$ python text_example_dawg.py
RAM at start 38.1MiB
RAM after creating dawg 200.8MiB, took 31.6s
Summed time to look up word 0.0044s
```

More importantly, if we persist the DAWG to disk, as shown in [Example 11-18](#), and then load it back into a fresh Python instance, we see a dramatic reduction in RAM usage—the disk file and the memory usage after loading are both 70 MB; this is a significant savings compared to the 1.2 GB `set` variant we built earlier!

Example 11-18. Loading the DAWG that was built and saved in an earlier session is more RAM efficient

```
$ python text_example_dawg_load_only.py
RAM at start 38.4MiB
RAM after load 109.0MiB
Summed time to look up word 0.0051s
```

Given that you'll typically create a DAWG once and then load it many times, you'll benefit from the construction costs repeatedly after you've persisted the structure to disk.

Marisa trie

The [Marisa trie](#) (dual-licensed LGPL and BSD) is a static `trie` using Cython bindings to an external library. As it is static, it cannot be modified after construction. Like the DAWG, it supports storing integer indices as values, as well as byte values and record values.

A key can be used to look up a value, and vice versa. All keys sharing the same prefix can be found efficiently. The trie's contents can be persisted. [Example 11-19](#) illustrates using a Marisa trie to store our sample data.

Example 11-19. Using a Marisa trie to store the data

```
import marisa_trie
...
words_trie = marisa_trie.Trie(text_example.readers)
```

In [Example 11-20](#), we can see that lookup times are slower than those offered by the DAWG.

Example 11-20. Running the Marisa trie example

```
$ python text_example_trie.py
RAM at start 38.3MiB
RAM after creating trie 419.9MiB, took 35.1s
The trie contains 11595290 words
Summed time to look up word 0.0148s
```

The trie offers a further saving in memory on this dataset. While the lookups are a little slower in [Example 11-21](#), the disk and RAM usage are approximately 30 MB in the following snippet, if we save the trie to disk and then load it back into a fresh process; this is twice as good as what the DAWG achieved.

Example 11-21. Loading the Trie that was built and saved in an earlier session is more RAM efficient

```
$ python text_example_trie_load_only.py
RAM at start 38.5MiB
RAM after loading trie from disk 76.7MiB, took 0.0s
The trie contains 11595290 words
Summed time to look up word 0.0092s
```

The trade-off between storage sizes after construction and lookup times will need to be investigated for your application. You may find that using one of these “works just well enough,” so you might avoid benchmarking other options and simply move on to your next challenge. We suggest that the Marisa trie is your first choice in this case; it has more stars than the DAWG on GitHub.

Using tries (and DAWGs) in production systems

The trie and DAWG data structures offer good benefits, but you must still benchmark them on your problem rather than blindly adopting them. If you have overlapping sequences in your strings, you'll likely see a RAM improvement.

Tries and DAWGs are less well known, but they can provide strong benefits in production systems. We have an impressive success story in “[Large-Scale Social Media Analysis at Smesh \(2014\)](#)” on page 422. Jamie Matthews at DabApps (a Python software house based in the United Kingdom) also has a story about the use of tries in client systems to enable more efficient and cheaper deployments for customers:

At DabApps, we often try to tackle complex technical architecture problems by dividing them into small, self-contained components, usually communicating over the network using HTTP. This approach (referred to as a *service-oriented* or *microservice* architecture) has all sorts of benefits, including the possibility of reusing or sharing the functionality of a single component between multiple projects.

One such task that is often a requirement in our consumer-facing client projects is postcode geocoding. This is the task of converting a full UK postcode (for example: BN1 1AG) into a latitude and longitude coordinate pair, to enable the application to perform geospatial calculations such as distance measurement.

At its most basic, a geocoding database is a simple mapping between strings and can conceptually be represented as a dictionary. The dictionary keys are the postcodes, stored in a normalized form (BN11AG), and the values are a representation of the coordinates (we used a geohash encoding, but for simplicity imagine a comma-separated pair such as 50.822921,-0.142871).

The UK has approximately 1.7 million postcodes. Naively loading the full dataset into a Python dictionary, as described previously, uses several hundred megabytes of memory. Persisting this data structure to disk using Python’s native Pickle format requires an unacceptably large amount of storage space. We knew we could do better.

We experimented with several different in-memory and on-disk storage and serialization formats, including storing the data externally in databases such as Redis and LevelDB, and compressing the key/value pairs. Eventually, we hit on the idea of using a trie. Tries are extremely efficient at representing large numbers of strings in memory, and the available open source libraries (we chose “marisa-trie”) make them very simple to use.

The resulting application, including a tiny web API built with the Flask framework, uses only 30 MB of memory to represent the entire UK postcode database, and can comfortably handle a high volume of postcode lookup requests. The code is simple; the service is very lightweight and painless to deploy and run on a free hosting platform such as Heroku, with no external requirements or dependencies on databases. Our implementation is open source, available at <https://github.com/j4mie/postcodeserver>.

—Jamie Matthews, *technical director of DabApps.com (UK)*

DAWG and tries are powerful data structures that can help you save RAM and time in exchange for a little additional effort in preparation. These data structures will be unfamiliar to many developers, so consider separating this code into a module that is reasonably isolated from the rest of your code to simplify maintenance.

Modeling More Text with Scikit-Learn’s FeatureHasher

Scikit-learn is Python’s best-known machine learning framework, and it has excellent support for text-based natural language processing (NLP) challenges. Here, we’ll look at classifying public posts from Usenet archives to one of 20 prespecified categories; this is similar to the two-category spam classification process that cleans up our email inboxes.

One difficulty with text processing is that the vocabulary under analysis quickly explodes. The English language uses many nouns (e.g., the names of people and places, medical labels, and religious terms) and verbs (the “doing words” that often end in “-ing,” like “running,” “taking,” “making,” and “talking”) and their conjugations (turning the verb “talk” into “talked,” “talking,” “talks”), along with all the other rich forms of language. Punctuation and capitalization add an extra nuance to the representation of words.

A powerful and simple technique for classifying text is to break the original text into *n-grams*, often unigrams, bigrams, and trigrams (also known as 1-grams, 2-grams, and 3-grams). A sentence like “there is a cat and a dog” can be turned into unigrams (“there,” “is,” “a,” and so on), bigrams (“there is,” “is a,” “a cat,” etc.), and trigrams (“there is a,” “is a cat,” “a cat and,” ...).

There are 7 unigrams, 6 bigrams, and 5 trigrams for this sentence; in total this sentence can be represented in this form by a vocabulary of 6 unique unigrams (since the term “a” is used twice), 6 unique bigrams, and 5 unique trigrams, making 17 descriptive items in total. As you can see, the n-gram vocabulary used to represent a sentence quickly grows; some terms are very common, and some are very rare.

There are techniques to control the explosion of a vocabulary, such as eliminating stop-words (removing the most common and often uninformative terms, like “a,” “the,” and “of”), lowercasing everything, and ignoring less frequent types of terms (such as punctuation, numbers, and brackets). If you practice natural language processing, you’ll quickly come across these approaches.

Introducing DictVectorizer and FeatureHasher

Before we look at the Usenet classification task, let’s look at two of scikit-learn’s feature processing tools that help with NLP challenges. The first is `DictVectorizer`, which takes a dictionary of terms and their frequencies and converts them into a variable-width sparse matrix (we will discuss sparse matrices in “[SciPy’s Sparse Matrices](#)” on page 366). The second is `FeatureHasher`, which converts the same dictionary of terms and frequencies into a fixed-width sparse matrix.

Example 11-22 shows two sentences—“there is a cat” and “there is a cat and a dog”—in which terms are shared between the sentences and the term “a” is used twice in

one of the sentences. `DictVectorizer` is given the sentences in the call to `fit`; in a first pass it builds a list of words into an internal `vocabulary_`, and in a second pass it builds up a sparse matrix containing a reference to each term and its count.

Doing two passes takes longer than the one pass of `FeatureHasher`, and storing a vocabulary costs additional RAM. Building a vocabulary is often a serial process; by avoiding this stage, the feature hashing can potentially operate in parallel for additional speed.

Example 11-22. Lossless text representation with DictVectorizer

```
In [2]: from sklearn.feature_extraction import DictVectorizer
...
...: dv = DictVectorizer()
...: # frequency counts for ["there is a cat", "there is a cat and a dog"]
...: token_dict = [{'there': 1, 'is': 1, 'a': 1, 'cat': 1},
...:                 {'there': 1, 'is': 1, 'a': 2, 'cat': 1, 'and': 1, 'dog': 1}]
In [3]: dv.fit(token_dict)
...
...: print("Vocabulary:")
...: pprint(dv.vocabulary_)
Vocabulary:
{'a': 0, 'and': 1, 'cat': 2, 'dog': 3, 'is': 4, 'there': 5}
In [4]: X = dv.transform(token_dict)
```

To make the output a little clearer, see the Pandas DataFrame view of the matrix `X` in [Figure 11-4](#), where the columns are set to the vocabulary. Note that here we've made a *dense* representation of the matrix—we have 2 rows and 6 columns, and each of the 12 cells contains a number. In the sparse form, we store only the 10 counts that are present and we do not store anything for the 2 items that are missing. With a larger corpus, the larger storage required by a dense representation, containing mostly 0s, quickly becomes prohibitive. For NLP the sparse representation is standard.

a	and	cat	dog	is	there
0	1	0	1	0	1
1	2	1	1	1	1

Figure 11-4. Transformed output of DictVectorizer shown in a Pandas DataFrame

One feature of `DictVectorizer` is that we can give it a matrix and reverse the process. In [Example 11-23](#), we use the vocabulary to recover the original frequency representation. Note that this does *not* recover the original sentence; there's more than one way to interpret the ordering of words in the first example (both “there is a cat” and “a cat is there” are valid interpretations). If we used bigrams, we'd start to introduce a constraint on the ordering of words.

Example 11-23. Reversing the output of matrix X to the original dictionary representation

```
In [5]: print("Reversing the transform:")
...: pprint(dv.inverse_transform(X))

Reversing the transform:
[{'a': 1, 'cat': 1, 'is': 1, 'there': 1},
 {'a': 2, 'and': 1, 'cat': 1, 'dog': 1, 'is': 1, 'there': 1}]
```

`FeatureHasher` takes the same input and generates a similar output but with one key difference: it does not store a vocabulary and instead employs a hashing algorithm to assign token frequencies to columns.

We've already looked at hash functions in [“How Do Dictionaries and Sets Work?” on page 83](#). A hash converts a unique item (in this case a text token) into a number, where multiple unique items might map to the same hashed value, in which case we get a collision. Good hash functions cause few collisions. Collisions are inevitable if we're hashing many unique items to a smaller representation. One feature of a hash function is that it can't easily be reversed, so we can't take a hashed value and convert it back to the original token.

In [Example 11-24](#), we ask for a fixed-width 10-column matrix—the default is a fixed-width matrix of 1 million elements, but we'll use a tiny matrix here to show a collision. The default 1-million-element width is a sensible default for many applications.

The hashing process uses the fast MurmurHash3 algorithm, which transforms each token into a number; this is then converted into the range we specify. Larger ranges have few collisions; a small range like our range of 10 will have many collisions. Since every token has to be mapped to one of only 10 columns, we'll get many collisions if we add a lot of sentences.

The output `X` has 2 rows and 10 columns; each token maps to one column, and we don't immediately know which column represents each word since hashing functions are one-way, so we can't map the output back to the input. In this case we can deduce, using `extra_token_dict`, that the tokens `there` and `is` both map to column 8, so we get nine 0s and one count of 2 in column 8.

Example 11-24. Using a 10-column FeatureHasher to show a hash collision

```
In [6]: from sklearn.feature_extraction import FeatureHasher
...:
...: fh = FeatureHasher(n_features=10, alternate_sign=False)
...: fh.fit(token_dict)
...: X = fh.transform(token_dict)
...: pprint(X.toarray().astype(np.int_))
...:
array([[1, 0, 0, 0, 0, 0, 0, 2, 0, 1],
       [2, 0, 0, 1, 0, 1, 0, 2, 0, 1]])

In [7]: extra_token_dict = [{'there': 1, 'is': 1}, ]
...: X = fh.transform(extra_token_dict)
...: print(X.toarray().astype(np.int_))
...:
[[0 0 0 0 0 0 0 2 0 0]]
```

Despite the occurrence of collisions, more than enough signal is often retained in this representation (assuming the default number of columns is used) to enable similar quality machine learning results with FeatureHasher compared to DictVectorizer.

Comparing DictVectorizer and FeatureHasher on a Real Problem

If we take the full 20 Newsgroups dataset, we have 20 categories, with approximately 18,000 emails spread across the categories. While some categories such as “sci.med” are relatively unique, others like “comp.os.ms-windows.misc” and “comp.windows.x” will contain emails that share similar terms. The machine learning task is to correctly identify the correct newsgroup from the 20 options for each item in the test set. The test set has approximately 4,000 emails; the training set used to learn the mapping of terms to the matching category has approximately 14,000 emails.

Note that this example does *not* deal with some of the necessities of a realistic training challenge. We haven’t stripped newsgroup metadata, which can be used to overfit on this challenge; rather than generalize just from the text of the emails, some extraneous metadata artificially boosts the scores. We have randomly shuffled the emails. Here, we’re not trying to achieve a single excellent machine learning result; instead, we’re demonstrating that a lossy hashed representation can be equivalent to a non-lossy and more memory-hungry variant.

In [Example 11-25](#), we take 18,846 documents and build up a training and test set representation using both DictVectorizer and FeatureHasher with unigrams, bigrams, and trigrams. The DictVectorizer sparse array has shape (14,134, 4,335,793) for the training set, where our 14,134 emails are represented using 4 million tokens. Building the vocabulary and transforming the training data takes 42 seconds.

Contrast this with the `FeatureHasher`, which has a fixed 1-million-element-wide hashed representation, and where the transformation takes 21 seconds. Note that in both cases, roughly 9.8 million nonzero items are stored in the sparse matrices, so they're storing similar quantities of information. The hashed version stores approximately 10,000 fewer items because of collisions.

If we'd used a dense matrix, we'd have 14 thousand rows by 10 million columns, making 140,000,000,000 cells of 8 bytes each—significantly more RAM than is typically available in any current machine. Only a tiny fraction of this matrix would be nonzero. Sparse matrices avoid this RAM consumption.

Example 11-25. Comparing `DictVectorizer` and `FeatureHasher` on a real machine learning problem

```
Loading 20 newsgroups training data
18846 documents - 35.855MB

DictVectorizer on frequency dicts
DictVectorizer has shape (14134, 4335793) with 78,872,376 bytes
and 9,859,047 non-zero items in 42.15 seconds
Vocabulary has 4,335,793 tokens
LogisticRegression score 0.89 in 1179.33 seconds

FeatureHasher on frequency dicts
FeatureHasher has shape (14134, 1048576) with 78,787,936 bytes
and 9,848,492 non-zero items in 21.59 seconds
LogisticRegression score 0.89 in 903.35 seconds
```

Critically, the `LogisticRegression` classifier used on `DictVectorizer` takes 30% longer to train with 4 million columns compared to the 1 million columns used by the `FeatureHasher`. Both show a score of 0.89, so for this challenge the results are reasonably equivalent.

Using `FeatureHasher`, we've achieved the same score on the test set, built our training matrix faster, and avoided building and storing a vocabulary, and we've trained faster than using the more common `DictVectorizer` approach. In exchange, we've lost the ability to transform a hashed representation back into the original features for debugging and explanation, and since we often want the ability to diagnose *why* a decision was made, this might be too costly a trade for you to make.

SciPy's Sparse Matrices

In “[Introducing `DictVectorizer` and `FeatureHasher`](#)” on page 362, we created a large feature representation using `DictVectorizer`, which uses a sparse matrix in the background. These sparse matrices can be used for general computation as well and are extremely useful when working with sparse data.

A *sparse matrix* is a matrix in which most matrix elements are 0. For these sorts of matrices, there are many ways to encode the nonzero values and then simply say “all other values are zero.” In addition to these memory savings, many algorithms have special ways of dealing with sparse matrices that give additional computational benefits:

```
>>> from scipy import sparse
>>> A_sparse = sparse.random(2048, 2048, 0.05).tocsr()
>>> A_sparse
<2048x2048 sparse matrix of type '<class 'numpy.float64'>'>
    with 209715 stored elements in Compressed Sparse Row format>
>>> %timeit A_sparse * A_sparse
150 ms ± 1.71 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> A_dense = A_sparse.todense()
>>> type(A_dense)
numpy.matrix
>>> %timeit A_dense * A_dense
571 ms ± 14.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The simplest implementation of this is for COO matrices in SciPy, where for each non-zero element we store the value in addition to the location of the value. This means that for each nonzero value, we store three numbers in total. As long as our matrix has at least 66% zero entries, we are reducing the amount of memory needed to represent the data as a sparse matrix as opposed to a standard numpy array. However, COO matrices are generally used only to *construct* sparse matrices and not for doing actual computation (for that, CSR/CSC is preferred).

We can see in [Figure 11-5](#) that for low densities, sparse matrices are much faster than their dense counterparts. On top of this, they also use much less memory.

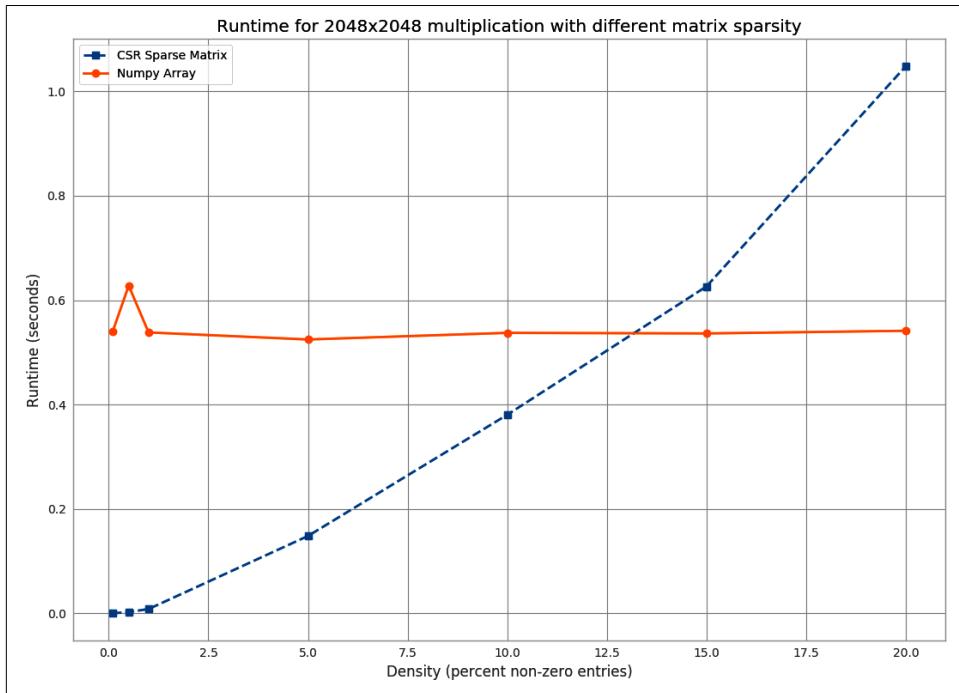


Figure 11-5. Sparse versus dense matrix multiplication

In [Figure 11-6](#), the dense matrix is always using 32.7 MB of memory ($2048 \times 2048 \times 64). However, a sparse matrix of 20% density uses only 10 MB, representing a 70% savings! As the density of the sparse matrix goes up, numpy quickly dominates in terms of speed because of the benefits of vectorization and better cache performance.$

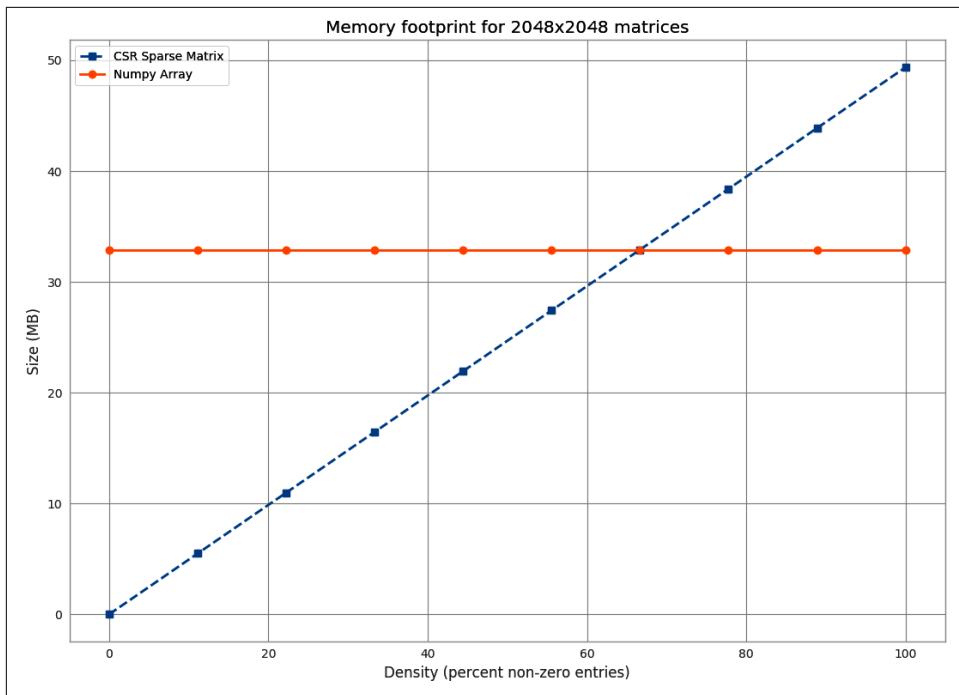


Figure 11-6. Sparse versus dense memory footprint

This extreme reduction in memory use is partly why the speeds are so much better. In addition to running only the multiplication operation for elements that are non-zero (thus reducing the number of operations needed), we also don't need to allocate such a large amount of space to save our result in. This is the push and pull of speed-ups with sparse arrays—it is a balance between losing the use of efficient caching and vectorization versus not having to do a lot of the calculations associated with the zero values of the matrix.

One operation that sparse matrices are particularly good at is cosine similarity. In fact, when creating a `DictVectorizer`, as we did in “[Introducing DictVectorizer and FeatureHasher](#)” on page 362, it’s common to use cosine similarity to see how similar two pieces of text are. In general for these item-to-item comparisons (where the value of a particular matrix element is compared to another matrix element), sparse matrices do quite well. Since the calls to `numpy` are the same whether we are using a normal matrix or a sparse matrix, we can benchmark the benefits of using a sparse matrix without changing the code of the algorithm.

While this is impressive, there are severe limitations. The amount of support for sparse matrices is quite low, and unless you are running special sparse algorithms or doing only basic operations, you’ll probably hit a wall in terms of support. In

addition, SciPy’s `sparse` module offers multiple implementations of sparse matrices, all of which have different benefits and drawbacks. Understanding which is the best one to use and when to use it demands some expert knowledge and often leads to conflicting requirements. As a result, sparse matrices probably aren’t something you’ll be using often, but when they are the correct tool, they are invaluable.

Tips for Using Less RAM

Generally, if you can avoid putting it into RAM, do. Everything you load costs you RAM. You might be able to load just a part of your data, for example, using a [memory-mapped file](#); alternatively, you might be able to use generators to load only the part of the data that you need for partial computations rather than loading it all at once.

If you are working with numeric data, you’ll almost certainly want to switch to using `numpy` arrays—the package offers many fast algorithms that work directly on the underlying primitive objects. The RAM savings compared to using lists of numbers can be huge, and the time savings can be similarly amazing. Furthermore, if you are dealing with very sparse arrays, using SciPy’s sparse array functionality can save incredible amounts of memory, albeit with a reduced feature set as compared to normal `numpy` arrays.

If you’re working with strings, stick to `str` rather than `bytes` unless you have strong reasons to work at the byte level. Dealing with a myriad set of text encodings is painful by hand, and UTF-8 (or other Unicode formats) tends to make these problems disappear. If you’re storing many Unicode objects in a static structure, you probably want to investigate the DAWG and trie structures that we’ve just discussed.

If you’re working with lots of bit strings, investigate `numpy` and the [bitarray](#) package; both have efficient representations of bits packed into bytes. You might also benefit from looking at Redis, which offers efficient storage of bit patterns.

The PyPy project is experimenting with more efficient representations of homogeneous data structures, so long lists of the same primitive type (e.g., integers) might cost much less in PyPy than the equivalent structures in CPython. The [MicroPython](#) project will be interesting to anyone working with embedded systems: this tiny-memory-footprint implementation of Python is aiming for Python 3 compatibility.

It goes (almost!) without saying that you know you have to benchmark when you’re trying to optimize on RAM usage, and that it pays handsomely to have a unit test suite in place before you make algorithmic changes.

Having reviewed ways of compressing strings and storing numbers efficiently, we’ll now look at trading accuracy for storage space.

Probabilistic Data Structures

Probabilistic data structures allow you to make trade-offs in accuracy for immense decreases in memory usage. In addition, the number of operations you can do on them is much more restricted than with a `set` or a trie. For example, with a single HyperLogLog++ structure using 2.56 KB, you can count the number of unique items up to approximately 7,900,000,000 items with 1.625% error.

This means that if we're trying to count the number of unique license plate numbers for cars, and our HyperLogLog++ counter said there were 654,192,028, we would be confident that the actual number is between 643,561,407 and 664,822,648. Furthermore, if this accuracy isn't sufficient, you can simply add more memory to the structure and it will perform better. Giving it 40.96 KB of resources will decrease the error from 1.625% to 0.4%. However, storing this data in a `set` would take 3.925 GB, even assuming no overhead!

On the other hand, the HyperLogLog++ structure would only be able to count a `set` of license plates and merge with another `set`. So, for example, we could have one structure for every state, find how many unique license plates are in each of those states, and then merge them all to get a count for the whole country. If we were given a license plate, we couldn't tell you with very good accuracy whether we've seen it before, and we couldn't give you a sample of license plates we have already seen.

Probabilistic data structures are fantastic when you have taken the time to understand the problem and need to put something into production that can answer a very small set of questions about a very large set of data. Each structure has different questions it can answer at different accuracies, so finding the right one is just a matter of understanding your requirements.



Much of this section goes into a deep dive into the mechanisms that power many of the popular probabilistic data structures. This is useful, because once you understand the mechanisms, you can use parts of them in algorithms you are designing. If you are just beginning with probabilistic data structures, it may be useful to first look at the real-world example (“Real-World Example” on [page 389](#)) before diving into the internals.

In almost all cases, probabilistic data structures work by finding an alternative representation for the data that is more compact and contains the relevant information for answering a certain set of questions. This can be thought of as a type of lossy compression, where we may lose some aspects of the data but we retain the necessary components. Since we are allowing the loss of data that isn't necessarily relevant for the particular set of questions we care about, this sort of lossy compression can be much more efficient than the lossless compression we looked at before with tries. It is

because of this that the choice of which probabilistic data structure you will use is quite important—you want to pick one that retains the right information for your use case!

Before we dive in, it should be made clear that all the “error rates” here are defined in terms of *standard deviations*. This term comes from describing Gaussian distributions and says how spread out the function is around a center value. When the standard deviation grows, so do the number of values further away from the center point. Error rates for probabilistic data structures are framed this way because all the analyses around them are probabilistic. So, for example, when we say that the HyperLogLog++ algorithm has an error of $err = \frac{1.04}{\sqrt{n}}$, we mean that 68% of the time the error will be smaller than err , 95% of the time it will be smaller than $2 \times err$, and 99.7% of the time it will be smaller than $3 \times err$.²

Very Approximate Counting with a 1-Byte Morris Counter

We’ll introduce the topic of probabilistic counting with one of the earliest probabilistic counters, the Morris counter (by Robert Morris of the NSA and Bell Labs). Applications include counting millions of objects in a restricted-RAM environment (e.g., on an embedded computer), understanding large data streams, and working on problems in AI like image and speech recognition.

The Morris counter keeps track of an exponent and models the counted state as $2^{exponent}$ (rather than a correct count)—it provides an *order of magnitude* estimate. This estimate is updated using a probabilistic rule.

We start with the exponent set to 0. If we ask for the *value* of the counter, we’ll be given `pow(2, exponent)=1` (the keen reader will note that this is off by one—we did say this was an *approximate* counter!). If we ask the counter to increment itself, it will generate a random number (using the uniform distribution), and it will test if `random.uniform(0, 1) <= 1/pow(2, exponent)`, which will always be true (`pow(2, 0) == 1`). The counter increments, and the exponent is set to 1.

The second time we ask the counter to increment itself, it will test if `random.uniform(0, 1) <= 1/pow(2, 1)`. This will be true 50% of the time. If the test passes, the exponent is incremented. If not, the exponent is not incremented for this increment request.

Table 11-1 shows the likelihoods of an increment occurring for each of the first exponents.

² These numbers come from the 68-95-99.7 rule of Gaussian distributions. More information can be found in the [Wikipedia entry](#).

Table 11-1. Morris counter details

Exponent	<code>pow(2,exponent)</code>	<code>P(increment)</code>
0	1	1
1	2	0.5
2	4	0.25
3	8	0.125
4	16	0.0625
...
254	2.894802e+76	3.454467e-77

The maximum we could approximately count where we use a single unsigned byte for the exponent is `math.pow(2,255) == 5e76`. The error relative to the actual count will be fairly large as the counts increase, but the RAM savings is tremendous, as we use only 1 byte rather than the 32 unsigned bytes we'd otherwise have to use. Example 11-26 shows a simple implementation of the Morris counter.

Example 11-26. Simple Morris counter implementation

```
"""Approximate Morris counter supporting many counters"""
import math
import random
import array

SMALLEST_UNSIGNED_INTEGER = 'B' # unsigned char, typically 1 byte

class MorrisCounter(object):
    """Approximate counter, stores exponent and counts approximately 2^exponent

    https://en.wikipedia.org/wiki/Approximate_counting_algorithm"""
    def __init__(self, type_code=SMALLEST_UNSIGNED_INTEGER, nbr_counters=1):
        self.exponents = array.array(type_code, [0] * nbr_counters)

    def __len__(self):
        return len(self.exponents)

    def add_counter(self):
        """Add a new zeroed counter"""
        self.exponents.append(0)

    def get(self, counter=0):
        """Calculate approximate value represented by counter"""
        return math.pow(2, self.exponents[counter])

    def add(self, counter=0):
        """Probabilistically add 1 to counter"""
        value = self.get(counter)
        if random.random() < value:
            self.exponents[counter] += 1
```

```

probability = 1.0 / value
if random.uniform(0, 1) < probability:
    self.exponents[counter] += 1

if __name__ == "__main__":
    mc = MorrisCounter()
    print("MorrisCounter has {} counters".format(len(mc)))
    for n in range(10):
        print("Iteration {}d, MorrisCounter has: {}d".format(n, mc.get()))
        mc.add()

    for n in range(990):
        mc.add()
    print("Iteration 1000, MorrisCounter has: {}d".format(mc.get()))

```

Using this implementation, we can see in [Example 11-27](#) that the first request to increment the counter succeeds, the second succeeds, and the third doesn't.³

Example 11-27. Morris counter library example

```

>>> mc = MorrisCounter()
>>> mc.get()
1.0

>>> mc.add()
>>> mc.get()
2.0

>>> mc.add()
>>> mc.get()
4.0

>>> mc.add()
>>> mc.get()
4.0

```

In [Figure 11-7](#), the thick black line shows a normal integer incrementing on each iteration. On a 64-bit computer, this is an 8-byte integer. The evolution of three 1-byte Morris counters is shown as dotted lines; the y-axis shows their values, which approximately represent the true count for each iteration. Three counters are shown to give you an idea about their different trajectories and the overall trend; the three counters are entirely independent of one another.

³ A more fully fleshed-out implementation that uses an array of bytes to make many counters is available at https://github.com/ianozsva/morris_counter.

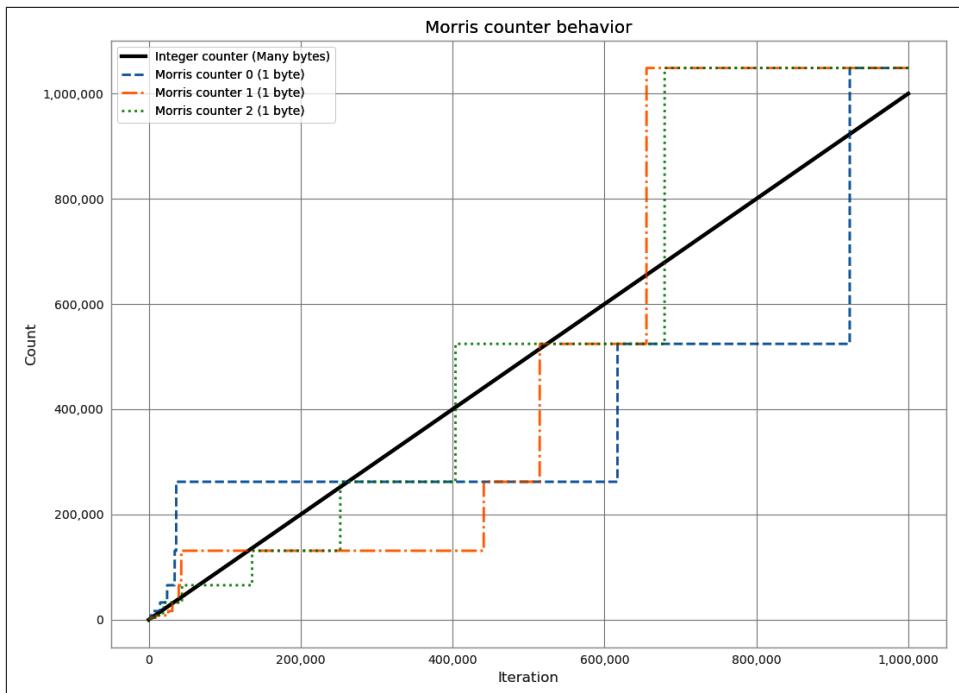


Figure 11-7. Three 1-byte Morris counters versus an 8-byte integer

This diagram gives you some idea about the error to expect when using a Morris counter. Further details about the error behavior are available [online](#).

K-Minimum Values

In the Morris counter, we lose any sort of information about the items we insert. That is to say, the counter's internal state is the same whether we do `.add("micha")` or `.add("ian")`. This extra information is useful and, if used properly, could help us have our counters count only unique items. In this way, calling `.add("micha")` thousands of times would increase the counter only once.

To implement this behavior, we will exploit properties of hashing functions (see “[Hash Functions and Entropy](#)” on page 88 for a more in-depth discussion of hash functions). The main property we would like to take advantage of is the fact that the hash function takes input and *uniformly* distributes it. For example, let’s assume we have a hash function that takes in a string and outputs a number between 0 and 1. For that function to be uniform means that when we feed it in a string, we are equally likely to get a value of 0.5 as a value of 0.2 or any other value. This also means that if we feed it in many string values, we would expect the values to be relatively evenly spaced. Remember, this is a probabilistic argument: the values won’t always be evenly

spaced, but if we have many strings and try this experiment many times, they will tend to be evenly spaced.

Suppose we took 100 items and stored the hashes of those values (the hashes being numbers from 0 to 1). Knowing the spacing is even means that instead of saying, “We have 100 items,” we could say, “We have a distance of 0.01 between every item.” This is where the K-Minimum Values algorithm finally comes in⁴--if we keep the k smallest unique hash values we have seen, we can approximate the overall spacing between hash values and infer the total number of items. In [Figure 11-8](#), we can see the state of a K-Minimum Values structure (also called a KMV) as more and more items are added. At first, since we don’t have many hash values, the largest hash we have kept is quite large. As we add more and more, the largest of the k hash values we have kept gets smaller and smaller. Using this method, we can get error rates of $O\left(\sqrt{\frac{2}{\pi(k-2)}}\right)$.

The larger k is, the more we can account for the hashing function we are using not being completely uniform for our particular input and for unfortunate hash values. An example of unfortunate hash values would be hashing ['A', 'B', 'C'] and getting the values [0.01, 0.02, 0.03]. If we start hashing more and more values, it is less and less probable that they will clump up.

Furthermore, since we are keeping only the smallest *unique* hash values, the data structure considers only unique inputs. We can see this easily because if we are in a state where we store only the smallest three hashes and currently [0.1, 0.2, 0.3] are the smallest hash values, then if we add in something with the hash value of 0.4, our state also will not change. Similarly, if we add more items with a hash value of 0.3, our state will also not change. This is a property called *idempotence*; it means that if we do the same operation, with the same inputs, on this structure multiple times, the state will not be changed. This is in contrast to, for example, an append on a `list`, which will always change its value. This concept of idempotence carries on to all of the data structures in this section except for the Morris counter.

[Example 11-28](#) shows a very basic K-Minimum Values implementation. Of note is our use of a `sortedset`, which, like a set, can contain only unique items. This uniqueness gives our `KMinValues` structure idempotence for free. To see this, follow the code through: when the same item is added more than once, the `data` property does not change.

⁴ Kevin Beyer et al., “On Synopses for Distinct-Value Estimation under Multiset Operations,” in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (New York: ACM, 2007), 199–210, <https://doi.org/10.1145/1247480.1247504>.

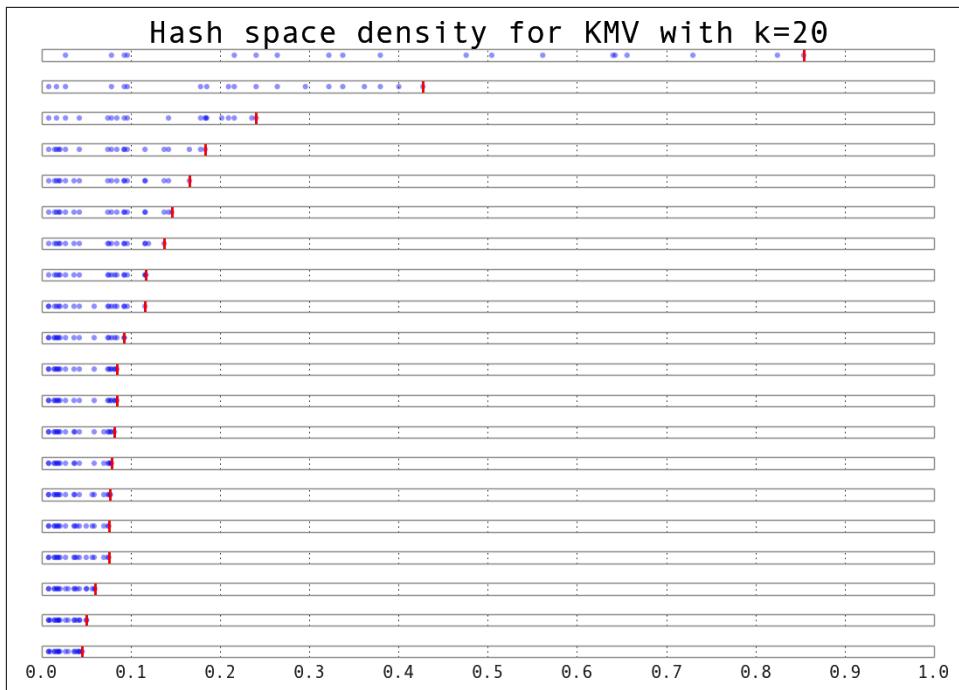


Figure 11-8. The value stores in a KMV structure as more elements are added

Example 11-28. Simple KMinValues implementation

```
import mmh3
from blist import sortedset

class KMinValues:
    def __init__(self, num_hashes):
        self.num_hashes = num_hashes
        self.data = sortedset()

    def add(self, item):
        item_hash = mmh3.hash(item)
        self.data.add(item_hash)
        if len(self.data) > self.num_hashes:
            self.data.pop()

    def __len__(self):
        if len(self.data) == 2:
            return 0
        length = (self.num_hashes - 1) * (2 ** 32 - 1) /
                  (self.data[-2] + 2 ** 31 - 1)
        return int(length)
```

Using the `KMinValues` implementation in the Python package `countmemaybe` ([Example 11-29](#)), we can begin to see the utility of this data structure. This implementation is very similar to the one in [Example 11-28](#), but it fully implements the other set operations, such as union and intersection. Also note that “size” and “cardinality” are used interchangeably (the word “cardinality” is from set theory and is used more in the analysis of probabilistic data structures). Here, we can see that even with a reasonably small value for k , we can store 50,000 items and calculate the cardinality of many set operations with relatively low error.

Example 11-29. countmemaybe KMinValues implementation

```
>>> from countmemaybe import KMinValues

>>> kmv1 = KMinValues(k=1024)

>>> kmv2 = KMinValues(k=1024)

>>> for i in range(0,50000): ❶
    kmv1.add(str(i))
    ...
    ...

>>> for i in range(25000, 75000): ❷
    kmv2.add(str(i))
    ...
    ...

>>> print(len(kmv1))
50416

>>> print(len(kmv2))
52439

>>> print(kmv1.cardinality_intersection(kmv2))
25900.2862992

>>> print(kmv1.cardinality_union(kmv2))
75346.2874158
```

- ① We put 50,000 elements into `kmv1`.
 - ② `kmv2` also gets 50,000 elements, 25,000 of which are also in `kmv1`.



With these sorts of algorithms, the choice of hash function can have a drastic effect on the quality of the estimates. Both of these implementations use `mmb3`, a Python implementation of `murmurhash3` that has nice properties for hashing strings. However, different hash functions could be used if they are more convenient for your particular dataset.

Bloom Filters

Sometimes we need to be able to do other types of set operations, for which we need to introduce new types of probabilistic data structures. *Bloom filters* were created to answer the question of whether we've seen an item before.⁵

Bloom filters work by having multiple hash values in order to represent a value as multiple integers. If we later see something with the same set of integers, we can be reasonably confident that it is the same value.

To do this in a way that efficiently utilizes available resources, we implicitly encode the integers as the indices of a list. This could be thought of as a list of `bool` values that are initially set to `False`. If we are asked to add an object with hash values [10, 4, 7], we set the tenth, fourth, and seventh indices of the list to `True`. In the future, if we are asked if we have seen a particular item before, we simply find its hash values and check if all the corresponding spots in the `bool` list are set to `True`.

This method gives us no false negatives and a controllable rate of false positives. If the Bloom filter says we have not seen an item before, we can be 100% sure that we haven't seen the item before. On the other hand, if the Bloom filter states that we *have* seen an item before, there is a probability that we actually have not and we are simply seeing an erroneous result. This erroneous result comes from the fact that we will have hash collisions, and sometimes the hash values for two objects will be the same even if the objects themselves are not the same. However, in practice Bloom filters are set to have error rates below 0.5%, so this error can be acceptable.



We can simulate having as many hash functions as we want simply by having two hash functions that are independent of each other. This method is called *double hashing*. If we have a hash function that gives us two independent hashes, we can do this:

```
def multi_hash(key, num_hashes):
    hash1, hash2 = hashfunction(key)
    for i in range(num_hashes):
        yield (hash1 + i * hash2) % (2^32 - 1)
```

The modulo ensures that the resulting hash values are 32-bit (we would modulo by $2^{64} - 1$ for 64-bit hash functions).

⁵ Burton H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM* 13, no. 7 (1970): 422–26, <http://doi.org/10.1145/362686.362692>.

The exact length of the `bool` list and the number of hash values per item we need will be fixed based on the capacity and the error rate we require. With some reasonably simple statistical arguments,⁶ we see that the ideal values are as follows:

$$\text{num_bits} = -\text{capacity} \times \frac{\log(\text{error})}{\log(2)^2}$$

$$\text{num_hashes} = \text{num_bits} \times \frac{\log(2)}{\text{capacity}}$$

If we wish to store 50,000 objects (no matter how big the objects themselves are) at a false positive rate of 0.05% (that is to say, 0.05% of the times we say we have seen an object before, we actually have not), it would require 791,015 bits of storage and 11 hash functions.

To further improve our efficiency in terms of memory use, we can use single bits to represent the `bool` values (a native `bool` actually takes 4 bits). We can do this easily by using the `bitarray` module. [Example 11-30](#) shows a simple Bloom filter implementation.

Example 11-30. Simple Bloom filter implementation

```
import math

import bitarray
import mmh3

class BloomFilter:
    def __init__(self, capacity, error=0.005):
        """
        Initialize a Bloom filter with given capacity and false positive rate
        """
        self.capacity = capacity
        self.error = error
        self.num_bits = int((-capacity * math.log(error)) // math.log(2) ** 2 + 1)
        self.num_hashes = int((self.num_bits * math.log(2)) // capacity + 1)
        self.data = bitarray.bitarray(self.num_bits)

    def _indexes(self, key):
        h1, h2 = mmh3.hash64(key)
        for i in range(self.num_hashes):
            yield (h1 + i * h2) % self.num_bits
```

⁶ The [Wikipedia page on Bloom filters](#) has a very simple proof for the properties of a Bloom filter.

```

def add(self, key):
    for index in self._indexes(key):
        self.data[index] = True

def __contains__(self, key):
    return all(self.data[index] for index in self._indexes(key))

def __len__(self):
    bit_off_num = self.data.count(True)
    bit_off_percent = 1.0 - bit_off_num / self.num_bits
    length = -1.0 * self.num_bits * math.log(bit_off_percent) / self.num_hashes
    return int(length)

@staticmethod
def union(bloom_a, bloom_b):
    assert bloom_a.capacity == bloom_b.capacity, "Capacities must be equal"
    assert bloom_a.error == bloom_b.error, "Error rates must be equal"

    bloom_union = BloomFilter(bloom_a.capacity, bloom_a.error)
    bloom_union.data = bloom_a.data | bloom_b.data
    return bloom_union

```

What happens if we insert more items than we specified for the capacity of the Bloom filter? At the extreme end, all the items in the `bool` list will be set to `True`, in which case we say that we have seen every item. This means that Bloom filters are very sensitive to what their initial capacity was set to, which can be quite aggravating if we are dealing with a set of data whose size is unknown (for example, a stream of data).

One way of dealing with this is to use a variant of Bloom filters called *scalable* Bloom filters.⁷ They work by chaining together multiple Bloom filters whose error rates vary in a specific way.⁸ By doing this, we can guarantee an overall error rate and add a new Bloom filter when we need more capacity. To check if we've seen an item before, we iterate over all of the sub-Blooms until either we find the object or we exhaust the list. A sample implementation of this structure can be seen in [Example 11-31](#), where we use the previous Bloom filter implementation for the underlying functionality and have a counter to simplify knowing when to add a new Bloom.

⁷ Paolo Sérgio Almeida et al., “Scalable Bloom Filters,” *Information Processing Letters* 101, no. 6 (2007) 255–61, <https://doi.org/10.1016/j.ipl.2006.10.007>.

⁸ The error values actually decrease like the geometric series. This way, when you take the product of all the error rates, it approaches the desired error rate.

Example 11-31. Simple scaling Bloom filter implementation

```
from bloomfilter import BloomFilter

class ScalingBloomFilter:
    def __init__(self, capacity, error=0.005, max_fill=0.8,
                 error_tightening_ratio=0.5):
        self.capacity = capacity
        self.base_error = error
        self.max_fill = max_fill
        self.items_until_scale = int(capacity * max_fill)
        self.error_tightening_ratio = error_tightening_ratio
        self.bloom_filters = []
        self.current_bloom = None
        self._add_bloom()

    def _add_bloom(self):
        new_error = self.base_error * self.error_tightening_ratio ** len(
            self.bloom_filters)
        )
        new_bloom = BloomFilter(self.capacity, new_error)
        self.bloom_filters.append(new_bloom)
        self.current_bloom = new_bloom
        return new_bloom

    def add(self, key):
        if key in self:
            return True
        self.current_bloom.add(key)
        self.items_until_scale -= 1
        if self.items_until_scale == 0:
            bloom_size = len(self.current_bloom)
            bloom_max_capacity = int(self.current_bloom.capacity * self.max_fill)

            # We may have been adding many duplicate values into the Bloom, so
            # we need to check if we actually need to scale or if we still have
            # space
            if bloom_size >= bloom_max_capacity:
                self._add_bloom()
                self.items_until_scale = bloom_max_capacity
            else:
                self.items_until_scale = int(bloom_max_capacity - bloom_size)
        return False

    def __contains__(self, key):
        return any(key in bloom for bloom in self.bloom_filters)

    def __len__(self):
        return int(sum(len(bloom) for bloom in self.bloom_filters))
```

Another way of dealing with this is using a method called *timing Bloom filters*. This variant allows elements to be expired out of the data structure, thus freeing up space for more elements. This is especially nice for dealing with streams, since we can have elements expire after, say, an hour and have the capacity set large enough to deal with the amount of data we see per hour. Using a Bloom filter this way would give us a nice view into what has been happening in the last hour.

Using this data structure will feel much like using a `set` object. In the following interaction, we use the scalable Bloom filter to add several objects, test if we've seen them before, and then try to experimentally find the false positive rate:

```
>>> bloom = BloomFilter(100)

>>> for i in range(50):
....:     bloom.add(str(i))
....:

>>> "20" in bloom
True

>>> "25" in bloom
True

>>> "51" in bloom
False

>>> num_false_positives = 0

>>> num_true_negatives = 0

>>> # None of the following numbers should be in the Bloom.
>>> # If one is found in the Bloom, it is a false positive.
>>> for i in range(51,10000):
....:     if str(i) in bloom:
....:         num_false_positives += 1
....:     else:
....:         num_true_negatives += 1
....:

>>> num_false_positives
54

>>> num_true_negatives
9895

>>> false_positive_rate = num_false_positives / float(10000 - 51)

>>> false_positive_rate
0.005427681173987335
```

```
>>> bloom.error
0.005

>>> bloom_a = BloomFilter(200)

>>> bloom_b = BloomFilter(200)

>>> for i in range(50):
...:     bloom_a.add(str(i))
...:

>>> for i in range(25,75):
...:     bloom_b.add(str(i))
...:

>>> bloom = BloomFilter.union(bloom_a, bloom_b)

>>> "51" in bloom_a ❶
Out[9]: False

>>> "24" in bloom_b ❷
Out[10]: False

>>> "55" in bloom ❸
Out[11]: True

>>> "25" in bloom
Out[12]: True
```

- ❶ The value of 51 is not in `bloom_a`.
- ❷ Similarly, the value of 24 is not in `bloom_b`.
- ❸ However, the `bloom` object contains all the objects in both `bloom_a` and `bloom_b`!

One caveat is that you can take the union of only two Blooms with the same capacity and error rate. Furthermore, the final Bloom's used capacity can be as high as the sum of the used capacities of the two Blooms unioned to make it. This means that you could start with two Bloom filters that are a little more than half full and, when you union them together, get a new Bloom that is over capacity and not reliable!



A **Cuckoo filter** is a modern Bloom filter-like data structure that provides similar functionality to a Bloom filter with the addition of better object deletion. Furthermore, the Cuckoo filter in most cases has lower overhead, leading to better space efficiencies than the Bloom filter. When a fixed number of objects needs to be kept track of, it is often a better option. However, its performance degrades dramatically when its load limit is reached and there are no options for automatic scaling of the data structure (as we saw with the scaling Bloom filter).

The work of doing fast set inclusion in a memory-efficient way is a very important and active part of database research. Cuckoo filters, **Bloomier filters**, **Xor filters**, and more are being constantly released. However, for most applications, it is still best to stick with the well-known, well-supported Bloom filter.

LogLog Counter

LogLog-type counters are based on the realization that the individual bits of a hash function can also be considered random. That is to say, the probability of the first bit of a hash being 1 is 50%, the probability of the first two bits being 01 is 25%, and the probability of the first three bits being 001 is 12.5%. Knowing these probabilities, and keeping the hash with the most 0s at the beginning (i.e., the least probable hash value), we can come up with an estimate of how many items we've seen so far.

A good analogy for this method is flipping coins. Imagine we would like to flip a coin 32 times and get heads every time. The number 32 comes from the fact that we are using 32-bit hash functions. If we flip the coin once and it comes up tails, we will store the number 0, since our best attempt yielded 0 heads in a row. Since we know the probabilities behind this coin flip, we can also tell you that our longest series was 0 long, and you can estimate that we've tried this experiment $2^0 = 1$ time. If we keep flipping our coin and we're able to get 10 heads before getting a tail, then we would store the number 10. Using the same logic, you could estimate that we've tried the experiment $2^{10} = 1024$ times. With this system, the highest we could count would be the maximum number of flips we consider (for 32 flips, this is $2^{32} = 4,294,967,296$).

To encode this logic with LogLog-type counters, we take the binary representation of the hash value of our input and see how many 0s there are before we see our first 1. The hash value can be thought of as a series of 32 coin flips, where 0 means a flip for heads and 1 means a flip for tails (i.e., 000010101101 means we flipped four heads before our first tails, and 010101101 means we flipped one head before flipping our first tail). This gives us an idea of how many tries happened before this hash value was reached. The mathematics behind this system is almost equivalent to that of the Morris counter, with one major exception: we acquire the “random” values by

looking at the actual input instead of using a random number generator. This means that if we keep adding the same value to a LogLog counter, its internal state will not change. [Example 11-32](#) shows a simple implementation of a LogLog counter.

Example 11-32. Simple implementation of LogLog register

```
import mmh3

def trailing_zeros(number):
    """
    Returns the index of the first bit set to 1 from the right side of a 32-bit
    integer
    >>> trailing_zeros(0)
    32
    >>> trailing_zeros(0b1000)
    3
    >>> trailing_zeros(0b10000000)
    7
    """
    if not number:
        return 32
    index = 0
    while (number >> index) & 1 == 0:
        index += 1
    return index

class LogLogRegister:
    counter = 0
    def add(self, item):
        item_hash = mmh3.hash(str(item))
        return self._add(item_hash)

    def _add(self, item_hash):
        bit_index = trailing_zeros(item_hash)
        if bit_index > self.counter:
            self.counter = bit_index

    def __len__(self):
        return 2**self.counter
```

The biggest drawback of this method is that we may get a hash value that increases the counter right at the beginning and skews our estimates. This would be similar to flipping 32 tails on the first try. To remedy this, we should have many people flipping coins at the same time and combine their results. The law of large numbers tells us that as we add more and more flippers, the total statistics become less affected by anomalous samples from individual flippers. The exact way that we combine the results is the root of the difference between LogLog-type methods (classic LogLog, SuperLogLog, HyperLogLog, HyperLogLog++, etc.).

We can accomplish this “multiple flipper” method by taking the first couple of bits of a hash value and using that to designate which of our flippers had that particular result. If we take the first 4 bits of the hash, this means we have $2^4 = 16$ flippers. Since we used the first 4 bits for this selection, we have only 28 bits left (corresponding to 28 individual coin flips per coin flipper), meaning each counter can count only up to $2^{28} = 268,435,456$. In addition, there is a constant (α) that depends on the number of flippers, which normalizes the estimation.⁹ All of this together gives us an algorithm with $1.05 / \sqrt{m}$ accuracy, where m is the number of registers (or flippers) used. [Example 11-33](#) shows a simple implementation of the LogLog algorithm.

Example 11-33. Simple implementation of LogLog

```
import mmh3

from llregister import LLRegister

class LL:
    def __init__(self, p):
        self.p = p
        self.num_registers = 2 ** p
        self.registers = [LLRegister() for i in range(int(2 ** p))]
        self.alpha = 0.7213 / (1.0 + 1.079 / self.num_registers)

    def add(self, item):
        item_hash = mmh3.hash(str(item))
        register_index = item_hash & (self.num_registers - 1)
        register_hash = item_hash >> self.p
        self.registers[register_index].add(register_hash)

    def __len__(self):
        register_sum = sum(h.counter for h in self.registers)
        length = (self.num_registers * self.alpha *
                  2 ** (register_sum / self.num_registers))
        return int(length)
```

In addition to this algorithm deduplicating similar items by using the hash value as an indicator, it has a tunable parameter that can be used to dial whatever sort of accuracy versus storage compromise you are willing to make.

⁹ A full description of the basic LogLog and SuperLogLog algorithms can be found at http://bit.ly/algorithm_desc.

In the `__len__` method, we are averaging the estimates from all of the individual LogLog registers. This, however, is not the most efficient way to combine the data! This is because we may get some unfortunate hash values that make one particular register spike up while the others are still at low values. Because of this, we are able to achieve an error rate of only $O\left(\frac{1.30}{\sqrt{m}}\right)$, where m is the number of registers used.

SuperLogLog was devised as a fix to this problem.¹⁰ With this algorithm, only the lowest 70% of the registers were used for the size estimate, and their value was limited by a maximum value given by a restriction rule. This addition decreased the error rate to $O\left(\frac{1.05}{\sqrt{m}}\right)$. This is counterintuitive, since we got a better estimate by disregarding information!

Finally, HyperLogLog came out in 2007 and gave us further accuracy gains.¹¹ It did so by changing the method of averaging the individual registers: instead of just averaging, we use a spherical averaging scheme that also has special considerations for different edge cases the structure could be in. This brings us to the current best error rate of $O\left(\frac{1.04}{\sqrt{m}}\right)$. In addition, this formulation removes a sorting operation that is necessary with SuperLogLog. This can greatly speed up the performance of the data structure when you are trying to insert items at a high volume. [Example 11-34](#) shows a basic implementation of HyperLogLog.

Example 11-34. Simple implementation of HyperLogLog

```
import math

from ll import LL

class HyperLogLog(LL):
    def __len__(self):
        indicator = sum(2 ** -m.counter for m in self.registers)
        E = self.alpha * (self.num_registers ** 2) / indicator

        if E <= 5.0 / 2.0 * self.num_registers:
            V = sum(1 for m in self.registers if m.counter == 0)
            if V != 0:
                Estar = (self.num_registers *
                          math.log(self.num_registers / (1.0 * V), 2))
```

¹⁰ Marianne Durand and Philippe Flajolet, “LogLog Counting of Large Cardinalities,” in *Algorithms—ESA 2003*, ed. Giuseppe Di Battista and Uri Zwick, vol. 2832 (Berlin, Heidelberg: Springer, 2003), 605–17, https://doi.org/10.1007/978-3-540-39658-1_55.

¹¹ Philippe Flajolet et al., “HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm,” in *AOFA ’07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, (AOFA, 2007), 127–46.

```

        else:
            Estar = E
    else:
        if E <= 2 ** 32 / 30.0:
            Estar = E
        else:
            Estar = -2 ** 32 * math.log(1 - E / 2 ** 32, 2)
    return int(Estar)

if __name__ == "__main__":
    import mmh3

    hll = HyperLogLog(8)
    for i in range(100000):
        hll.add(mmh3.hash(str(i)))
    print(len(hll))

```

The only further increase in accuracy was given by the HyperLogLog++ algorithm, which increases the accuracy of the data structure while it is relatively empty. When more items are inserted, this scheme reverts to standard HyperLogLog. This is actually quite useful, since the statistics of the LogLog-type counters require a lot of data to be accurate—having a scheme for allowing better accuracy with fewer number items greatly improves the usability of this method. This extra accuracy is achieved by having a smaller but more accurate HyperLogLog structure that can later be converted into the larger structure that was originally requested. Also, some empirically derived constants are used in the size estimates that remove biases.

Real-World Example

To obtain a better understanding of the data structures, we first created a dataset with many unique keys, and then one with duplicate entries. Figures 11-9 and 11-10 show the results when we feed these keys into the data structures we've just looked at and periodically query, "How many unique entries have there been?" We can see that the data structures that contain more stateful variables (such as HyperLogLog and KMin Values) do better, since they more robustly handle bad statistics. On the other hand, the Morris counter and the single LogLog register can quickly have very high error rates if one unfortunate random number or hash value occurs. For most of the algorithms, however, we know that the number of stateful variables is directly correlated with the error guarantees, so this makes sense.

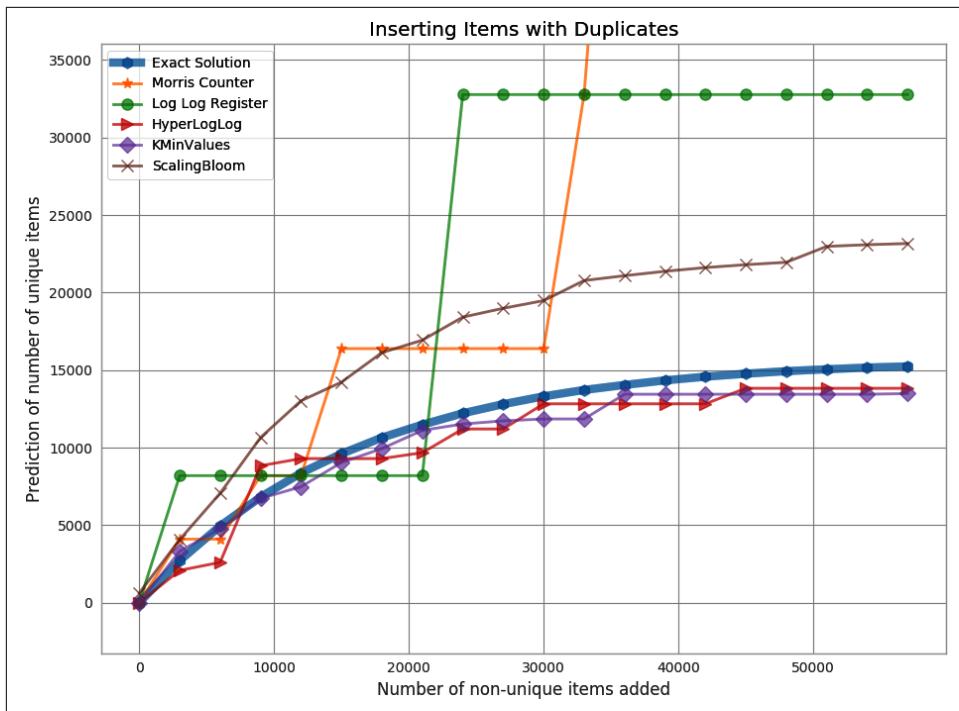


Figure 11-9. The approximate count of duplicate data using multiple probabilistic data structures. To do this, we generate 60,000 items with many duplicates and insert them into the various probabilistic data structures. Graphed is the structures prediction of the number of unique items during the process.

Looking just at the probabilistic data structures that have the best performance (which really are the ones you will probably use), we can summarize their utility and their approximate memory usage (see Table 11-2). We can see a huge change in memory usage depending on the questions we care to ask. This simply highlights the fact that when using a probabilistic data structure, you must first consider what questions you really need to answer about the dataset before proceeding. Also note that only the Bloom filter's size depends on the number of elements. The sizes of the HyperLogLog and KMinValues structures are sizes dependent *only* on the error rate.

As another, more realistic test, we chose to use a dataset derived from the text of Wikipedia. We ran a very simple script in order to extract all single-word tokens with five or more characters from all articles and store them in a newline-separated file. The question then was, “How many unique tokens are there?” The results can be seen in Table 11-3.

Table 11-2. Comparison of major probabilistic data structures and the set operations available on them

	Size	Union ^a	Intersection	Contains	Size ^b
HyperLogLog	Yes ($O\left(\frac{1.04}{\sqrt{m}}\right)$)	Yes	No ^c	No	2.704 MB
KMinValues	Yes ($O\left(\sqrt{\frac{2}{n(m-2)}}\right)$)	Yes	Yes	No	20.372 MB
Bloom filter	Yes ($O\left(\frac{0.78}{\sqrt{m}}\right)$)	Yes	No ^c	Yes	197.8 MB

^a Union operations occur without increasing the error rate.

^b Size of data structure with 0.05% error rate, 100 million unique elements, and using a 64-bit hashing function.

^c These operations can be done but at a considerable penalty in terms of accuracy.

The major takeaway from this experiment is that if you are able to specialize your code, you can get amazing speed and memory gains. This has been true throughout the entire book: when we specialized our code in “Selective Optimizations: Finding What Needs to Be Fixed” on page 137, we were similarly able to get speed increases.

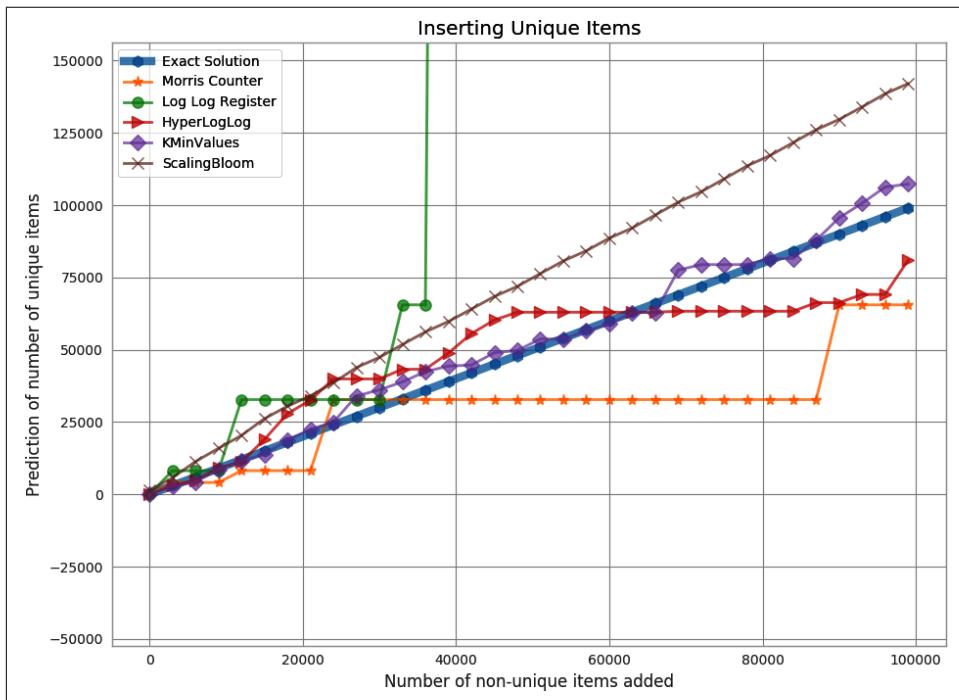


Figure 11-10. The approximate count of unique data using multiple probabilistic data structures. To do this, we insert the numbers 1 through 100,000 into the data structures. Graphed is the structures prediction of the number of unique items during the process.

Probabilistic data structures are an algorithmic way of specializing your code. We store only the data we need in order to answer specific questions with given error bounds. By having to deal with only a subset of the information given, we can not only make the memory footprint much smaller, but also perform most operations over the structure faster.

Table 11-3. Size estimates for the number of unique words in Wikipedia

	Elements	Relative error	Processing time ^a	Structure size ^b
Morris counter ^c	1,073,741,824	6.52%	751s	5 bits
LogLog register	1,048,576	78.84%	1,690s	5 bits
LogLog	4,522,232	8.76%	2,112s	5 bits
HyperLogLog	4,983,171	-0.54%	2,907s	40 KB
KMinValues	4,912,818	0.88%	3,503s	256 KB
Scaling Bloom	4,949,358	0.14%	10,392s	11,509 KB
True value	4,956,262	0.00%	-----	49,558 KB ^d

^a Processing time has been adjusted to remove the time to read the dataset from disk. We also use the simple implementations provided earlier for testing.

^b Structure size is theoretical given the amount of data since the implementations used were not optimized.

^c Since the Morris counter doesn't deduplicate input, the size and relative error are given with regard to the total number of values.

^d The dataset is 49,558 KB considering only unique tokens, or 8.742 GB with all tokens.

As a result, whether or not you use probabilistic data structures, you should always keep in mind what questions you are going to be asking of your data and how you can most effectively store that data in order to ask those specialized questions. This may come down to using one particular type of list over another, using one particular type of database index over another, or maybe even using a probabilistic data structure to throw out all but the relevant data!

Lessons from the Field

Questions You'll Be Able to Answer After This Chapter

- How do successful start-ups handle large volumes of data and machine learning?
- What monitoring and deployment technologies keep systems stable?
- What lessons have successful CTOs learned about their technologies and teams?
- How widely can PyPy be deployed?

In this chapter we have collected stories from successful companies that use Python in high-data-volume and speed-critical situations. The stories are written by key people in each organization who have many years of experience; they share not just their technology choices but also some of their hard-won wisdom. We have four great new stories for you from other experts in our domain. We've also kept the "Lessons from the Field" from the first edition of this book; their titles are marked "(2014)."

Streamlining Feature Engineering Pipelines with Feature-engine

Soledad Galli (trainindata.com)

Soledad Galli is the lead data scientist and founder of Train in Data. She has experience in finance and insurance, received a Data Leadership Award in 2018, and was selected as one of LinkedIn's "Top Voices" in data science and analytics in 2019. Soledad is passionate about sharing knowledge and helping others succeed in data science.

Train in Data is an education project led by experienced data scientists and AI software engineers. We help professionals improve coding and data science skills and adopt machine learning best practices. We create advanced online courses on machine learning and AI software engineering and open source libraries, like [Feature-engine](#), to smooth the delivery of machine learning solutions.

Feature Engineering for Machine Learning

Machine learning models take in a bunch of input variables and output a prediction. In finance and insurance, we build models to predict, for example, the likelihood of a loan being repaid, the probability of an application being fraudulent, and whether a car should be repaired or replaced after an accident. The data we collect and store or recall from third-party APIs is almost never suitable to train machine learning models or return predictions. Instead, we transform variables extensively before feeding them to machine learning algorithms. We refer to the collection of variable transformations as *feature engineering*.

Feature engineering includes procedures to impute missing data, encode categorical variables, transform or discretize numerical variables, put features in the same scale, combine features into new variables, extract information from dates, aggregate transactional data, and derive features from time series, text, or even images. There are many techniques for each of these feature engineering steps, and your choice will depend on the characteristics of the variables and the algorithms you intend to use. Thus, when feature engineers build and consume machine learning in organizations, we do not speak of machine learning models but of machine learning pipelines, where a big part of the pipeline is devoted to feature engineering and data transformation.

The Hard Task of Deploying Feature Engineering Pipelines

Many feature engineering transformations learn parameters from data. I have seen organizations utilize config files with hardcoded parameters. These files limit versatility and are hard to maintain (every time you retrain your model, you need to rewrite the config file with the new parameters). To create highly performant feature engineering pipelines, it's better to develop algorithms that automatically learn and store these parameters and can also be saved and loaded, ideally as one object.

At Train in Data, we develop machine learning pipelines in the research environment and deploy them in the production environment. These pipelines should be reproducible. Reproducibility is the ability to duplicate a machine learning model exactly, such that, given the same data as input, both models return the same output. Utilizing the same code in the research and production environment smooths the deployment of machine learning pipelines by minimizing the amount of code to be rewritten, maximizing reproducibility.

Feature engineering transformations need to be tested. Unit tests for each feature engineering procedure ensure that the algorithm returns the desired outcome. Extensive code refactoring in production to add unit and integration tests is extremely time-consuming and provides new opportunities to introduce bugs, or find bugs introduced during the research phase due to the lack of testing. To minimize code refactoring in production, it is better to introduce unit testing as we develop the engineering algorithms in the research phase.

The same feature engineering transformations are used across projects. To avoid different code implementations of the same technique, which often occurs in teams with many data scientists, and to enhance team performance, speed up model development, and smooth model operationalization, we want to reuse code that was previously built and tested. The best way to do that is to create in-house packages. Creating packages may seem time-consuming, since it involves building tests and documentation. But it is more efficient in the long run, because it allows us to enhance code and add new features incrementally, while reusing code and functionality that has already been developed and tested. Package development can be tracked through versioning and even shared with the community as open source, raising the profile of the developers and the organization.

Leveraging the Power of Open Source Python Libraries

It's important to use established open source projects or thoroughly developed in-house libraries. This is more efficient for several reasons:

- Well-developed projects tend to be thoroughly documented, so it is clear what each piece of code intends to achieve.

- Well-established open source packages are tested to prevent the introduction of bugs, ensure that the transformation achieves the intended outcome, and maximize reproducibility.
- Well-established projects have been widely adopted and approved by the community, giving you peace of mind that the code is of quality.
- You can use the same package in the research and production environment, minimizing code refactoring during deployment.
- Packages are clearly versioned, so you can deploy the version you used when developing your pipeline to ensure reproducibility, while newer versions continue to add functionality.
- Open source packages can be shared, so different organizations can build tools together.
- While open source packages are maintained by a group of experienced developers, the community can also contribute, providing new ideas and features that raise the quality of the package and code.
- Using a well-established open source library removes the task of coding from our hands, improving team performance, reproducibility, and collaboration, while reducing model research and deployment timelines.

Open source Python libraries like [scikit-learn](#), [Category encoders](#), and [Featuretools](#) provide feature engineering functionality. To expand on existing functionality and smooth the creation and deployment of machine learning pipelines, I created the open source Python package [Feature-engine](#), which provides an exhaustive battery of feature engineering procedures and supports the implementation of different transformations to distinct feature spaces.

Feature-engine Smooths Building and Deployment of Feature Engineering Pipelines

Feature engineering algorithms need to learn parameters from data automatically, return a data format that facilitates use in research and production environments, and include an exhaustive battery of transformations to encourage adoption across projects. Feature-engine was conceived and designed to fulfill all these requirements. Feature-engine transformers—that is, the classes that implement a feature engineering transformation—learn and store parameters from data. Feature-engine transformers return Pandas DataFrames, which are suitable for data analysis and visualization during the research phase. Feature-engine supports the creation and storage of an entire end-to-end engineering pipeline in a single object, making deployment easier. And to facilitate adoption across projects, it includes an exhaustive list of feature transformations.

Feature-engine includes many procedures to impute missing data, encode categorical variables, transform and discretize numerical variables, and remove or censor outliers. Each transformer can learn, or alternatively have specified, the group of variables it should modify. Thus, the transformer can receive the entire dataframe, yet it will alter only the selected variable group, removing the need of additional transformers or manual work to slice the dataframe and then join it back together.

Feature-engine transformers use the fit/transform methods from scikit-learn and expand its functionality to include additional engineering techniques. Fit/transform functionality makes Feature-engine transformers usable within the scikit-learn pipeline. Thus with Feature-engine we can store an entire machine learning pipeline into a single object that can be saved and retrieved or placed in memory for live scoring.

Helping with the Adoption of a New Open Source Package

No matter how good an open source package is, if no one knows it exists or if the community can't easily understand how to use it, it will not succeed. Making a successful open source package entails making code that is performant, well tested, well documented, and useful—and then letting the community know that it exists, encouraging its adoption by users who can suggest new features, and attracting a developer community to add more functionality, improve the documentation, and enhance code quality to raise its performance. For a package developer, this means that we need to factor in time to develop code and to design and execute a sharing strategy. Here are some strategies that have worked for me and for other package developers.

We can leverage the power of well-established open source functionality to facilitate adoption by the community. Scikit-learn is the reference library for machine learning in Python. Thus, adopting scikit-learn fit/transform functionality in a new package facilitates an easy and fast adoption by the community. The learning curve to use the package is shorter as users are already familiar with this implementation. Some packages that leverage the use of fit/transform are [Keras](#), Category encoders (perhaps the most renowned), and of course Feature-engine.

Users want to know how the package can be used and shared, so include a license stating these conditions in the code repository. Users also need instructions and examples of the code functionality. Docstrings in code files with information about functionality and examples of its use is a good start, but it is not enough. Widespread packages include additional documentation (which can be generated with ReStructuredText files) with descriptions of code functionality, examples of its use and the outputs returned, installation guidelines, the channels where the package is available (PyPI, conda), how to get started, changelogs, and more. Good documentation should empower users to use the library without reading the source code. The

documentation of the machine learning visualization library [Yellowbrick](#) is a good example. I have adopted this practice for Feature-engine as well.

How can we increase package visibility? How do we reach potential users? Teaching an online course can help you reach people, especially if it's on a prestigious online learning platform. In addition, publishing documentation in [Read the Docs](#), creating YouTube tutorials, and presenting the package at meetups and meetings all increase visibility. Presenting the package functionality while answering relevant questions in well-established user networks like Stack Overflow, Stack Exchange, and Quora can also help. The developers of Featuretools and Yellowbrick have leveraged the power of these networks. Creating a dedicated Stack Overflow issues list lets users ask questions and shows the package is being actively maintained.

Developing, Maintaining, and Encouraging Contribution to Open Source Libraries

For a package to be successful and relevant, it needs an active developer community. A developer community is composed of one or, ideally, a group of dedicated developers or maintainers who will watch for overall functionality, documentation, and direction. An active community allows and welcomes additional ad hoc contributors.

One thing to consider when developing a package is code maintainability. The simpler and shorter the code, the easier it is to maintain, which makes it more likely to attract contributors and maintainers. To simplify development and maintainability, Feature-engine leverages the power of scikit-learn base transformers. Scikit-learn provides an API with a bunch of base classes that developers can build upon to create new transformers. In addition, scikit-learn's API provides many tests to ensure compatibility between packages and also that the transformer delivers the intended functionality. By using these, Feature-engine developers and maintainers focus only on feature engineering functionality, while the maintenance of base code is taken care of by the bigger scikit-learn community. This, of course, has a trade-off. If scikit-learn changes its base functionality, we need to update our library to ensure it is compatible with the latest version. Other open source packages that use scikit-learn API are Yellowbrick and Category encoders.

To encourage developers to collaborate, [NumFOCUS](#) recommends creating a code of conduct and encouraging inclusion and diversity. The project needs to be open, which generally means the code should be publicly hosted, with guidelines to orient new contributors about project development and discussion forums open to public participation, like a mailing list or a Slack channel. While some open source Python libraries have their own codes of conduct, others, like Yellowbrick and Feature-engine, follow the [Python Community Code of Conduct](#). Many open source projects, including Feature-engine, are publicly hosted on GitHub. Contributing guidelines list ways new contributors can help—for example, by fixing bugs, adding new

functionality, or enhancing the documentation. Contributing guidelines also inform new developers of the contributing cycle, how to fork the repository, how to work on the contributing branch, how the code review cycle works, and how to make a Pull Request.

Collaboration can raise the quality and performance of the library by enhancing code quality or functionality, adding new features, and improving documentation. Contributions can be as simple as reporting typos in the documentation, reporting code that doesn't return the intended outcome, or requesting new features. Collaborating on open source libraries can also help raise the collaborator's profile while exposing them to new engineering and coding practices, improving their skills.

Many developers and data scientists believe that they need to be top-notch developers to contribute to open source projects. I used to believe that myself, and it discouraged me from making contributions or requesting features—even though, as a user, I had a clear idea of what features were available and which were missing. This is far from true. Any user can contribute to the library. And package maintainers love contributions.

Useful contributions to Feature-engine have included simple things like adding a line to the `.gitignore`, sending a message through LinkedIn to bring typos in the docs to my attention, making a PR to correct typos themselves, highlighting warning issues raised by newer versions of scikit-learn, requesting new functionality, or expanding the battery of unit tests.

If you want to contribute but have no experience, it is useful to go through the issues found on the package repository. Issues are lists with the modifications to the code that have priority. They are tagged with labels like “code enhancement,” “new functionality,” “bug fix,” or “docs.” To start, it is good to go after issues tagged as “good first issue” or “good for new contributors”; those tend to be smaller code changes and will allow you to get familiar with the contribution cycle. Then you can jump into more complex code modifications. Just by solving an easy issue, you will learn a lot about software development, Git, and code review cycles.

Feature-engine is currently a small package with straightforward code implementations. It is easy to navigate and has few dependencies, so it is a good starting point for contributing to open source. If you want to get started, do get in touch. I will be delighted to hear from you. Good luck!

Highly Performant Data Science Teams

Linda Uruchurtu (Fiit)

Linda Uruchurtu is a senior data scientist and software engineer, currently working at Fiit. Since 2013, she has been helping small-to-medium start-ups build products using data. She has experience in analytics, statistics, machine learning, and product building in a variety of industries, including transportation, retail, health, and fitness.

Linda holds a PhD in theoretical physics from Cambridge University. She has been both a speaker and a reviewer at PyData London, and has served as chair of the PyData London review committee since 2017.

Data science teams are different from other technical teams, because the scope of what they do varies according to where they sit and the type of problems they tackle. However, whether the team is responsible for answering “why” and “how” questions or simply delivering fully operational ML services, in order to deliver successfully, they need to keep the stakeholders happy.

This can be challenging. Most data science projects have a degree of uncertainty attached to them, and since there are different types of stakeholders, “happy” can mean different things. Some stakeholders might be concerned only with final deliverables, whereas others might care about side effects or common interfaces. Additionally, some might not be technical or have a limited understanding of the specifics of the project. Here I will share some lessons I have learned that make a difference in the way projects are carried out and delivered.

How Long Will It Take?

This is possibly the most common question data science team leads are asked. Picture the following: management asks the project manager (PM), or whoever is responsible for delivery, to solve a given problem. The PM goes to the team, presents them with this information, and asks them to plan a solution. Cue this question, from the PM or from other stakeholders: How long will it take?

First, the team should ask questions to better define the scope of their solutions. These might include the following:

- Why is this a problem?
- What is the impact of solving this problem?
- What is the definition of done?
- What is the minimal version of a solution that satisfies said definition?

- Is there a way to validate a solution early on?

Notice that “How long will it take?” isn’t on this list.

The strategy should be twofold. First, get a time-boxed period to ask these questions and propose one or more solutions. Once a solution is agreed upon, the PM should explain to stakeholders that the team can provide a timeline once the work for said solution is planned.

Discovery and Planning

The team has a fixed amount of time to come up with solutions. What’s next? They need to generate hypotheses, followed by exploratory work and quick prototyping, to keep or discard potential solutions successively.

Depending on the solution chosen, other teams may become stakeholders. Development teams could have requirements from APIs they hold, or they could become consumers of a service; product, operations, customer service, and other teams might use visualizations and reports. The PM’s team should discuss their ideas with these teams.

Once this process has taken place, the team is usually in a good position to determine how much uncertainty and/or risk adheres to each option. The PM can now assess which option is preferred.

Once an option is chosen, the PM can define a timeline for milestones and deliverables. Useful points to raise here are as follows:

- Can the deliverables be reasonably reviewed and tested?
- If work depends on other teams, can work be scheduled so no delay is introduced?
- Can the team provide value from intermediate milestones?
- Is there a way to reduce risk from parts of the project that have a significant amount of uncertainty?

Tasks derived from the plan can then be sized and time-boxed to provide a time estimate. It is a good idea to allow for extra time: some people like to double or triple the time they think they’ll need!

Some tasks are frequently underestimated and simplified, including data collection and dataset building, testing, and validation. Getting good data for model building can often be more complex and expensive than it initially seems. One option may be to start with small datasets for prototyping and postpone further collection until after proof of concept. Testing, too, is fundamental, both for correctness and for reproducibility. Are inputs as expected? Are processing pipelines introducing errors? Are

outputs correct? Unit testing and integration tests should be part of every effort. Finally, validation is important, particularly in the real world. Be sure to factor in realistic estimates for all of these tasks.

Once you've done that, the team has not only an answer to the "time" question, but also a plan with milestones that everyone can use to understand what work is being carried out.

Managing Expectations and Delivery

Lots of issues can affect the time required before a delivery is achieved. Keep an eye on the following points to make sure you manage the team's expectations:

Scope creep

Scope creep is the subtle shifting of the scope of work, so that more work is expected than was initially planned. Pairing and reviews can help mitigate this.

Underestimating nontechnical tasks

Discussions, user research, documentation, and many other tasks can easily be underestimated by those who don't know them well.

Availability

Team members' scheduling and availability can also introduce delays.

Issues with data quality

From making sure working datasets are good to go to discovering sources of bias, data quality can introduce complications or even invalidate pieces of work.

Alternative options

When unexpected difficulties arise, it might make sense to consider other approaches. However, sunk costs might prevent the team from wanting to raise this, which could delay the work and risk creating the impression that the team doesn't know what they are doing.

Lack of testing

Sudden changes in data inputs or bugs in data pipelines can invalidate assumptions. Having good test coverage from the start will improve team velocity and pay dividends at the end.

Difficulty testing or validating

If not enough time is allowed for testing and validating hypotheses, the schedule can be delayed. Changes in assumptions can also lead to alterations in the testing plan.

Use weekly refinement and planning sessions to spot issues and discuss whether tasks need to be added or removed. This will give the PM enough information to update the final stakeholders. Prioritization should also happen with the same cadence. If

opportunities arise to do some tasks earlier than expected, these should be pushed forward.

Intermediate deliverables, particularly if they provide value outside the scope of the project, continuously justify the work. That's good for the team, in terms of focus and morale, as well as stakeholders, who will have a sense of progress. The continuous process of redrawing the game plan and reviewing and adjusting iterations will ensure the team has a clear sense of direction and freedom to work, while providing enough information and value to keep stakeholders keen on continuing to support the project.

To become highly performant while tackling a new project, your data science team's main focus has to be on making data uncertainty and business-need uncertainty less risky by delivering lightweight minimum viable product (MVP) solutions (think scripts and Python notebooks). The initial conceived MVP might actually turn out to be leaner than or different from the first concept, given findings down the line or changes in business needs. Only after validation should you proceed with a production-ready version.

The discovery and planning process is critical, and so is thinking in terms of iterations. Keep in mind that the discovery phase is always ongoing and that external events will always affect the plan.

Numba

Valentin Haenel (<http://haenel.co>)

Valentin Haenel is a longtime “Python for Data” user and developer who still remembers hearing Travis Oliphant’s keynote about NumPy at the first EuroSciPy conference in 2008. He then proceeded to use Python for simple modeling of spiking neurons and to evaluate data from perception experiments while pursuing his master’s degree in computational neuroscience. Since then he has contributed to more than 80 open source projects. He now works for Anaconda as an open source developer on the Numba project.

The author would like to thank three of the Numba core developers, Stuart Archibald, Siu Kwan Lam, and Stan Seibert, for productive discussions, advice, and feedback about this text.

Numba is an open source, JIT function compiler for numerically focused Python. Initially created at Continuum Analytics (now Anaconda Inc) in 2012, it has since grown into a mature open source project on GitHub with a large and diverse group of contributors. Its primary use case is the acceleration of numerical and/or scientific

Python code. The main entry point is a decorator—the `@jit` decorator—which is used to annotate the specific functions, ideally the bottlenecks of the application, that should be compiled. Numba will compile these functions just-in-time, which simply means that the function will be compiled at the first, or initial, execution. All subsequent executions with the same argument types will then use the compiled variant of the function, which should be faster than the original.

Numba not only can compile Python, but also is NumPy-aware and can handle code that uses NumPy. Under the hood, Numba relies on the well-known [LLVM project](#), a collection of modular and reusable compiler and toolchain technologies. Last, Numba isn’t a fully fledged Python compiler. It can compile only a subset of both Python and NumPy—albeit a large enough subset to make it useful in a wide range of applications. For more information, please consult the [documentation](#).

A Simple Example

As a simple example, let’s use Numba to accelerate a Python implementation of an ancient algorithm for finding all prime numbers up to a given maximum (N): the sieve of Eratosthenes. It works as follows:

- First, initialize a Boolean array of length N to all true values.
- Then starting with the first prime, 2, cross off (set the position in the Boolean list corresponding to that number to false) all multiples of the number up to N .
- Proceed to the next number that has not yet been crossed off, in this case 3, and again cross off all multiples of it.
- Continue to proceed through the numbers and cross off their multiples until you reach N .
- When you reach N , all the numbers that have not been crossed off are the set of prime numbers up to N .

A reasonably efficient Python implementation might look something like this:

```
import numpy as np
from numba import jit

@jit(nopython=True) # simply add the jit decorator
def primes(N=100000):
    numbers = np.ones(N, dtype=np.uint8) # initialize the boolean array
    for i in range(2, N):
        if numbers[i] == 0: # has previously been crossed off
            continue
        else: # it is a prime, cross off all multiples
            x = i + i
            while x < N:
                numbers[x] = 0
```

```
x += i
# return all primes, as indicated by all boolean positions that are one
return np.nonzero(numbers)[0][2:]
```

After placing this in a file called `sieve.py`, you can use the `%timeit` magic to micro-benchmark the code:

```
In [1]: from sieve import primes

In [2]: primes() # run it once to make sure it is compiled
Out[2]: array([ 2,  3,  5, ..., 99971, 99989, 99991])

In [3]: %timeit primes.py_func() # 'py_func' contains
           # the original Python implementation
145 ms ± 1.86 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [4]: %timeit primes() # this benchmarks the Numba compiled version
340 µs ± 3.98 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

That is a speedup of roughly four hundredfold; your mileage may vary. Nonetheless, there are a few things of interest here:

- Compilation happened at the function level.
- Simply adding the decorator `@jit` was enough to instruct Numba to compile the function. No further modifications to the function source code, such as type annotations of the variables, were needed.
- Numba is NumPy-aware, so all the NumPy calls in this implementation were supported and could be compiled successfully.
- The original, pure Python function is available as the `py_func` attribute of the compiled function.

There is a faster but less educational version of this algorithm, the implementation of which is left to the interested reader.

Best Practices and Recommendations

One of the most important recommendations for Numba is to use `nopython` mode whenever possible. To activate this mode, simply use the `nopython=True` option with the `@jit` decorator, as shown in the prime number example. Alternatively, you can use the `@njit` decorator alias, which is accessible by doing `from numba import njit`. In `nopython` mode, Numba attempts a large number of optimizations and can significantly improve performance. However, this mode is very strict; for compilation to succeed, Numba needs to be able to infer the types of all the variables within your function.

You can also use object mode by doing `@jit(forceobj=True)`. In this mode, Numba becomes very permissive about what it can and cannot compile, which limits it to

performing a minimal set of optimizations. This is likely to have a significant negative effect on performance. To take advantage of Numba's full potential, you really should use `nopython` mode.

If you can't decide whether you want to use object mode or not, there is the option to use an object-mode block. This can come in handy when only a small part of your code needs to execute in object mode: for example, if you have a long-running loop and would like to use string formatting to print the current progress of your program. For example:

```
from numba import njit, objmode

@njit()
def foo():
    for i in range(1000000):
        # do compute
        if i % 100000 == 0:
            with objmode: # to escape to object-mode
                # using 'format' is permissible here
                print("epoch: {}".format(i))

foo()
```

Pay attention to the types of the variables that you use. Numba works very well with both NumPy arrays and NumPy views on other datatypes. Therefore, if you can, use NumPy arrays as your preferred data structure. Tuples, strings, enums, and simple scalar types such as int, float, and Boolean are also reasonably well supported. Globals are fine for constants, but pass the rest of your data as arguments to your function. Python lists and dictionaries are unfortunately not very well supported. This largely stems from the fact that they can be type heterogeneous: a specific Python list may contain differently typed items; for example, integers, floats and strings. This poses a problem for Numba, because it needs the container to hold only items of a single type in order to compile it. However, these two data structures are probably one of the most used features of the Python language and are even one of the first things programmers learn about.

To remedy this shortcoming, Numba supports the so-called typed containers: `typed-list` and `typed-dict`. These are homogeneously typed variants of the Python list and dict. This means that they may contain only items of a single type: for example, a `typed-list` of only integer values. Beyond this limitation, they behave much like their Python counterparts and support a largely identical API. Additionally, they may be used within regular Python code or within Numba compiled functions, and can be passed into and returned from Numba compiled functions. These are available from the `numba.typed` submodule. Here is a simple example of a `typed-list`:

```
from numba import njit
from numba.typed import List
```

```

@njit
def foo(x):
    """ Copy x, append 11 to the result. """
    result = x.copy()
    result.append(11)
    return result

a = List() # Create a new typed-list
for i in (2, 3, 5, 7):
    # Add the content to the typed-list,
    # the type is inferred from the first item added.
    a.append(i)
b = foo(a) # make the call, append 11; this list will go to eleven

```

While Python does have limitations, you can rethink them and understand which ones can be safely disregarded when using Numba. Two specific examples come to mind: calling functions and for loops. Numba enables a technique called *inlining* in the underlying LLVM library to optimize away the overhead of calling functions. This means that during compilation, any function calls that are amenable to inlining are replaced with a block of code that is the equivalent of the function being called. As a result, there is practically no performance impact from breaking up a large function into a few or many small ones in order to aid readability and comprehensibility.

One of the main criticisms of Python is that its for loops are slow. Many people recommend using alternative constructs instead when attempting to improve the performance of a Python program: list comprehensions or even NumPy arrays. Numba does not suffer from this limitation, and using for loops in Numba compiled functions is fine. Observe:

```

from numba import njit

@njit
def numpy_func(a):
    # uses Numba's implementation of NumPy's sum, will also be fast in
    # Python
    return a.sum()

@njit
def for_loop(a):
    # uses a simple for-loop over the array
    acc = 0
    for i in a:
        acc += i
    return acc

```

We can now benchmark the preceding code:

```

In [1]: ... # import the above functions

In [2]: import numpy as np

```

```
In [3]: a = np.arange(1000000, dtype=np.int64)

In [4]: numpy_func(a) # sanity check and compile
Out[4]: 499999500000

In [5]: for_loop(a) # sanity check and compile
Out[5]: 499999500000

In [6]: %timeit numpy_func(a) # Compiled version of the NumPy func
174 µs ± 3.05 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [7]: %timeit for_loop(a) # Compiled version of the for-loop
186 µs ± 7.59 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [8]: %timeit numpy_func.py_func(a) # Pure NumPy func
336 µs ± 6.72 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [9]: %timeit for_loop.py_func(a) # Pure Python for-loop
156 ms ± 3.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

As you can see, both Numba compiled variants have very similar performance characteristics, whereas the pure Python `for` loop implementation is significantly (800 times) slower than its compiled counterpart.

If you are now thinking about rewriting your NumPy array expressions as `for` loops, don't! As shown in the preceding example, Numba is perfectly happy with NumPy arrays and their associated functions. In fact, Numba has an additional ace up its sleeve: an optimization known as *loop fusion*. Numba performs this technique predominantly on array expression operations. For example:

```
from numba import njit

@njit
def loop_fused(a, b):
    return a * b - 4.1 * a > 2.5 * b

In [1]: ... # import the example

In [2]: import numpy as np

In [3]: a, b = np.arange(1e6), np.arange(1e6)

In [4]: loop_fused(a, b) # compile the function
Out[4]: array([False, False, False, ..., True, True, True])

In [5]: %timeit loop_fused(a, b)
643 µs ± 18 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [6]: %timeit loop_fused.py_func(a, b)
5.2 ms ± 205 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

As you can see, the Numba compiled version is eight times faster than the pure NumPy one. What is going on? Without Numba, the array expression will lead to several `for` loops and several so-called temporaries in memory. Loosely speaking, for each arithmetic operation in the expression, a `for` loop over arrays must execute, and the result of each must be stored in a temporary array in memory. What loop fusion does is fuse the various loops over arithmetic operations together into a single loop, thereby reducing both the total number of memory lookups and the overall memory required to compute the result. In fact, the loop-fused variant may well look something like this:

```
import numpy as np
from numba import njit

@njit
def manual_loop_fused(a, b):
    N = len(a)
    result = np.empty(N, dtype=np.bool_)
    for i in range(N):
        a_i, b_i = a[i], b[i]
        result[i] = a_i * b_i - 4.1 * a_i > 2.5 * b_i
    return result
```

Running this will show performance characteristics similar to those of the loop-fusion example:

```
In [1]: %timeit manual_loop_fused(a, b)
636 µs ± 49.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Finally, I recommend targeting serial execution initially, but keep parallel execution in mind. Don't assume from the outset that only a parallel version will lead to your targeted performance characteristics. Instead, focus on developing a clean serial implementation first. Parallelism makes everything harder to reason about and can be a source of difficulty when debugging problems. If you are satisfied with your results and would still like to investigate parallelizing your code, Numba does come with a `parallel=True` option for the `@jit` decorator and a corresponding parallel range, the `prange` construct, to make creating parallel loops easier.

Getting Help

As of early 2020, the two main recommended communication channels for Numba are the [GitHub issue tracker](#) and the [Gitter chat](#); this is where the action happens. There are also a mailing list and a Twitter account, but these are fairly low-traffic and mostly used to announce new releases and other important project news.

Optimizing Versus Thinking

Vincent D. Warmerdam, Senior Person at GoDataDriven (<http://koaning.io>)

This is a story of a team that was solving the wrong problem. We were optimizing efficiency while ignoring effectiveness. My hope is that this story is a cautionary tale for others. This story actually happened, but I've changed parts and kept the details vague in the interest of keeping things incognito.

I was consulting for a client with a common logistics problem: they wanted to predict the number of trucks that would arrive at their warehouses. There was a good business case for this. If we knew the number of vehicles, we would know how large the workforce had to be to handle the workload for that day.

The planning department had been trying to tackle this problem for years (using Excel). They were skeptical that an algorithm could improve things. Our job was to explore if machine learning could help out here.

From the start, it was apparent it was a difficult time-series problem:

- There were many (seriously, many!) holidays that we needed to keep in mind since the warehouses operated internationally. The effect of the holiday might depend on the day of the week, since the warehouses did not open during weekends. Certain holidays meant demand would go up, while other holidays meant that the warehouses were closed (which would sometimes cause a three-day weekend).
- Seasonal shifts were not unheard of.
- Suppliers would frequently enter and leave the market.
- The seasonal patterns were always changing because the market was continuously evolving.
- There were many warehouses, and although they were in separate buildings, there was a reason to believe the number of trucks arriving at the different warehouses were correlated.

The diagram in [Figure 12-1](#) shows the process for the algorithm that would calculate the seasonal effect as well as the long-term trend. As long as there weren't any holidays, our method would work. The planning department warned us about this; the holidays were the hard part. After spending a lot of time collecting relevant features, we ended up building a system that mainly focused on trying to deal with the holidays.

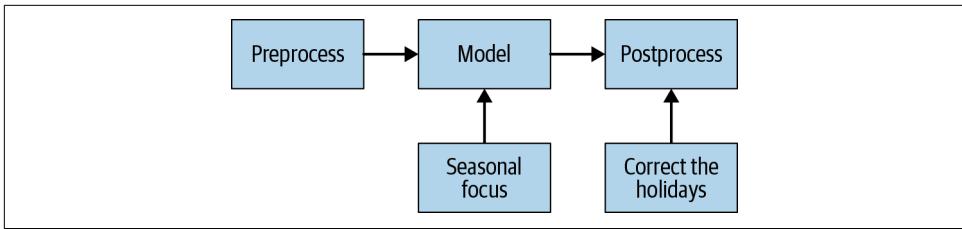


Figure 12-1. Seasonal effects and long-term trend

So we iterated, did more feature engineering, and designed the algorithm. It got to the point where we needed to calculate a time-series model per warehouse, which would be post-processed with a heuristic model per holiday per day of the week. A holiday just before the weekend would cause a different shift than a holiday just after the weekend. As you might imagine, this calculation can get quite expensive when you also want to apply a grid search, as shown in [Figure 12-2](#).

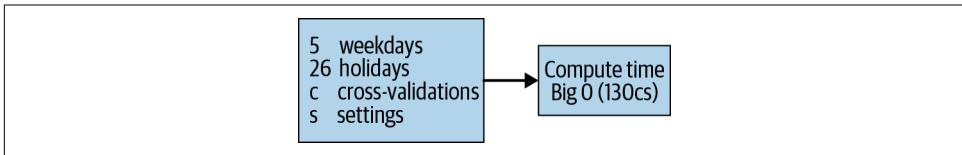


Figure 12-2. Many variations cost a lot of compute time

There were many effects that we had to estimate accurately, including the decay of the past measurements, how smooth the seasonal effect is, the regularization parameters, and how to tackle the correlation between different warehouses.

What didn't help was that we needed to predict months ahead. Another hard thing was the cost function: it was discrete. The planning department did not care about (or even appreciate) mean squared error; they cared only about the number of days where the prediction error would exceed 100 trucks.

You can imagine that, in addition to statistical concerns, the model presented performance concerns. To keep this at bay, we restricted ourselves to simpler machine learning models. We gained a lot of iteration speed by doing this, which allowed us to focus on feature engineering. A few weeks went by before we had a version we could demo. We had still made a model that performed well enough, except for the holidays.

The model went into a proof-of-concept phase; it performed reasonably well but not significantly better than the current planning team's method. The model was useful since it allowed the planning department to reflect if the model disagreed, but no one was comfortable having the model automate the planning.

Then it happened. It was my final week with the client, just before a colleague would be taking over. I was hanging out at the coffee corner, talking with an analyst about a different project for which I needed some data from him. We started reviewing the available tables in the database. Eventually, he told me about a “carts” table (shown in Figure 12-3).

Me: “A carts table? What’s in there?” Analyst: “Oh, it contains all the orders for the carts.” Me: “Suppliers buy them from the warehouse?” Analyst: “No, actually, they rent them. They usually rent them three to five days in advance before they return them filled with goods for the warehouse.” Me: “All of your suppliers work this way?” Analyst: “Pretty much.”

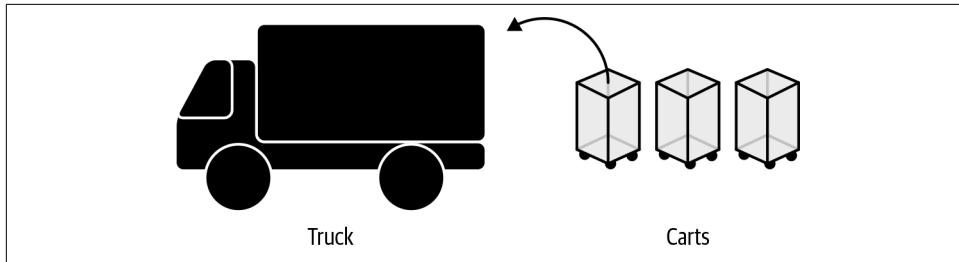


Figure 12-3. Carts contain the leading information that’s critical to this challenge!

I spotted the most significant performance issue of them all: we were solving the wrong problem. This wasn’t a machine learning problem; it was a SQL problem. The number of rented carts was a robust proxy for how many trucks the company would send. They wouldn’t need a machine learning model. We could just project the number of carts being rented a few days ahead, divide by the number of carts that fit into a truck, and get a sensible approximation of what to expect. If we had realized this earlier, we would not have needed to optimize the code for a gigantic grid search because there would have been no need.

It is rather straightforward to translate a business case into an analytical problem that doesn’t reflect reality. Anything that you can do to prevent this will yield the most significant performance benefit you can imagine.

Adaptive Lab’s Social Media Analytics (2014)

Ben Jackson (adaptivelab.com)

Adaptive Lab is a product development and innovation company based in London’s Tech City area, Shoreditch. We apply our lean, user-centric method of product design and delivery collaboratively with a wide range of companies, from start-ups to large corporates.

YouGov is a global market research company whose stated ambition is to supply a live stream of continuous, accurate data and insight into what people are thinking and doing all over the world—and that's just what we managed to provide for them. Adaptive Lab designed a way to listen passively to real discussions happening in social media and gain insight into users' feelings on a customizable range of topics. We built a scalable system capable of capturing a large volume of streaming information, processing it, storing it indefinitely, and presenting it through a powerful, filterable interface in real time. The system was built using Python.

Python at Adaptive Lab

Python is one of our core technologies. We use it in performance-critical applications and whenever we work with clients that have in-house Python skills, so that the work we produce for them can be taken on in-house.

Python is ideal for small, self-contained, long-running daemons, and it's just as great with flexible, feature-rich web frameworks like Django and Pyramid. The Python community is thriving, which means that there's a huge library of open source tools out there that allow us to build quickly and with confidence, leaving us to focus on the new and innovative stuff, solving problems for users.

Across all of our projects, we at Adaptive Lab reuse several tools that are built in Python but that can be used in a language-agnostic way. For example, we use Salt-Stack for server provisioning and Mozilla's Circus for managing long-running processes. The benefit to us when a tool is open source and written in a language we're familiar with is that if we find any problems, we can solve them ourselves and get those solutions taken up, which benefits the community.

SoMA's Design

Our Social Media Analytics (SoMA) tool needed to cope with a high throughput of social media data and the storage and retrieval in real time of a large amount of information. After researching various data stores and search engines, we settled on Elasticsearch as our real-time document store. As its name suggests, it's highly scalable, but it is also easy to use and is capable of providing statistical responses as well as search—ideal for our application. Elasticsearch itself is built in Java, but like any well-architected component of a modern system, it has a good API and is well catered to with a Python library and tutorials.

The system we designed uses queues with Celery held in Redis to quickly hand a large stream of data to any number of servers for independent processing and indexing. Each component of the whole complex system was designed to be small, individually simple, and able to work in isolation. Each focused on one task, like analyzing a conversation for sentiment or preparing a document for indexing into Elasticsearch. Several of these were configured to run as daemons using Mozilla's Circus, which keeps

all the processes up and running and allows them to be scaled up or down on individual servers.

SaltStack is used to define and provision the complex cluster and handles the setup of all of the libraries, languages, databases, and document stores. We also make use of Fabric, a Python tool for running arbitrary tasks on the command line. Defining servers in code has many benefits: complete parity with the production environment; version control of the configuration; having everything in one place. It also serves as documentation on the setup and dependencies required by a cluster.

Our Development Methodology

We aim to make it as easy as possible for a newcomer to a project to be able to quickly get into adding code and deploying confidently. We use Vagrant to build the complexities of a system locally, inside a virtual machine that has complete parity with the production environment. A simple `vagrant up` is all a newcomer needs to get set up with all the dependencies required for their work.

We work in an agile way, planning together, discussing architecture decisions, and determining a consensus on task estimates. For SoMA, we made the decision to include at least a few tasks considered as corrections for “technical debt” in each sprint. Also included were tasks for documenting the system (we eventually established a wiki to house all the knowledge for this ever-expanding project). Team members review each other’s code after each task, to sanity check, offer feedback, and understand the new code that is about to get added to the system.

A good test suite helped bolster confidence that any changes weren’t going to cause existing features to fail. Integration tests are vital in a system like SoMA, composed of many moving parts. A staging environment offers a way to test the performance of new code; on SoMA in particular, it was only through testing against the kind of large datasets seen in production that problems could occur and be dealt with, so it was often necessary to reproduce that amount of data in a separate environment. Amazon’s Elastic Compute Cloud (EC2) gave us the flexibility to do this.

Maintaining SoMA

The SoMA system runs continuously, and the amount of information it consumes grows every day. We have to account for peaks in the data stream, network issues, and problems in any of the third-party service providers it relies on. So, to make things easy on ourselves, SoMA is designed to fix itself whenever it can. Thanks to Circus, processes that crash out will come back to life and resume their tasks from where they left off. A task will queue up until a process can consume it, and there’s enough breathing room there to stack up tasks while the system recovers.

We use Server Density to monitor the many SoMA servers. It's very simple to set up, but quite powerful. A nominated engineer can receive a push message via phone as soon as a problem is likely to occur, in order to react in time to ensure it doesn't become a problem. With Server Density, it's also very easy to write custom plug-ins in Python, making it possible, for example, to set up instant alerts on aspects of Elasticsearch's behavior.

Advice for Fellow Engineers

Above all, you and your team need to be confident and comfortable that what is about to be deployed into a live environment is going to work flawlessly. To get to that point, you have to work backward, spending time on all of the components of the system that will give you that sense of comfort. Make deployment simple and foolproof; use a staging environment to test the performance with real-world data; ensure you have a good, solid test suite with high coverage; implement a process for incorporating new code into the system; and make sure technical debt gets addressed sooner rather than later. The more you shore up your technical infrastructure and improve your processes, the happier and more successful at engineering the right solutions your team will be.

If a solid foundation of code and ecosystem are not in place but the business is pressuring you to get things live, it's only going to lead to problem software. It's going to be your responsibility to push back and stake out time for incremental improvements to the code, and the tests and operations involved in getting things out the door.

Making Deep Learning Fly with RadimRehurek.com (2014)

Radim Řehůřek (radimrehurek.com)

When Ian asked me to write my “lessons from the field” on Python and optimizations for this book, I immediately thought, “Tell them how you made a Python port faster than Google’s C original!” It’s an inspiring story of making a machine learning algorithm, Google’s poster child for deep learning, 12,000× faster than a naive Python implementation. Anyone can write bad code and then trumpet about large speedups. But the optimized Python port also runs, somewhat astonishingly, almost four times faster than the original code written by Google’s team! That is, four times faster than opaque, tightly profiled, and optimized C.

But before drawing “machine-level” optimization lessons, some general advice about “human-level” optimizations.

The Sweet Spot

I run a small consulting business laser-focused on machine learning, where my colleagues and I help companies make sense of the tumultuous world of data analysis, in order to make money or save costs (or both). We help clients design and build wondrous systems for data processing, especially text data.

The clients range from large multinationals to nascent start-ups, and while each project is different and requires a different tech stack, plugging into the client's existing data flows and pipelines, Python is a clear favorite. Not to preach to the choir, but Python's no-nonsense development philosophy, its malleability, and the rich library ecosystem make it an ideal choice.

First, a few thoughts "from the field" on what works:

Communication, communication, communication

This one's obvious, but worth repeating. Understand the client's problem on a higher (business) level before deciding on an approach. Sit down and talk through what they think they need (based on their partial knowledge of what's possible and/or what they Googled up before contacting you), until it becomes clear what they really need, free of cruft and preconceptions. Agree on ways to validate the solution beforehand. I like to visualize this process as a long, winding road to be built: get the starting line right (problem definition, available data sources) and the finish line right (evaluation, solution priorities), and the path in between falls into place.

Be on the lookout for promising technologies

An emergent technology that is reasonably well understood and robust, is gaining traction, yet is still relatively obscure in the industry can bring huge value to the client (or yourself). As an example, a few years ago, Elasticsearch was a little-known and somewhat raw open source project. But I evaluated its approach as solid (built on top of Apache Lucene, offering replication, cluster sharding, etc.) and recommended its use to a client. We consequently built a search system with Elasticsearch at its core, saving the client significant amounts of money in licensing, development, and maintenance compared to the considered alternatives (large commercial databases). Even more importantly, using a new, flexible, powerful technology gave the product a massive competitive advantage. Nowadays, Elasticsearch has entered the enterprise market and conveys no competitive advantage at all—everyone knows it and uses it. Getting the timing right is what I call hitting the "sweet spot," maximizing the value/cost ratio.

KISS (Keep It Simple, Stupid!)

This is another no-brainer. The best code is code you don't have to write and maintain. Start simple, and improve and iterate where necessary. I prefer tools that follow the Unix philosophy of "do one thing, and do it well." Grand pro-

gramming frameworks can be tempting, with everything imaginable under one roof and fitting neatly together. But invariably, sooner or later, you need something the grand framework didn't imagine, and then even modifications that seem simple (conceptually) cascade into a nightmare (programmatically). Grand projects and their all-encompassing APIs tend to collapse under their own weight. Use modular, focused tools, with APIs in between that are as small and uncomplicated as possible. Prefer text formats that are open to simple visual inspection, unless performance dictates otherwise.

Use manual sanity checks in data pipelines

When optimizing data processing systems, it's easy to stay in the "binary mind-set" mode, using tight pipelines, efficient binary data formats, and compressed I/O. As the data passes through the system unseen, and unchecked (except for perhaps its type), it remains invisible until something outright blows up. Then debugging commences. I advocate sprinkling a few simple log messages throughout the code, showing what the data looks like at various internal points of processing, as good practice—nothing fancy, just an analogy to the Unix `head` command, picking and visualizing a few data points. Not only does this help during the aforementioned debugging, but seeing the data in a human-readable format leads to "aha!" moments surprisingly often, even when all seems to be going well. Strange tokenization! They promised input would always be encoded in latin1! How did a document in this language get in there? Image files leaked into a pipeline that expects and parses text files! These are often insights that go way beyond those offered by automatic type checking or a fixed unit test, hinting at issues beyond component boundaries. Real-world data is messy. Catch early even things that wouldn't necessarily lead to exceptions or glaring errors. Err on the side of too much verbosity.

Navigate fads carefully

Just because a client keeps hearing about X and says they must have X too doesn't mean they really need it. It might be a marketing problem rather than a technology one, so take care to discern the two and deliver accordingly. X changes over time as hype waves come and go; a recent value would be X = big data.

All right, enough business talk—here's how I got *word2vec* in Python to run faster than C.

Lessons in Optimizing

word2vec is a deep learning algorithm that allows detection of similar words and phrases. With interesting applications in text analytics and search engine optimization (SEO), and with Google's lustrous brand name attached to it, start-ups and businesses flocked to take advantage of this new tool.

Unfortunately, the only available code was that produced by Google itself, an open source Linux command-line tool written in C. This was a well-optimized but rather hard-to-use implementation. The primary reason I decided to port *word2vec* to Python was so I could extend *word2vec* to other platforms, making it easier to integrate and extend for clients.

The details are not relevant here, but *word2vec* requires a training phase with a lot of input data to produce a useful similarity model. For example, the folks at Google ran *word2vec* on their GoogleNews dataset, training on approximately 100 billion words. Datasets of this scale obviously don't fit in RAM, so a memory-efficient approach must be taken.

I've authored a machine learning library, [gensim](#), that targets exactly that sort of memory-optimization problem: datasets that are no longer trivial ("trivial" being anything that fits fully into RAM), yet not large enough to warrant petabyte-scale clusters of MapReduce computers. This "terabyte" problem range fits a surprisingly large portion of real-world cases, *word2vec* included.

Details are described [on my blog](#), but here are a few optimization takeaways:

Stream your data, watch your memory

Let your input be accessed and processed one data point at a time, for a small, constant memory footprint. The streamed data points (sentences, in the case of *word2vec*) may be grouped into larger batches internally for performance (such as processing 100 sentences at a time), but a high-level, streamed API proved a powerful and flexible abstraction. The Python language supports this pattern very naturally and elegantly, with its built-in generators—a truly beautiful problem-tech match. Avoid committing to algorithms and tools that load everything into RAM, unless you know your data will always remain small, or you don't mind reimplementing a production version yourself later.

Take advantage of Python's rich ecosystem

I started with a readable, clean port of *word2vec* in `numpy`. `numpy` is covered in depth in [Chapter 6](#) of this book, but as a short reminder, it is an amazing library, a cornerstone of Python's scientific community and the de facto standard for number crunching in Python. Tapping into `numpy`'s powerful array interfaces, memory access patterns, and wrapped BLAS routines for ultrafast common vector operations leads to concise, clean, and fast code—code that is hundreds of times faster than naive Python code. Normally I'd call it a day at this point, but "hundreds of times faster" was still 20× slower than Google's optimized C version, so I pressed on.

Profile and compile hotspots

word2vec is a typical high performance computing app, in that a few lines of code in one inner loop account for 90% of the entire training runtime. Here I rewrote

a single core routine (approximately 20 lines of code) in C, using an external Python library, Cython, as the glue. While it's technically brilliant, I don't consider Cython a particularly convenient tool conceptually—it's basically like learning another language, a nonintuitive mix between Python, `numpy`, and C, with its own caveats and idiosyncrasies. But until Python's JIT technologies mature, Cython is probably our best bet. With a Cython-compiled hotspot, performance of the Python `word2vec` port is now on par with the original C code. An additional advantage of having started with a clean `numpy` version is that we get free tests for correctness, by comparing against the slower but correct version.

Know your BLAS

A neat feature of `numpy` is that it internally wraps Basic Linear Algebra Subprograms (BLAS), where available. These are sets of low-level routines, optimized directly by processor vendors (Intel, AMD, etc.) in assembly, Fortran, or C, and designed to squeeze out maximum performance from a particular processor architecture. For example, calling an `axpy` BLAS routine computes `vector_y += scalar * vector_x` way faster than what a generic compiler would produce for an equivalent explicit `for` loop. Expressing `word2vec` training as BLAS operations resulted in another 4 \times speedup, topping the performance of C `word2vec`. Victory! To be fair, the C code could link to BLAS as well, so this is not some inherent advantage of Python per se. `numpy` just makes things like this stand out and makes them easy to take advantage of.

Parallelization and multiple cores

`gensim` contains distributed cluster implementations of a few algorithms. For `word2vec`, I opted for multithreading on a single machine, because of the fine-grained nature of its training algorithm. Using threads also allows us to avoid the fork-without-exec POSIX issues that Python's multiprocessing brings, especially in combination with certain BLAS libraries. Because our core routine is already in Cython, we can afford to release Python's GIL (global interpreter lock; see “[Parallelizing the Solution with OpenMP on One Machine](#)” on page 178), which normally renders multithreading useless for CPU-intensive tasks. Speedup: another 3 \times , on a machine with four cores.

Static memory allocations

At this point, we're processing tens of thousands of sentences per second. Training is so fast that even little things like creating a new `numpy` array (calling `malloc` for each streamed sentence) slow us down. Solution: preallocate a static “work” memory and pass it around, in good old Fortran fashion. Brings tears to my eyes. The lesson here is to keep as much bookkeeping and app logic in the clean Python code as possible, and to keep the optimized hotspot lean and mean.

Problem-specific optimizations

The original C implementation contained specific micro-optimizations, such as aligning arrays onto specific memory boundaries or precomputing certain functions into memory lookup tables. A nostalgic blast from the past, but with today's complex CPU instruction pipelines, memory cache hierarchies, and coprocessors, such optimizations are no longer a clear winner. Careful profiling suggested a few percent improvement, which may not be worth the extra code complexity. Takeaway: use annotation and profiling tools to highlight poorly optimized spots. Use your domain knowledge to introduce algorithmic approximations that trade accuracy for performance (or vice versa). But never take it on faith; profile, preferably using real production data.

Conclusion

Optimize where appropriate. In my experience, there's never enough communication to fully ascertain the problem scope, priorities, and connection to the client's business goals—a.k.a. the "human-level" optimizations. Make sure you deliver on a problem that matters, rather than getting lost in "geek stuff" for the sake of it. And when you do roll up your sleeves, make it worth it!

Large-Scale Productionized Machine Learning at Lyst.com (2014)

Sebastjan Trepca ([lyst.com](#))

Lyst.com is a fashion recommendation engine based in London; it has over 2 million monthly users who learn about new fashion through Lyst's scraping, cleaning, and modeling processes. Founded in 2010, it has raised \$20 million of investment.

Sebastjan Trepca was the technical founder and is the CTO; he created the site using Django, and Python has helped the team to quickly test new ideas.

Python and Django have been at the heart of Lyst since the site's creation. As internal projects have grown, some of the Python components have been replaced with other tools and languages to fit the maturing needs of the system.

Cluster Design

The cluster runs on Amazon EC2. In total there are approximately 100 machines, including the more recent C3 instances, which have good CPU performance.

Redis is used for queuing with PyRes and storing metadata. The dominant data format is JSON, for ease of human comprehension. `supervisord` keeps the processes alive.

Elasticsearch and PyES are used to index all products. The Elasticsearch cluster stores 60 million documents across seven machines. Solr was investigated but discounted because of its lack of real-time updating features.

Code Evolution in a Fast-Moving Start-Up

It is better to write code that can be implemented quickly so that a business idea can be tested than to spend a long time attempting to write “perfect code” in the first pass. If code is useful, it can be refactored; if the idea behind the code is poor, it is cheap to delete it and remove a feature. This can lead to a complicated code base with many objects being passed around, but this is acceptable as long as the team makes time to refactor code that is useful to the business.

Docstrings are used heavily in Lyst—an external Sphinx documentation system was tried but dropped in favor of just reading the code. A wiki is used to document processes and larger systems. We also started creating very small services instead of chucking everything into one code base.

Building the Recommendation Engine

At first the recommendation engine was coded in Python, using `numpy` and `scipy` for computations. Subsequently, performance-critical parts of the recommender were sped up using Cython. The core matrix factorization operations were written entirely in Cython, yielding an order of magnitude improvement in speed. This was mostly due to the ability to write performant loops over `numpy` arrays in Python, something that is extremely slow in pure Python and performed poorly when vectorized because it necessitated memory copies of `numpy` arrays. The culprit was `numpy`’s fancy indexing, which always makes a data copy of the array being sliced: if no data copy is necessary or intended, Cython loops will be far faster.

Over time, the online components of the system (responsible for computing recommendations at request time) were integrated into our search component, Elasticsearch. In the process, they were translated into Java to allow full integration with Elasticsearch. The main reason behind this was not performance, but the utility of integrating the recommender with the full power of a search engine, allowing us to apply business rules to served recommendations more easily. The Java component itself is extremely simple and implements primarily efficient sparse vector inner products. The more complex offline component remains written in Python, using standard components of the Python scientific stack (mostly Python and Cython).

In our experience, Python is useful as more than a prototyping language: the availability of tools such as `numpy`, `Cython`, and `weave` (and more recently `Numba`) allowed us to achieve very good performance in the performance-critical parts of the code while maintaining Python’s clarity and expressiveness where low-level optimization would be counterproductive.

Reporting and Monitoring

Graphite is used for reporting. Currently, performance regressions can be seen by eye after a deployment. This makes it easy to drill into detailed event reports or to zoom out and see a high-level report of the site’s behavior, adding and removing events as necessary.

Internally, a larger infrastructure for performance testing is being designed. It will include representative data and use cases to properly test new builds of the site.

A staging site will also be used to let a small fraction of real visitors see the latest version of the deployment—if a bug or performance regression is seen, it will have affected only a minority of visitors, and this version can quickly be retired. This will make the deployment of bugs significantly less costly and problematic.

Sentry is used to log and diagnose Python stack traces.

Jenkins is used for continuous integration (CI) with an in-memory database configuration. This enables parallelized testing so that check-ins quickly reveal any bugs to the developer.

Some Advice

It’s really important to have good tools to track the effectiveness of what you’re building, and to be super-practical at the beginning. Start-ups change constantly, and engineering evolves: you start with a super-exploratory phase, building prototypes all the time and deleting code until you hit the gold mine, and then you start to go deeper, improving code, performance, etc. Until then, it’s all about quick iterations and good monitoring/analytics. I guess this is pretty standard advice that has been repeated over and over, but I think many don’t really get how important it is.

I don’t think technologies matter that much nowadays, so use whatever works for you. I’d think twice before moving to hosted environments like App Engine or Heroku, though.

Large-Scale Social Media Analysis at Smesh (2014)

Alex Kelly ([sme.sh](#))

At Smesh, we produce software that ingests data from a wide variety of APIs across the web; filters, processes, and aggregates it; and then uses that data to build bespoke apps for a variety of clients. For example, we provide the tech that powers the tweet filtering and streaming in Beamly's second-screen TV app, run a brand and campaign monitoring platform for mobile network EE, and run a bunch of Adwords data analysis projects for Google.

To do that, we run a variety of streaming and polling services, frequently polling Twitter, Facebook, YouTube, and a host of other services for content and processing several million tweets daily.

Python's Role at Smesh

We use Python extensively—the majority of our platform and services are built with it. The wide variety of libraries, tools, and frameworks available allows us to use it across the board for most of what we do.

That variety gives us the ability to (hopefully) pick the right tool for the job. For example, we've created apps using Django, Flask, and Pyramid. Each has its own benefits, and we can pick the one that's right for the task at hand. We use Celery for tasks; Boto for interacting with AWS; and PyMongo, MongoEngine, redis-py, Psycopg, etc. for all our data needs. The list goes on and on.

The Platform

Our main platform consists of a central Python module that provides hooks for data input, filtering, aggregations and processing, and a variety of other core functions. Project-specific code imports functionality from that core and then implements more specific data processing and view logic, as each application requires.

This has worked well for us up to now, and allows us to build fairly complex applications that ingest and process data from a wide variety of sources without much duplication of effort. However, it isn't without its drawbacks—each app is dependent on a common core module, making the process of updating the code in that module and keeping all the apps that use it up-to-date a major task.

We're currently working on a project to redesign that core software and move toward more of a service-oriented architecture (SoA) approach. It seems that finding the right time to make that sort of architectural change is one of the challenges that faces most software teams as a platform grows. There is overhead in building components as individual services, and often the deep domain-specific knowledge required to build each service is acquired only through an initial iteration of development, where that architectural overhead is a hindrance to solving the real problem at hand. Hopefully, we've chosen a sensible time to revisit our architectural choices to move things forward. Time will tell.

High Performance Real-Time String Matching

We consume lots of data from the Twitter Streaming API. As we stream in tweets, we match the input strings against a set of keywords so that we know which of the terms we're tracking that each tweet is related to. That's not such a problem with a low rate of input, or a small set of keywords, but doing that matching for hundreds of tweets per second, against hundreds or thousands of possible keywords, starts to get tricky.

To make things even trickier, we're not interested in simply whether the keyword string exists in the tweet, but in more complex pattern matching against word boundaries, start and end of line, and optionally the use of # and @ characters to prefix the string. The most effective way to encapsulate that matching knowledge is using regular expressions. However, running thousands of regex patterns across hundreds of tweets per second is computationally intensive. Previously, we had to run many worker nodes across a cluster of machines to perform the matching reliably in real time.

Knowing this was a major performance bottleneck in the system, we tried a variety of things to improve the performance of our matching system: simplifying the regexes, running enough processes to ensure we were utilizing all the cores on our servers, ensuring all our regex patterns are compiled and cached properly, running the matching tasks under PyPy instead of CPython, etc. Each of these resulted in a small increase in performance, but it was clear this approach was going to shave only a fraction of our processing time. We were looking for an order-of-magnitude speedup, not a fractional improvement.

It was obvious that rather than trying to increase the performance of each match, we needed to reduce the problem space before the pattern matching takes place. So we needed to reduce either the number of tweets to process, or the number of regex patterns we needed to match the tweets against. Dropping the incoming tweets wasn't an option—that's the data we're interested in. So we set about finding a way to reduce the number of patterns we need to compare an incoming tweet to in order to perform the matching.

We started looking at various trie structures for allowing us to do pattern matching between sets of strings more efficiently, and came across the Aho-Corasick string-matching algorithm. It turned out to be ideal for our use case. The dictionary from which the trie is built needs to be static—you can't add new members to the trie after the automaton has been finalized—but for us this isn't a problem, as the set of keywords is static for the duration of a session streaming from Twitter. When we change the terms we're tracking we must disconnect from and reconnect to the API, so we can rebuild the Aho-Corasick trie at the same time.

Processing an input against the strings using Aho-Corasick finds all possible matches simultaneously, stepping through the input string a character at a time and finding

matching nodes at the next level down in the trie (or not, as the case may be). So we can very quickly find which of our keyword terms may exist in the tweet. We still don't know for sure, as the pure string-in-string matching of Aho-Corasick doesn't allow us to apply any of the more complex logic that is encapsulated in the regex patterns, but we can use the Aho-Corasick matching as a prefilter. Keywords that don't exist in the string can't match, so we know we have to try only a small subset of all our regex patterns, based on the keywords that do appear in the text. Rather than evaluating hundreds or thousands of regex patterns against every input, we rule out the majority and need to process only a handful for each tweet.

By reducing the number of patterns that we attempt to match against each incoming tweet to just a small handful, we've managed to achieve the speedup we were looking for. Depending on the complexity of the trie and the average length of the input tweets, our keyword matching system now performs somewhere between 10–100× faster than the original naive implementation.

If you're doing a lot of regex processing, or other pattern matching, I highly recommend having a dig around the different variations of prefix and suffix tries that might help you to find a blazingly fast solution to your problem.

Reporting, Monitoring, Debugging, and Deployment

We maintain a bunch of different systems running our Python software and the rest of the infrastructure that powers it all. Keeping it all up and running without interruption can be tricky. Here are a few lessons we've learned along the way.

It's really powerful to be able to see both in real time and historically what's going on inside your systems, whether that be in your own software or the infrastructure it runs on. We use Graphite with `collectd` and `statsd` to allow us to draw pretty graphs of what's going on. That gives us a way to spot trends, and to retrospectively analyze problems to find the root cause. We haven't got around to implementing it yet, but Etsy's Skyline also looks brilliant as a way to spot the unexpected when you have more metrics than you can keep track of. Another useful tool is Sentry, a great system for event logging and keeping track of exceptions being raised across a cluster of machines.

Deployment can be painful, no matter what you're using to do it. We've been users of Puppet, Ansible, and Salt. They all have pros and cons, but none of them will make a complex deployment problem magically go away.

To maintain high availability for some of our systems, we run multiple geographically distributed clusters of infrastructure, running one system live and others as hot spares, with switchover being done by updates to DNS with low Time-to-Live (TTL) values. Obviously that's not always straightforward, especially when you have tight constraints on data consistency. Thankfully, we're not affected by that too badly,

making the approach relatively straightforward. It also provides us with a fairly safe deployment strategy, updating one of our spare clusters and performing testing before promoting that cluster to live and updating the others.

Along with everyone else, we're really excited by the prospect of what can be done with **Docker**. Also along with pretty much everyone else, we're still just at the stage of playing around with it to figure out how to make it part of our deployment processes. However, having the ability to rapidly deploy our software in a lightweight and reproducible fashion, with all its binary dependencies and system libraries included, seems to be just around the corner.

At a server level, there's a whole bunch of routine stuff that just makes life easier. Monit is great for keeping an eye on things for you. Upstart and **supervisord** make running services less painful. Munin is useful for some quick and easy system-level graphing if you're not using a full Graphite/collectd setup. And Corosync/Pacemaker can be a good solution for running services across a cluster of nodes (for example, when you have a bunch of services that you need to run somewhere, but not everywhere).

I've tried not to just list buzzwords here, but to point you toward software we're using every day, which is really making a difference in how effectively we can deploy and run our systems. If you've heard of them all already, I'm sure you must have a whole bunch of other useful tips to share, so please drop me a line with some pointers. If not, go check them out—hopefully, some of them will be as useful to you as they are to us.

PyPy for Successful Web and Data Processing Systems (2014)

Marko Tasic (<https://github.com/mtasic85>)

Since I had a great experience early on with PyPy, I chose to use it everywhere where it was applicable. I have used it from small toy projects where speed was essential to medium-sized projects. The first project where I used it was a protocol implementation; the protocols we implemented were Modbus and DNP3. Later, I used it for a compression algorithm implementation, and everyone was amazed by its speed. The first version I used in production was PyPy 1.2 with JIT out of the box, if I recall correctly. By version 1.4, we were sure it was the future of all our projects, because many bugs got fixed and the speed just increased more and more. We were surprised how simple cases were made 2–3× faster just by upgrading PyPy up to the next version.

I will explain two separate but deeply related projects that share 90% of the same code here, but to keep the explanation simple to follow, I will refer to both of them as “the project.”

The project was to create a system that collects newspapers, magazines, and blogs, applies optical character recognition (OCR) if necessary, classifies them, translates, applies sentiment analyzing, analyzes the document structure, and indexes them for later search. Users can search for keywords in any of the available languages and retrieve information about indexed documents. Search is cross-language, so users can write in English and get results in French. Additionally, users will receive articles and keywords highlighted from the document's page with information about the space occupied and price of publication. A more advanced use case would be report generation, where users can see a tabular view of results with detailed information on spending by any particular company on advertising in monitored newspapers, magazines, and blogs. As well as advertising, it can also “guess” if an article is paid or objective and determine its tone.

Prerequisites

Obviously, PyPy was our favorite Python implementation. For the database, we used Cassandra and Elasticsearch. Cache servers used Redis. We used Celery as a distributed task queue (workers), and for its broker, we used RabbitMQ. Results were kept in a Redis backend. Later on, Celery used Redis more exclusively for both brokers and backend. The OCR engine used is Tesseract. The language translation engine and server used is Moses. We used Scrapy for crawling websites. For distributed locking in the whole system, we use a ZooKeeper server, but initially Redis was used for that. The web application is based on the excellent Flask web framework and many of its extensions, such as Flask-Login, Flask-Principal, etc. The Flask application was hosted by Gunicorn and Tornado on every web server, and nginx was used as a reverse proxy server for the web servers. The rest of the code was written by us and is pure Python that runs on top of PyPy.

The whole project is hosted on an in-house OpenStack private cloud and executes between 100 and 1,000 instances of ArchLinux, depending on requirements, which can change dynamically on the fly. The whole system consumes up to 200 TB of storage every 6–12 months, depending on the mentioned requirements. All processing is done by our Python code, except OCR and translation.

The Database

We developed a Python package that unifies model classes for Cassandra, Elasticsearch, and Redis. It is a simple object relational mapper (ORM) that maps everything to a dict or list of dicts, in the case where many records are retrieved from the database.

Since Cassandra 1.2 did not support complex queries on indices, we supported them with join-like queries. However, we allowed complex queries over small datasets (up to 4 GB) because much of that had to be processed while held in memory. PyPy ran in cases where CPython could not even load data into memory, thanks to its strategies applied to homogeneous lists to make them more compact in the memory. Another benefit of PyPy is that its JIT compilation kicked in loops where data manipulation or analysis happened. We wrote code in such a way that the types would stay static inside loops because that's where JIT-compiled code is especially good.

Elasticsearch was used for indexing and fast searching of documents. It is very flexible when it comes to query complexity, so we did not have any major issues with it. One of the issues we had was related to updating documents; it is not designed for rapidly changing documents, so we had to migrate that part to Cassandra. Another limitation was related to facets and memory required on the database instance, but that was solved by having more smaller queries and then manually manipulating data in Celery workers. No major issues surfaced between PyPy and the PyES library used for interaction with Elasticsearch server pools.

The Web Application

As we've mentioned, we used the Flask framework with its third-party extensions. Initially, we started everything in Django, but we switched to Flask because of rapid changes in requirements. This does not mean that Flask is better than Django; it was just easier for us to follow code in Flask than in Django, since its project layout is very flexible. Gunicorn was used as a Web Server Gateway Interface (WSGI) HTTP server, and its I/O loop was executed by Tornado. This allowed us to have up to one hundred concurrent connections per web server. This was lower than expected because many user queries can take a long time—a lot of analyzing happens in user requests, and data is returned in user interactions.

Initially, the web application depended on the Python Imaging Library (PIL) for article and word highlighting. We had issues with the PIL library and PyPy because at that time many memory leaks were associated with PIL. Then we switched to Pillow, which was more frequently maintained. In the end, we wrote a library that interacted with GraphicsMagick via a subprocess module.

PyPy runs well, and the results are comparable with CPython. This is because usually web applications are I/O-bound. However, with the development of STM in PyPy, we hope to have scalable event handling on a multicore instance level soon.

OCR and Translation

We wrote pure Python libraries for Tesseract and Moses because we had problems with CPython API-dependent extensions. PyPy has good support for the CPython API using CPyExt, but we wanted to be more in control of what happens under the hood. As a result, we made a PyPy-compatible solution with slightly faster code than on CPython. The reason it was not faster is that most of the processing happened in the C/C++ code of both Tesseract and Moses. We could only speed up output processing and building Python structure of documents. There were no major issues at this stage with PyPy compatibility.

Task Distribution and Workers

Celery gave us the power to run many tasks in the background. Typical tasks are OCR, translation, analysis, etc. The whole thing could be done using Hadoop for MapReduce, but we chose Celery because we knew that the project requirements might change often.

We had about 20 workers, and each worker had between 10 and 20 functions. Almost all functions had loops, or many nested loops. We cared that types stayed static, so the JIT compiler could do its job. The end results were a 2–5× speedup over CPython. The reason we did not get better speedups was because our loops were relatively small, between 20,000 and 100,000 iterations. In some cases where we had to do analysis on the word level, we had over 1 million iterations, and that's where we got over a 10× speedup.

Conclusion

PyPy is an excellent choice for every pure Python project that depends on speed of execution of readable and maintainable large source code. We found PyPy also to be very stable. All our programs were long-running with static and/or homogeneous types inside data structures, so JIT could do its job. When we tested the whole system on CPython, the results did not surprise us: we had roughly a 2× speedup with PyPy over CPython. In the eyes of our clients, this meant 2× better performance for the same price. In addition to all the good stuff that PyPy has brought to us so far, we hope that its software transactional memory (STM) implementation will bring to us scalable parallel execution for Python code.

Task Queues at Lanyrd.com (2014)

Andrew Godwin

Lanyrd is a website for social discovery of conferences—our users sign in, and we use their friend graphs from social networks, as well as other indicators like their industry of work or their geographic location, to suggest relevant conferences.

The main work of the site is in distilling this raw data down into something we can show to the users—essentially, a ranked list of conferences. We have to do this offline, because we refresh the list of recommended conferences every couple of days and because we’re hitting external APIs that are often slow. We also use the Celery task queue for other things that take a long time, like fetching thumbnails for links people provide and sending email. There are usually well over 100,000 tasks in the queue each day, and sometimes many more.

Python’s Role at Lanyrd

Lanyrd was built with Python and Django from day one, and virtually every part of it is written in Python—the website itself, the offline processing, our statistical and analysis tools, our mobile backend servers, and the deployment system. It’s a very versatile and mature language and one that’s incredibly easy to write things in quickly, mostly thanks to the large amount of libraries available and the language’s easily readable and concise syntax, which means it’s easy to update and refactor as well as easy to write initially.

The Celery task queue was already a mature project when we evolved the need for a task queue (very early on), and the rest of Lanyrd was already in Python, so it was a natural fit. As we grew, there was a need to change the queue that backed it (which ended up being Redis), but it’s generally scaled very well.

As a start-up, we had to ship some known technical debt in order to make some headway—this is something you just have to do, and as long as you know what your issues are and when they might surface, it’s not necessarily a bad thing. Python’s flexibility in this regard is fantastic; it generally encourages loose coupling of components, which means it’s often easy to ship something with a “good enough” implementation and then easily refactor a better one in later.

Anything critical, such as payment code, had full unit test coverage, but for other parts of the site and task queue flow (especially display-related code) things were often moving too fast to make unit tests worthwhile (they would be too fragile). Instead, we adopted a very agile approach and had a two-minute deploy time and excellent error tracking; if a bug made it into live, we could often fix it and deploy within five minutes.

Making the Task Queue Performant

The main issue with a task queue is throughput. If it gets backlogged, the website keeps working but starts getting mysteriously outdated—lists don’t update, page content is wrong, and emails don’t get sent for hours.

Fortunately, though, task queues also encourage a very scalable design; as long as your central messaging server (in our case, Redis) can handle the messaging overhead of the job requests and responses, for the actual processing you can spin up any number of worker daemons to handle the load.

Reporting, Monitoring, Debugging, and Deployment

We had monitoring that kept track of our queue length, and if it started becoming long we would just deploy another server with more worker daemons. Celery makes this very easy to do. Our deployment system had hooks where we could increase the number of worker threads on a box (if our CPU utilization wasn’t optimal) and could easily turn a fresh server into a Celery worker within 30 minutes. It’s not like website response times going through the floor—if your task queues suddenly get a load spike, you have some time to implement a fix and usually it’ll smooth over itself, if you’ve left enough spare capacity.

Advice to a Fellow Developer

My main advice is to shove as much as you can into a task queue (or a similar loosely coupled architecture) as soon as possible. It takes some initial engineering effort, but as you grow, operations that used to take half a second can grow to half a minute, and you’ll be glad they’re not blocking your main rendering thread. Once you’ve got there, make sure you keep a close eye on your average queue latency (how long it takes a job to go from submission to completion), and make sure there’s some spare capacity for when your load increases.

Finally, be aware that having multiple task queues for different priorities of tasks makes sense. Sending email isn’t very high priority; people are used to emails taking minutes to arrive. However, if you’re rendering a thumbnail in the background and showing a spinner while you do it, you want that job to be high priority, as otherwise you’re making the user experience worse. You don’t want your 100,000-person mailshot to delay all thumbnailing on your site for the next 20 minutes!

Index

Symbols

%memit, 343, 352
%timeit function, 22, 32, 58, 158
-p portability flag, 33
/ (integer division/float division), xvi
1D diffusion, 112
2D diffusion, 114
32-bit Python, xv
64-bit Python, xv
>> (jump points), 57
@jit decorator, 180, 404, 405, 409
@profile decorator, 60

A

abs function, 165, 174
Accelerate, 295
Adaptive Lab, 412-415
Aho-Corasick trie, 424
aiohttp, 229-233
Airflow, 278, 335
algorithms, 67, 100, 375-378
Amazon EC2, 316, 414, 420
Amazon Machine Image, 318
Amazon Web Services (AWS), 311, 423
Amazon's Simple Queue Service (SQS), 335
Amdahl's law, 4, 246
Anaconda Inc., 15, 180, 403
anomaly detection, 105
Ansible, 425
AOT compilers, JIT compilers versus, 164
Apache Spark, 323
API bindings, 15
App Engine, 422
Archibald, Stuart, 403

architectures, 2-9
 communication layers, 8-9
 computing units, 2-5
 constructing, 10-13
 memory units, 5-7
 multi-core, 4
ArchLinux, 427
array (data structure)
 about, 65, 322-324
 (see also lists, tuples and)
 dynamic, 71, 72-76
 static, 71, 76
array module, 14, 125, 128, 176, 200, 344-350
asciidoc, 340
asizeof tool, 351
assert statements, 19
async function, 217
asynchronous I/O, 7
asynchronous job feeding, 276-278
asynchronous programming, 213-244
 aiohttp, 229-233
 async function, 217
 AsyncIO (module), 217, 226, 229, 277
 await statement, 217
 database examples, 233-241
 gevent, 221-225, 228, 243, 278
 serial crawler, 219-220
 tornado, 15, 226-229, 243, 277, 330, 427
asynchronous systems, 277
AsyncIO (module), 14, 217, 226, 229, 277
ATLAS, 295
Auth0, 335
autograd, 190
availability, 402

await statement, 217
AWS (Amazon Web Services), 311, 423
Azure (Microsoft), 316

B

bag, 323
Basic Linear Algebra Subprograms (BLAS), 204, 295, 419
batched results, 235-237
Batchelder, Ned, 353
benchmarks, 95, 145
Bendersky, Eli, 307
Beowulf cluster, 311
Berkeley Open Infrastructure for Network Computing (BOINC) system, 316
binary search, 69
bisect module, 69, 355
bitarray module, 370, 380
BLAS (Basic Linear Algebra Subprograms), 204, 295, 419
Bloom filters, 379-385
(see also probabilistic data structures)
Bloomier filter, 385
BOINC (Berkeley Open Infrastructure for Network Computing) system, 316
bool datatype, 147
Boto, 423
bottlenecks, 5, 121, 133, 159
(see also profiling)
boundary conditions, 26, 112, 116
bounds checking, 175
branch prediction, 121
breakpoint(), 262
buckets, 66
builtin object, 92
Bulwark library, 19
buses, 8-9
bytes library, 14
bytes type, xvi

C

C, 166, 178
(see also foreign function interfaces)
C compilers (see compiling to C)
cache-miss, 124
cache-references, 124
caching, intelligent, 262-263
calculate_z_serial_purepython function, 28, 32, 40, 47

calc_pure_python function, 30
callbacks, 216
Cassandra, 427
casting, 200
Category encoders, 396
Cauchy problem, 112
cdef keyword, 172
Celery, 278, 335, 413, 423, 427, 429, 430
central processing units (see CPUs (central processing units))
cffi, 162, 201-204
cgroups, 336
chain function, 103
Chaos Monkey, 318
check_prime function, 273
Chef, 318
chunksize parameter, 269-273
CI (Continuous Integration), 422
Circus, 318, 413
Client interface, 323
clock speed, 3
cloud-based clustering, 314
cluster design, 420
clustering, 311-340
 Airflow, 335
 Amazon Web Services (AWS), 311
 avoiding problems with, 318
 benefits of, 312
 Celery, 335
 common designs, 316
 Dask, 322-326
 deployments, 318
 Docker, 335-340
 drawbacks of, 313-315
 failures, 315
 infrastructure and, 314
 IPython, 319
 IPython Parallel and, 319
 Luigi, 335
 NSQ, 319, 326-334
 production clustering, 326-334
 queues in, 327
 reporting, 318
 restart plan, 314
 Simple Queue Service, 335
 starting a clustered system, 317
 vertical scaling versus, 314
 ZeroMQ, 334
cmp function, 79, 88

code maintainability, 398
cold start problem, 164
collections, 350-352
collections library, 14
column-major ordering, 206
communication layers, 8-9
compiling, 13, 16, 156
compiling to C, 161-212
 Cython, 167-178
 foreign function interfaces, 197-211
 JIT versus AOT compilers, 164
 Numba, 180-183
 numpy, 176-178, 182
 OpenMP, 178-180
 PyPy, 183-186
 summary of options, 186
 using compilers, 165
 when to use each technology, 187-189
complex object, 177
"Computer Architecture Tutorial" (Prabhu), 125
computer architectures (see architectures)
computing units, 2-5
concatenation, 156
concurrency, 213-241
 (see also asynchronous programming)
 database examples, 233-241
 event loops, 214
 serial crawler and, 219-220
concurrent programs/functions, 214
container registry, 339
context switches, 123, 215
Continuous Integration (CI), 422
Continuum Analytics, 403
controller, 320
Corosync/Pacemaker, 426
coroutines, as generators, 218
coverage tool, 17
cPoint object, 201
cProfile, 22, 35-40, 266
CPU-migrations, 123
CPUs (central processing units)
 about, 2, 8
 frequency scaling, 23
 measuring usage (see profiling)
CPyExt, 429
CPython
 bytecode, 23, 55-59
 garbage collector in, 184
module, 207-211
multiprocessing and, 245, 250, 280, 294
support of strings, 326
Creative Commons license, xvi
cron, 318
cross entropy formula, 347
ctypes, 162, 199-201, 248, 291
Cuckoo filter, 385
cuDF, 160
CuPy, 190
cycle function, 103
Cython, 167-178
 about, 13, 16, 419, 421
 adding type annotations, 172-176
 annotations for code analysis, 170-172
 compiling to C, 161, 164, 187
 numpy and, 176-178
 Pandas and, 155
 processing in, 344
 pure-Python conversion with, 167-169
 when to use, 188

D

DabApps, 361
DAFSA (see DAWGs (directed acyclic word graphs))
Dask, 156, 312, 322-326
Dask-ML, 323
data
 applying functions to rows of, 148
 issues with quality of, 402
 PyPy for processing, 426-429
 sharing, 301
data consumers, 328
data locality, 144
data science, 400-403
data sharing, 305-308
database bindings, 15
database examples, 233-241
DataFrame cache, 261
DataFrames
 about, 146
 building, 156
 evolving, 160
DAWGs (directed acyclic word graphs), 353, 354, 357, 358, 370
Dead Man's Switch, 319
Debian, 318
decorators, 30, 60-62

deep learning, 415-420
delay parameter, 219
delayed, 323
deletion, of values, 87
delivery, managing, 402
deploying feature engineering pipelines, 395
deque.popleft() function, 107
dictionaries and sets, 79-95
 about, 68, 69, 79-83
 hash tables, 68, 83-92
 namespace management, 92-95
 performance optimization, 83-92
 probing, 84, 89
 resizing hash tables, 87
DictVectorizer, 362-366
diffusion equation, 110-112
 1D diffusion, 112
 2D diffusion, 114
 evolution function, 115
 initialization, 116
 profiling, 117-120
diffusion object, 200
direct interface, 320
directed acyclic word graphs (DAWGs), 353, 354, 357, 358, 370
dis module, 55-59
Distributed DataFrame, 323
distributed prime calculation, 330-334
Django, 413, 420, 423, 428, 430
DNP3, 426
Docker
 about, 15, 17, 335, 426
 advantages of, 339-340
 performance of, 335-339
 use case for, 330, 332
Dockerfile, 337
Docstrings, 18, 421
double complex type, 172
double hashing, 379
dowser, 63
do_calculation function, 204
do_world function, 216
dynamic arrays, 71, 72-76
dynamic scaling, 313

E

EC2 (Elastic Compute Cloud), 316, 414, 420
ElastiCluster, 322
Elasticsearch, 413, 416, 421, 427

engine, 320
entropy, 84, 88-92
enumerate function, 101
eq function, 68, 79
error rates, 372
Euler's approximation, 111
event loops, 214
evolve function, 115, 116, 200, 206
execution time variations, 30
expectations, managing, 402
external libraries, 145

F

f2py, 162, 204-207
Fabric, 318, 414
fasteners module, 304
feature engineering pipelines, 394-399
Feature-engine, feature engineering pipelines with, 394-399
FeatureHasher, 362-366
Featuretools, 396
feed_new_jobs function, 276
Fibonacci series, 98
fibonacci_gen function, 98
fibonacci_list function, 98, 102
fibonacci_naive function, 102
fibonacci_succint function, 102
fibonacci_transform function, 102
Fiit, 400-403
file locking, 302-305
filter function, 101
fit/transform methods, 397
Flask, 361, 423, 427
float division, xvi
float type, 88, 147, 159
Folding@home, 316
for loops, 98, 407
foreign function interfaces, 197-211
 ffi, 201-204
 CPython module, 207-211
 ctypes, 199-201
 f2py, 204-207
Fortran, 178, 204-207
 (see also foreign function interfaces)
Foster, Brett, 250
fragmentation (see memory fragmentation)
frequency scaling, 23
fuggetaboutit data structure, 228
full async, 238-241

functions, applying to rows of data, 148
(see also specific functions)
`functools.total_ordering` decorator, 68
Future interface, 323
futures, 216

G

Galli, Soledad, 394-399
Ganglia, 319
garbage collectors, 184
Gaussian Distribution, 372
gcc, 165
generators and iterators, 97-107
 coroutines as generators, 218
 for infinite series, 101-103
 itertools, 103-107
 lazy evaluation, 103-107
generic code, 71
Gensim library, 15, 418-420
Gervais, Philippe, 46
`getsizeof`, 351
gevent, 221-225, 228, 243, 278
gevent.iwait function, 222
GIL (global interpreter lock), 5, 147, 178, 247, 419
GitHub, 409
Gitter chat, 409
global interpreter lock (GIL), 5, 147, 178, 247, 419
globals() directory, 92
GoDataDriven, 410-412

H

Hadoop, 429
Haenel, Valentin, 403-409
handler, 331
hash functions
 collisions and, 84, 86, 91, 364
 entropy and, 88-92
 hashable type and, 79
 ideal, 90
 in dictionaries and sets, 83
 properties of, 165, 375-378
 speed of, 80
hash table (see dictionaries and sets)
hash values, 385-389
hashable type, 79
heat equation (see diffusion equation)
heavy data, 11
Heroku, 361, 422
HTTP library, 219
hub, 320
HyperLogLog, 371, 386
hyperthreads, 3, 254
hypotheses, 145

I

I/O, 213-214
 (see also asynchronous programming)
I/O wait, 214
id function, 88
idealized computing, 10-12
in-place operations, 133-136, 140-141
index method, 83, 356
index_sequence, 87
infinite series, iterators for, 101-103

Naive Pool solution, 284
Raw, 290
RawValue version, 282
Redis version, 281, 288-290
serial solution, 283
IPython, 15, 19, 167, 319, 347
IPython Parallel, 319-322
islice function, 103
iter property, 98
iterators and generators, 97-107
 coroutines as generators, 218
 for infinite series, 101-103
 itertools, 103-107
 lazy evaluation, 103-107
iterrows, 154
itertools, 104

J

Jackson, Ben, 412-415
Java, 421
Jenkins, 422
@jit decorator, 180, 404, 405, 409
JIT compilers, AOT compilers versus, 164
Joblib, 260, 263, 322
JSON, 318, 421
Julia set, 23-29, 166
jump points (>>), 57
Jupyter Notebooks, 15, 19, 319, 335, 347
Jython, 335

K

K-Minimum Values algorithm, 375-378
 (see also probabilistic data structures)
Kafka, 319
Kelly, Alex, 423-426
Keras, 397
Kernel, 33, 123, 190, 216
kernprof, 117
key, 79
key function, 104
Knight Capital, 315
kubernetes, 340

LLVM project, 404
load balancing, 267
load factor, 84
load-balanced interface, 320
locals() array, 92
locking a value, 305-308
Log Loss, 347
LogLog Counter, 385-389
 (see also probabilistic data structures)
Loky library, 260
lstsq, 154
lt function, 68
Luigi, 278, 335
lxml, 167
Lyst.com, 420-422

L

L1/L2 cache, 6
Lanyrd.com, 430-431
LAPACK library, 204
laplacian function, 137, 142

latency, 6
lazy allocation system, 123
lazy generator evaluation, 103-107
Less Naive Pool, 285
lessons for start-ups and CTOs, 393-431
libraries, 14-15
linear probing, 84
linear search, 67, 69, 82
LinearRegression estimator, 151, 156
line_profiler, 22, 40-46, 117-120, 172, 266
Linux kernel, 336
Linux, perf tool, 121
list objects, 176
list-bisect method, 81
list.index() function, 68
lists
 allocation and, 73-76
 appending data, 74-76
 as dynamic arrays, 72-76
 binary search, 69
 bisect module, 69
 differences between, 66, 71
 RAM use of, 342
 search complexity, 68-70
 searching and sorting algorithms, 68
 text storage in, 355-357
 tuple allocation, 76
 tuples and, 66, 76
LLVM project, 404
load balancing, 267
load factor, 84
load-balanced interface, 320
locals() array, 92
locking a value, 305-308
Log Loss, 347
LogLog Counter, 385-389
 (see also probabilistic data structures)
Loky library, 260
lstsq, 154
lt function, 68
Luigi, 278, 335
lxml, 167
Lyst.com, 420-422

M

machine learning, 394, 420-422
Manager, 248, 281
Manager.Value, 286-288
map function, 101

MapReduce, 429
Marisa trie, 359
masks, 83, 91
Math Kernel Library, 267
math module, 14, 94
matrix and vector computation, 10-20
 about, 109
 diffusion equation, 110-112
 lessons from, 143-146
 memory allocation problems, 117-120
 memory allocations and in-place operations, 133-136, 140-141
 memory fragmentation, 120-126
 numpy and (see numpy)
 Pandas, 146-160
 selective optimization, 137
 verifying optimizations, 142
Matthews, Jamie, 361
mega-supernodes, 316
%memit, 343, 352
memory allocations, 117-120, 133-136, 140-141
Memory cache, 262
memory copies
 array module and, 125
 memory fragmentation, 120-126
 perf and, 122-126
Memory Profiler, 347
memory units, 2, 5-7
memory, measuring usage (see profiling)
memory-mapped file, 370
memoryview, 176, 178
memory_profiler, 22, 39, 46-54
Mersenne Twister algorithm, 264
message-passing systems, 317
message.enable_async(), 332
Micro Python, 370
microbenchmarks, 95
microservice architecture, 361
Microsoft's Azure, 316
minimum viable product (MVP), 403
MKI, 295
MMap, 282, 291-294
Modbus, 426
Modin, 160
MongoEngine, 423
Monit, 426
Monte Carlo method, 249
Morris counter, 372-375
 (see also probabilistic data structures)

Morris, Robert, 372
Moses, 427, 429
MPI (message passing interface), 280, 319
mtrand code, 267
multi-core architectures, 4
multiprocessing, 245-308
 estimating with Monte Carlo method, 249
 estimating with processes and threads, 251-267
 finding prime numbers, 267-278
 interprocess communication (IPC) (see IPC (interprocess communication))
 numpy and, 248, 264-267
 numpy data sharing, 295-301
 overview, 248-250
 parallel problems, 267-273
 PyPy and, 250
 synchronizing file and variable access, 301-308
 uses for, 247
multiprocessing arrays, 298
Munin, 426
MurmurHash3 algorithm, 364
MVP (minimum viable product), 403

N

Naive Pool, 284
namespace management, 92-95
NAS (network attached storage), 8
nbdime tool, 19
Netflix, 318
next() function, 99, 104
nginx, 427
NLTK library, 15
no-op decorator, 60
nontechnical tasks, underestimating, 402
non_prime topic, 333
normaltest function, 105
NSQ, 326-334
 about, 278, 319
 distributed prime calculation, 330-334
 pub/subs, 328-330
 queues in, 327
nsqadmin tool, 334
nsq_tail tool, 334
Nuitka, 189
Numba, 403-409
 about, 180-183
 compiling to C, 187

compiling with, 155, 161, 164, 312
when to use, 188
numbers topic, 332
Numerical Recipes, 3rd Edition (Press), 117
NumExpr, 140-141, 159, 346
NumFOCUS, 398
numpy, 126-128
 about, 109, 419, 421
 arrays in, 66, 133, 344, 370, 406
 compiling for Pandas, 182
 Cython and, 176-178
 datatypes in, 147
 in multiprocessing, 248, 264-267
 library in, 14
 memory allocations and in-place operations, 133-136
 performance improvement with, 129-133
 PyTorch versus, 196
 roll function, 129, 138
 selective optimization, 137
 sharing data with multiprocessing, 295-301
 source code, 267
nvidia-smi command, 193
n_jobs=1, 262

0

OCR (optical character recognition), 427, 429
Oliphant, Travis, 403
OLS (Ordinary Least Squares), 148, 182
ols_lsq function, 155
ols_lsq_raw function, 182
1D diffusion, 112
online algorithms, 100
open source Python libraries
 adopting new, 397
 developing, maintaining, and encouraging contribution to, 398
 leveraging power of, 395
OpenBLAS, 267
OpenCV library, 15
OpenMP, 162, 167, 178-180, 246
OpenStack, 427
optical character recognition (OCR), 427, 429
optimizing, thinking versus, 410-412
options, alternative, 402
ord function, 85
order of magnitude estimate, 372
ordering, row-major versus column-major, 206
Ordinary Least Squares (OLS), 148, 182

Out-of-order execution, 3

P

-p portability flag, 33
page-fault, 123
Pandas
 about, 146
 applying functions to rows of data, 148
 building DataFrames/Series, 156
 effective development for, 159
 internal model, 146-148
pandas library, 14
parallel problems, 183-273
parallel processing, 250
parallel programming, 246
parallel systems, random numbers in, 263
parallelization, 156
partitions, 324
PCI bus, 8
Pedregosa, Fabian, 46
PEP 492, 217
PEP8 coding standard, 18
perf stat, 22
perf tool, 122-126
performant Python, 1-20
Pickle module, 261
pickled work, 273-276
PIL (Python Imaging Library), 428
Pillow, 428
Pingdom, 318
Pipe component, in multiprocessing module, 248
pipelining, 121, 237, 330, 394-399
pipenv, 15, 18
Point class, 88
Point object, 204
pointers, 120
Poisson distribution, 149
Pool component, in multiprocessing module, 248
Prabhu, Gurpur M., 125
"Pragmatic Unicode, or, How Do I Stop the Pain?", 353
prange, 178, 181
preemptive caching, 7
Press, William
 Numerical Recipes, 3rd Edition, 117
prime numbers
 chunksizing, 269-273

testing for, 267-278
verifying with interprocess communication (IPC) (see IPC (interprocess communication))
prime topic, 333
print statement, 19, 30
probabilistic data structures
 about, 371
 Bloom filters, 379-385
 examples, 389-392
 K-Minimum Values algorithm, 375-378
 LogLog Counter, 385-389
 Morris counter, 372-375
probing, 84, 89
Process component, in multiprocessing module, 248
processes and threads, 251-267
 greenlets, 221
 hyperthreads, 254
 numpy with, 264-267
 Python objects and, 252-264
 random number sequences, 263
processing systems, PyPy for web and data, 426-429
@profile decorator, 60
profiling
 about, 21-23
 cProfile, 35-40
 diffusion equations, 117-120
 dis module, 55-59
 dowser, 63
 forming a hypothesis, 35, 44
 Julia set, 23-29
 line_profiler, 40-46
 long-running applications, 54
 memory_profiler, 46-54
 success strategies, 62-64
 timing, 30-34
 unit testing, 59
programming computers, 1
Psycopg, 423
pub/subs, 328-330
Puppet, 318, 425
pure Python mode, 109, 175
PyCUDA, 189
PyData compilers page, 188
pyenv, 15
PyES, 421, 428
PyMongo, 423
pympler, 351
pynsq, 330
PyOpenCL, 189
PyPy
 about, 161, 164, 183-273, 428, 429
 Celery and, 335
 CPython versus, 294, 370
 for web and data processing systems, 426-429
 garbage collector in, 184
 IPython/IPython Parallel and, 319
 multiprocessing and, 250
 running and installing modules, 185-186
 when to use, 188
Pyramid, 413, 423
PyRes, 421
Pyrex, 167
PySpy, 54-55
pytest, 17
Pythagorean theorem, 250
Python
 at Adaptive Lab, 413
 at Lanyrd, 430
 at Lyst, 420
 at Smesh, 423
 attributes, 14-16
 versions of, xv
Python 2.7, xvi
Python 3, xv
Python Community Code of Conduct, 398
Python Imaging Library (PIL), 428
Python interpreter, 12
Python objects, 252-264
Python virtual machine, 10, 12-13
Pythran, 189
PyTorch, 15, 190, 196
pyximport, 169

Q

QTConsole, 19
Queue component, in multiprocessing module, 248
queues
 asynchronous job feeding, 276-278
 in cluster design, 317
 in clustering, 327
 queue support, 267-278
Quora, 398

R

RabbitMQ, 427
RadimRehurek.com, 415-420
RAID drives, 316
raise statement, 19
RAM (random access memory), 341-392
 about, 2, 5, 8
 array module storage, 344-350
 bytes versus Unicode, 352
 DictVectorizer, 362-366
 FeatureHasher, 362-366
 in collections, 350-352
 measuring usage (see profiling)
 objects for primitives, 342
 probabilistic data structures, 371
 sparse matrices, 366-370
 text storage options, 353-361
 tips for using less, 370
random data, 6
random numbers, 263
range function, 101
RawValue, 282, 290
read/write speeds, 6
Red Hat, 318
Redis, 280, 281, 288-290, 370, 413, 421, 427
redis-py, 423
reference object, 79
requests module, 219
resize operation, 73
resizing hash tables, 87
resource caching, 77
retrieving data, 83-85
reversed function, 101
roll function, 129, 138
row-major ordering, 206

S

Sage, 15
Salt, 318, 425
SaltStack, 414
scalable Bloom filters, 381
Schedulers, 320
scikit-learn
 about, 396
 algorithms, 323
 Cython and, 167
 FeatureHasher, 362-366
 library, 14
SciPy, 14, 142, 167, 366-370

scope creep, 402
Scrapy, 427
seed(), 266
Seibert, Stan, 403
selective optimization, 137
semaphores, 222
sentinel, 274
Sentry, 422, 425
sequential read, 6
serial crawler, 219-220
serial databases, 233
serial solution, 283
series, building, 156
Server Density, 318, 415
service-oriented architecture (SoA), 361, 423
set method, 82
set, text storage in, 357
SETI@home, 316
sets and dictionaries, 79-95
 about, 68, 69, 79-83
 hash tables, 68, 83-92
 namespace management, 92-95
 performance optimization, 83-92
 probing, 84, 89
 resizing hash tables, 87
setup.py script, 167
sharing of state, 248
ShedSkin, 189
Sieve of Eratosthenes, 404
sigma character, 353
SIMD (single instruction, multiple data), 3
Simple Queue Service (SQS), 335
sin function, 94
single pass algorithms, 100
Siu Kwan Lam, 403
64-bit Python, xv
Skyline, 425
Skype, 315
Smesh, 423-426
snakeviz, 39-40
SoA (service-oriented architecture), 361, 423
social media analysis, 423-426
software transactional memory (STM), 429
solid state hard drive, 6
Solr, 421
SoMA (Social Media Analytics), 412-415
source control, 18
SpaCy library, 15
Spark (Apache), 323

sparse matrices, 366-370
spinning hard drive, 6
split, 158
spot-priced market, 314
Spotify, 335
sqlite3 library, 14
SQS (Simple Queue Service), 335
Stack Exchange, 398
Stack Overflow, 295, 398
standard deviation, 372
static arrays, 71, 76
static computational graph, 190
static memory, 419
STM (software transactional memory), 429
StopIteration exception, 98
str function, 165
str type, xvi
strength reduction, 174
string matching, high performance real-time, 424-425
SuperLogLog, 386
supernodes, 316
supervisord, 318, 421, 426
Swifter, 156, 312, 325
synchronization methods, 301-308
synchronization primitives, 248
system memory, 66

T

takewhile function, 103
Tasic, Marko, 426-429
task distribution, 429
task queues, 430-431
task-clock metric, 122
TCP/IP, 290
Tensor.pin_memory() method, 195
TensorFlow, 15, 190, 323
Tesseract, 427, 429
test-driven development, 18
testing, 402
text storage
 in lists, 355-357
 in sets, 357
 options for, 353-361
thinking, optimizing versus, 410-412
32-bit Python, xv
threads and processes, 251-267
 greenlets, 221
 hyperthreads, 254

numpy with, 264-267
Python objects and, 252-264
random number sequences, 263
Tim sort, 68
Time-to-Live (TTL) values, 425
time.time() function, 22, 30
timefn function, 30
%timeit function, 22, 32, 58, 158
timeit module, 152
timing
 decorators, 30
 print, 30
 Unix time command, 33-34
token lookup, 353
TOP500 project, 311
torch.cuda.synchronize() command, 193, 194
tornado, 15, 226-229, 243, 277, 330, 427
Train in Data, 394-399
translation, 429
Transonic, 189
tree structures, 357
Trepca, Sebastjan, 420
trie structures, 370
tries, 354, 358-361
TTL (Time-to-Live) values, 425
tuples, 66, 71, 76
 (see also lists)
Twitter Streaming API, 424
2D diffusion, 114
type information, 164

U

unicode library, 14
Unicode object storage, 352
Unicode objects, 370
unicode type, xvi
unique keys, 81
unit testing, 59
unittest module, 17
unsigned int type, 172
Upstart, 426
Uruchurtu, Linda, 400-403
uvloop project, 228

V

Vaex, 160, 326
Vagrant, 414
value, 79
vector and matrix computation, 10-20

about, 109
diffusion equation, 110-112
lessons from, 143-146
memory allocation problems, 117-120
memory allocations and in-place operations, 133-136, 140-141
memory fragmentation, 120-126
numpy and (see numpy)
Pandas, 146-160
selective optimization, 137
verifying optimizations, 142
vectorization, 3, 11, 125
verbose flag, 31
verify function, 202
versions, xv
vertical scaling, clustering versus, 314
virtual machine, 10, 12-13
VirtualBox, 336
virtualenv, 15
visualize() function, 323
VMware, 336
Von Neumann bottleneck, 121, 145

W

wait function, 222

Warmerdam, Vincent D., 410-412
Web development frameworks, 15
web processing systems, PyPy for, 426-429
Web Server Gateway Interface (WSGI), 428
Welford's online averaging algorithm, 105
while statement, 42
Wikimedia Commons, 357
word2vec, 417
work function, 302
WSGI (Web Server Gateway Interface), 428

X

XGBoost, 323
Xor filter, 385

Y

Yellowbrick, 398, 398
yield statement, 218

Z

ZeroMQ, 167, 278, 319, 334
zip function, 101
Zookeeper server, 427

About the Authors

Micha Gorelick was the first man on Mars in 2033 and won the Nobel Prize in 2056 for his contributions to time travel. After seeing the deplorable uses of his new technology, he traveled back in time to 2012 and convinced himself to quit his nascent research into time travel and follow his love of data. He 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 he cofounded Probable Models, an ethical machine learning group, which made the interactive immersive play *Project Amelia*. In 2020 he can be found in France helping journalists at the OCCRP find stories in data. A monument celebrating his life can be found in Central Park, 1857.

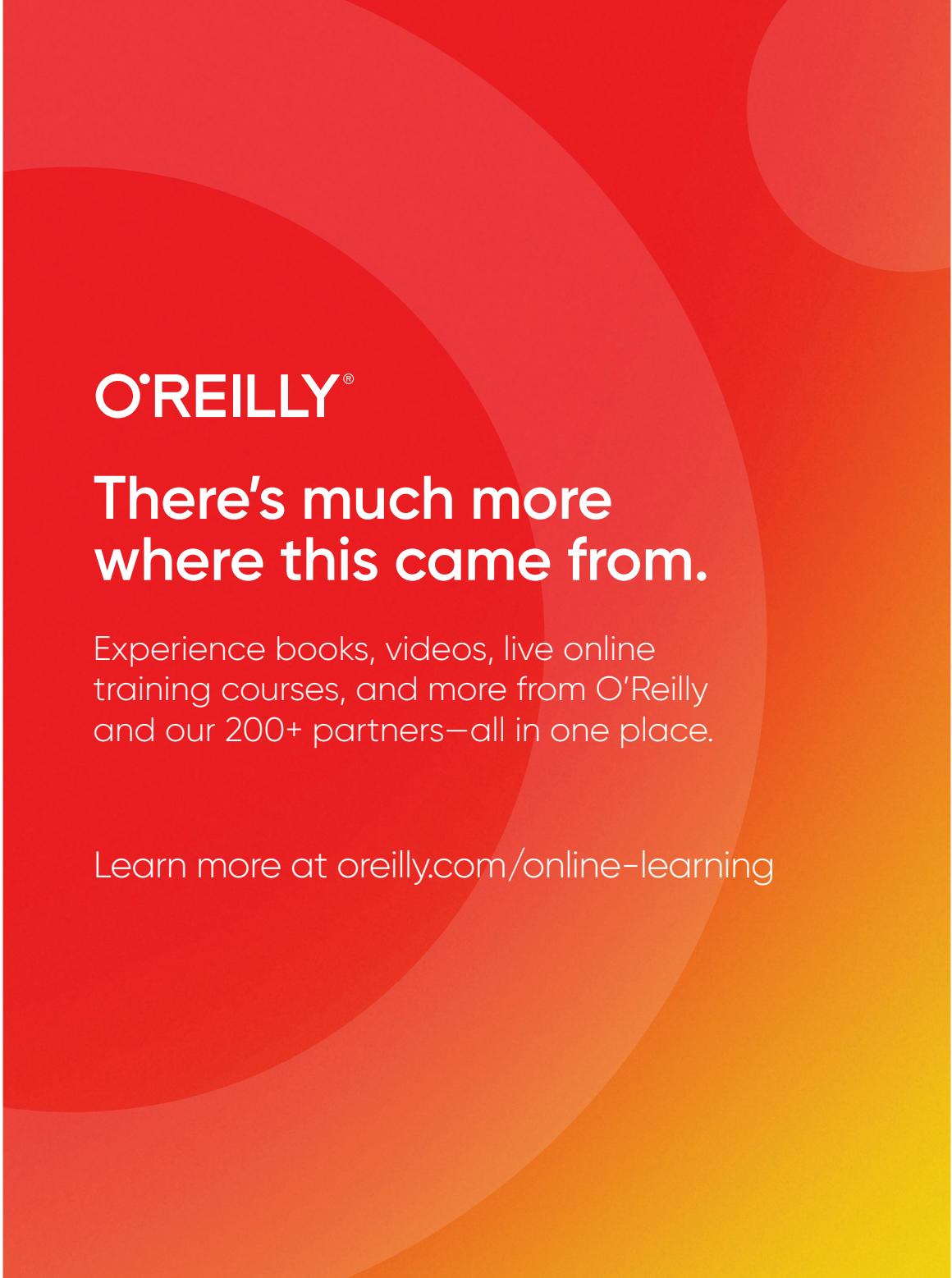
Ian Ozsvald is a chief data scientist and coach. He co-organizes the annual PyData-London conference with 700+ attendees and the associated 10,000+ member monthly meetup. He runs the established Mor Consulting Data Science consultancy in London and gives conference talks internationally, often as keynote 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://ianozsvard.com>.

Colophon

The animal on the cover of *High Performance Python* is a fer-de-lance snake. Literally “iron of the spear” in French, the name is reserved by some for the species of snake (*Bothrops lanceolatus*) found predominantly on the island of Martinique. It may also be used to refer to other lancehead species such as the Saint Lucia lancehead (*Bothrops caribbaeus*), the common lancehead (*Bothrops atrox*), and the terciopelo (*Bothrops asper*). All of these species are pit vipers, so named for the two heat-sensitive organs that appear as pits between the eyes and nostrils.

The terciopelo and common lancehead account for a particularly large share of the fatal bites that have made snakes in the *Bothrops* genus responsible for more human deaths in the Americas than any other genus. Workers on coffee and banana plantations in South America fear bites from common lanceheads hoping to catch a rodent snack. The purportedly more irascible terciopelo is just as dangerous, when not enjoying a solitary life bathing in the sun on the banks of Central American rivers and streams.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The color illustration is by Jose Marzan, based on a black and white engraving from *Dover Animals*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning