PROFILE · CALL TREE

build_daily_report 100%

shipping_costs

nearest_depot

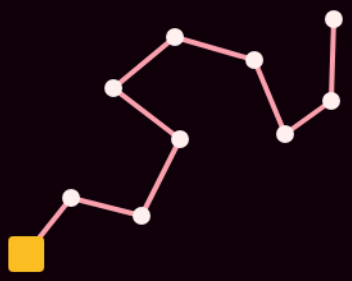
haversine_km

haversine_km

pings

hot path found → vectorize it

ROUTE



```
@njit(parallel=True)
def nearest_depot(lat, lon, dl, dn):
  out = np.empty(lat.shape[0])
  for j in prange(lat.shape[0]):
    ...
  return out
```

SAME ANSWER, LESS TIME

baseline

set / dict

NumPy

Numba

checksums match ✓

```
$ python -m timeit -s "s = set(ids)" "1019999 in s"
```

MEASURE · PROFILE · OPTIMIZE

High-Performance Python

Profile, optimize, and scale Python code with NumPy, Numba, and smart algorithms

timeit · cProfile · pyinstrument · tracemalloc · Big-O · caching
NumPy · pandas · Polars · Numba · multiprocessing · asyncio

MEASURE FIRST · PROVE EVERY SPEEDUP

THENEXBOOKS

High-Performance Python

Profile, Optimize, and Scale Python Code with NumPy, Numba, and Smart Algorithms
Measure first, fix the real bottleneck, and prove every speedup — step by step

First Edition

THENEXBOOKS

High-Performance Python
Profile, Optimize, and Scale Python Code with NumPy, Numba, and Smart Algorithms

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python and its libraries change over time; features, defaults, and performance characteristics can differ between versions. All timings in this book were measured on our test machine and are shown to illustrate ratios and trends — your numbers will differ. Always measure on your own hardware and data.

Python is a registered trademark of the Python Software Foundation. NumPy, Numba, pandas, Polars, DuckDB, Apache Arrow, Cython, PyPy, Rust, Windows, and other product names mentioned are trademarks or registered trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Python Software Foundation or any of the projects or companies mentioned.

Stonebridge Logistics, the delivery company used as the running example in this book, is fictional, as are its depots, vans, parcels, routes, GPS traces, and all sample data. The service area and coordinates are invented for teaching purposes.

Preface

Sooner or later, every Python developer meets a program that is simply too slow. A nightly report that used to take a minute now takes an hour. An API endpoint times out when the data grows. A data-cleaning script that was fine on a sample freezes on the real file. The usual reaction is to guess: rewrite a loop, add a cache somewhere, or declare that "Python is slow" and start planning a rewrite in another language.

Guessing is expensive. Most of a program's time is usually spent in a small part of its code, and that part is rarely where we expect it to be. The developers who make Python fast do something different: they measure first, find the real bottleneck, apply the cheapest fix that works, check that the answers are still correct, and measure again. With that discipline, and with the right tools — better algorithms and data structures, caching, NumPy vectorization, Numba's just-in-time compiler, multiple processes, efficient file formats — Python code can often become many times faster without leaving Python. In this book's own examples, measured speedups on our test machine range from about 5 times to several hundred times, depending on the workload and the technique.

This book teaches that discipline and those tools. We do not collect random speed tricks. We take one realistic, slow codebase and make it fast step by step, explaining *why* each change helps, proving that the results stay identical, and showing the measured effect every time.

Who This Book Is For

- **Intermediate Python developers** who are comfortable with functions, classes, comprehensions, and packages, and now have code that is too slow.
- **Data engineers and analysts** whose scripts and pipelines have outgrown their data, and who want to know when to reach for NumPy, Polars, DuckDB, or Numba.
- **Backend developers** who need to speed up batch jobs, reports, and CPU-heavy endpoints, and want to understand processes, threads, and asyncio properly.
- **Scientists and engineers** who write numerical code in Python and want compiled-language speed without giving up Python's productivity.

We assume you already know Python. If you are new to the language, start with a beginner book first; this one begins where everyday Python ends.

What You Will Learn

- How to time code correctly with `perf_counter` and `timeit`, and how to avoid the classic benchmarking mistakes.
- How to profile CPU time with cProfile, pstats, and pyinstrument, and memory with tracemalloc — and how to read the results.

- How Big-O thinking and the right data structure (set, dict, deque, heapq, bisect) produce the largest wins of all.
- Which built-ins and idioms are genuinely faster, and which "tricks" are folklore — with measurements.
- How to cache and memoize safely, and how generators save both time and memory.
- How to vectorize with NumPy: arrays and dtypes, broadcasting, fancy indexing, reductions, views, and in-place operations.
- How to speed up pandas and Polars code, and when DuckDB is the simplest answer.
- How to compile hot loops with Numba, including parallel loops with `prange`.
- How to use multiprocessing for CPU-bound work and asyncio or threads for I/O-bound work — and why the GIL matters.
- What C extensions, Cython, mypyc, PyO3, PyPy, and the newer experimental CPython builds offer, and how to evaluate them.
- How to cut memory use and choose fast file formats, and how to process data larger than memory.
- How to benchmark like a scientist, protect performance with regression tests, and choose the right technique for each bottleneck.

How This Book Is Organized

Part I — Measure Before You Optimize (Chapters 1–5) introduces the performance mindset, the Stonebridge reference code, timing, CPU profiling, and memory profiling. **Part II — Faster Pure Python** (Chapters 6–10) covers algorithms, data structures, fast idioms, caching, and generators. **Part III — Numerical Speed with NumPy and DataFrames** (Chapters 11–15) covers NumPy from a performance point of view — arrays, vectorization, broadcasting, views and temporaries — plus pandas and Polars performance. **Part IV — Compiling and Parallel Execution** (Chapters 16–20) covers Numba, multiprocessing, threads and asyncio, and compiled extensions. **Part V — Memory, I/O, and the Runtime** (Chapters 21–24) covers memory optimization, file formats, out-of-core processing, and the interpreter itself. **Part VI — Performance Engineering in Practice** (Chapters 25–28) covers benchmarking discipline, regression testing, a decision playbook, and a complete end-to-end case study. The appendices provide a cheat sheet, templates and reference tables, an optimization checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Stonebridge Logistics

Throughout the book we work on the code of **Stonebridge Logistics**, a fictional parcel delivery company with twelve depots, 200,000 parcels a day, and a fleet of fifty vans that report their GPS position every four seconds. Stonebridge's developers wrote clear, correct, pure-Python code: it finds

the nearest depot for every parcel, prices each delivery, measures how far every van drove, checks scanned parcels against the manifest, and plans delivery routes. Then the business grew, and the daily operations report became painfully slow.

All of Stonebridge's data is generated from fixed random seeds, so you get exactly the same parcels, pings, and routes that we use, and every result in the book comes with a checksum you can reproduce. We keep the original "baseline" code untouched and build faster versions next to it, so every speedup is compared against the same reference and checked for identical answers.

Versions and Conventions

Examples target **Python 3.12 or later** on CPython, with NumPy 2.x, recent versions of Numba, pandas, Polars, DuckDB, and pyinstrument. Every example was run, and the output shown is real output. Where a tool is optional, experimental, or needs a compiler we did not use, a Note says so and we do not show invented results.

About timings. Performance depends on your CPU, memory, operating system, library versions, and what else your computer is doing. Every time in this book is labeled as measured *on our test machine* (a 16-core desktop running Windows and Python 3.12). Treat the absolute numbers as illustrations and focus on the **ratios** — "about 20 times faster" — and on the method. Run the examples yourself; your numbers will differ, and measuring them is the point.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can produce wrong results, wasted effort, or misleading measurements.

Code appears in `monospace` with the file name above each listing, such as `stonebridge/fast.py`; program output is shown separately under the caption **Output**. Long lines are wrapped to fit the page, and long profiler reports are trimmed with `...`. Commands are written as `python -m ...`; use the command that runs Python 3.12+ on your system.

Contents

Preface

PART I — MEASURE BEFORE YOU OPTIMIZE

1 The Performance Mindset: Why Python Is Slow and When It Matters

2 The Stonebridge Workbench: Datasets, Baselines, and Correctness Checks

3 Timing Code Correctly: perf_counter, timeit, and Benchmark Hygiene

4 Profiling CPU Time: cProfile, pstats, and pyinstrument

5 Profiling Lines and Memory: Hot Lines, tracemalloc, and Allocation Hunting

PART II — FASTER PURE PYTHON

6 Algorithms First: Big-O for Working Developers

7 Choosing the Right Data Structure: list, dict, set, deque, heapq, and bisect

8 Fast Built-ins and Idioms: Loops, Comprehensions, Strings, and Lookups

9 Caching and Memoization: lru_cache, cache, and Precomputation

10 Generators and Lazy Pipelines: Saving Time and Memory

PART III — NUMERICAL SPEED WITH NUMPY AND DATAFRAMES

11 NumPy for Performance: Arrays, dtypes, and Memory Layout

12 Vectorization: Replacing Python Loops with Array Operations

13 Broadcasting, Fancy Indexing, and Fast Reductions

14 Advanced NumPy: Views, Copies, In-Place Operations, and Temporaries

15 DataFrames at Speed: pandas and Polars Performance Basics

PART IV — COMPILING AND PARALLEL EXECUTION

16 Numba Fundamentals: JIT-Compiling Hot Loops

17 Numba in Depth: Types, parallel=True, prange, and Pitfalls

18 Multiprocessing for CPU-Bound Work: Pools, Chunks, and Shared Memory

19 Threads, the GIL, and asyncio for I/O-Bound Work

20 Compiled Extensions: C, Cython, mypyc, and Rust with PyO3

PART V — MEMORY, I/O, AND THE RUNTIME

21 Memory Optimization: Object Overhead, __slots__, array, and Compact Types

22 Fast File I/O: CSV, Binary Formats, Parquet, and Memory Mapping

23 Out-of-Core Data: Chunking, Streaming, and Lazy Queries

24 The Runtime Itself: Startup, Imports, Garbage Collection, PyPy, and New CPython Builds

PART VI — PERFORMANCE ENGINEERING IN PRACTICE

25 Benchmarking Discipline: Noise, Statistics, and Fair Comparisons

26 Performance Regression Testing: Budgets, pytest, and CI

27 The Optimization Playbook: Matching Techniques to Bottlenecks

28 Case Study: Optimizing the Stonebridge Daily Report End to End

APPENDICES

A Performance Cheat Sheet

B Templates and Reference

C Optimization Checklist

D Glossary

PART I

Measure Before You Optimize

Why Python code is slow and when it matters, the Stonebridge reference code and its correctness checks, timing code without fooling yourself, and profiling CPU time and memory to find the real bottleneck before changing a single line.

1 · The Performance Mindset
2 · The Stonebridge Workbench
3 · Timing Code Correctly
4 · Profiling CPU Time
5 · Profiling Lines and Memory

CHAPTER 1 · PART I — MEASURE BEFORE YOU OPTIMIZE

The Performance Mindset: Why Python Is Slow and When It Matters

Before we change a single line of code, we need a clear picture of what "slow" means, why pure Python loops cost what they cost, and how to decide whether a speedup is worth the effort. This chapter builds that mindset and introduces Stonebridge Logistics, the company whose slow code we will make fast throughout the book.

A Report That Used to Be Fast

Every performance story starts the same way. A piece of code that was perfectly fine last year is now the thing everyone complains about. Nobody wrote it badly on purpose. The data grew, a feature was bolted on, a second loop appeared next to the first, and one morning the operations team noticed that the daily report takes long enough to go and make coffee.

That is exactly where we meet **Stonebridge Logistics**, a fictional parcel delivery company that will be our companion for the whole book. Stonebridge runs 12 depots spread over a regional service area, handles about 200,000 parcels a day, and operates a fleet of 50 vans that each send a GPS ping every 4 seconds. Every evening a Python program reads the day's parcels and pings and produces the *daily operations report*: how many parcels each depot is responsible for, the day's shipping revenue, the busiest depot, the total distance driven by the fleet, and which van drove the furthest.

The code is clear, correct, and written in plain Python with lists of records and simple loops. It is also slow. In this book we take that code apart, find out where the time goes, and make it fast without ever breaking the numbers the business depends on. Along the way you will learn a toolbox that applies to any slow Python program: timing, profiling, better algorithms and data structures, caching, NumPy, DataFrames, Numba, multiple processes, asynchronous I/O, compact memory layouts, and efficient file formats.

But the toolbox is not the most important thing. The most important thing is a habit of mind: **measure first, find the real bottleneck, apply the cheapest fix that works, verify that the answer is still correct, and measure again**. This chapter is about that habit.

What "Slow" Actually Means

"The program is slow" is a complaint, not a measurement. Before we can fix anything we need to know which kind of slow we are dealing with, because different kinds of slowness call for different fixes.

Latency and throughput

Latency is how long a single operation takes from start to finish: the time between clicking "Generate report" and seeing the result, or between a web request arriving and the response leaving. **Throughput** is how much work gets done per unit of time: parcels priced per second, pings processed per minute, requests served per hour. The two are related but not the same: a nightly batch job can have excellent throughput and terrible latency.

The distinction matters because the fixes differ. Running work on several CPU cores at once (Chapter 18) can increase throughput dramatically while doing nothing for the latency of an individual item. Caching (Chapter 9) can cut latency for repeated requests to almost nothing while doing nothing for the first request. Vectorizing a loop (Chapter 12) usually improves both, because it simply does the same work with fewer instructions.

User-visible versus batch

The second question is *who waits*. When a person is waiting, small delays matter a great deal: a dashboard that takes four seconds to refresh feels broken, while one that takes a fifth of a second feels instant. When a machine is waiting, as in a nightly batch job, nobody cares whether the job takes four seconds or forty, as long as it finishes before the next step needs its output. In batch work the cost that matters is often money (cloud compute billed by the minute) or the time window available before the business day starts.

The Stonebridge daily report sits in between. It was designed as a batch job, but the operations team has started to rerun it during the day after correcting data, and every rerun means people sitting in front of a screen. That change of usage is what turned an acceptable runtime into a problem, and it is a very common pattern: code becomes "slow" when the way people use it changes, not only when the code changes.

Setting a target

Once we know what kind of slow we are dealing with, we can set a **performance target**: a concrete number that tells us when we are done. "Faster" is not a target. "The daily report must finish in under one second for 200,000 parcels and 1,000,000 pings on the operations laptop" is a target. A target gives you permission to stop. Without one, optimization work tends to continue long after it has stopped paying for itself, and the code gets harder to read for no business benefit.

Why Pure Python Is Slow in Tight Loops

Python is not slow at everything. Reading a file, sorting a list with the built-in `sorted`, looking up a key in a dictionary, or multiplying two large NumPy arrays all run at speeds close to what a compiled language achieves, because the heavy lifting happens in carefully optimized C code. Where pure

Python is genuinely slow is in **tight loops of small operations written in Python itself**: millions of iterations, each doing a few arithmetic steps on individual numbers. Unfortunately, that is precisely what most slow business code looks like.

The standard Python interpreter, CPython, pays several costs on every one of those small steps. Understanding them will help you predict which optimizations can work and which cannot.

Cost 1: interpreting bytecode

CPython compiles your source code into **bytecode**, a compact list of instructions for a virtual machine, and then executes those instructions one at a time in a big loop written in C. You can see the bytecode with the standard `dis` module. Here is a single line of the kind you find inside a revenue loop.

boxes.py

```
import dis
import sys

print("float object:", sys.getsizeof(4.64), "bytes")
print("int object  :", sys.getsizeof(1_000_001), "bytes")
dis.dis("total = total + p.weight_kg")
```

Output

```
float object: 24 bytes
int object  : 28 bytes
  0           0 RESUME                   0

  1           2 LOAD_NAME                0 (total)
              4 LOAD_NAME                1 (p)
              6 LOAD_ATTR                4 (weight_kg)
             26 BINARY_OP                0 (+)
             30 STORE_NAME               0 (total)
             32 RETURN_CONST             0 (None)
```

One line of Python became several virtual-machine instructions: load a name, load another name, look up an attribute, perform a binary operation, store the result. Each of those instructions is dispatched by the interpreter loop, which costs a handful of machine instructions of bookkeeping before any useful work is done. A C compiler would turn the same addition into one or two machine instructions operating directly on CPU registers.

Cost 2: dynamic typing

The `BINARY_OP (+)` instruction does not know in advance what it is adding. `total` could be an integer, a float, a string, a list, a NumPy array, or an instance of your own class that defines `__add__`. So at run time the interpreter inspects the types of both operands and finds the right implementation. Recent versions of CPython are much better at this than older ones: since Python 3.11 the *specializing adaptive interpreter* (from the "Faster CPython" work, PEP 659) notices that a particular instruction

keeps seeing floats and swaps in a faster, specialized version. That helps a lot, but the interpreter must still check that its guess remains correct every time, and it still operates on Python objects rather than raw numbers.

Cost 3: boxed objects

That brings us to the third cost. In Python, **every value is an object**. The number `4.64` is not 8 bytes of floating-point data sitting in a CPU register; it is a heap-allocated object with a reference count, a pointer to its type, and then the 8 bytes of the value. The output above shows a float object occupying 24 bytes and a small integer 28 bytes. Python integers are arbitrary-precision objects, which is wonderful for correctness (they never overflow) and expensive for speed.

Because values are boxed, arithmetic often creates new objects. Adding two floats allocates a new float object for the result (CPython reuses freed float objects to soften this), and when the old value is no longer referenced its reference count drops to zero and it is freed. A list of 200,000 weights is not a compact block of 1.6 MB of numbers; it is an array of 200,000 pointers to 200,000 separate float objects scattered around memory. Walking such a list means following pointers, which is far less friendly to the CPU cache than reading a contiguous block. Chapters 11 and 21 measure exactly how much memory this costs for Stonebridge's data.

Cost 4: function calls

Calling a Python function is one of the more expensive things the interpreter does: it creates a frame, binds the arguments, and sets up the call before the body even starts. Recent versions have made calls considerably cheaper, but a small function called millions of times still pays that price millions of times. As we will see in Chapter 4, the Stonebridge report calls a small distance function, `haversine_km`, about 5.8 million times for one day of data, 4.8 million of them for the parcels alone. Even at a microsecond or so per call, those calls add up to seconds.

The GIL, conceptually

Finally, standard CPython has a **Global Interpreter Lock** (GIL): only one thread at a time can execute Python bytecode in a given process. The GIL keeps the interpreter's internal data structures simple and safe, but it means that starting four threads to run a pure-Python loop does not make it four times faster. Threads are still very useful for waiting on I/O, because the GIL is released while a thread waits on the network or the disk, and many C extensions such as NumPy release it during long computations. Chapter 18 shows how to use several processes for CPU-bound work, Chapter 19 covers threads and `asyncio` for I/O-bound work, and Chapter 24 previews the **experimental** free-threaded builds that recent CPython versions offer.

> **NOTE**
>
> None of these costs make Python a bad language. They are the price of flexibility: dynamic typing, arbitrary-precision integers, and a simple and safe interpreter. Most of the speedups in this book work by moving the hot loop out of the interpreter (into C code inside a library, or into machine code generated by a compiler) while keeping the rest of the program in ordinary, readable Python.

When Python Is Fast Enough

It is easy to read the previous section and conclude that every Python program needs rescuing. It does not. In practice, a large share of Python code is fast enough as written, for a few recurring reasons.

- **The program is waiting, not computing.** A script that downloads files, queries a database, or calls a web API spends most of its time waiting for the other side. Making its Python code ten times faster changes nothing the user can notice. The fix, if one is needed, is to wait for several things at once (Chapter 19), not to make the loops faster.
- **The heavy work already runs in C.** Code that calls NumPy, pandas, Polars, DuckDB, regular expressions, or built-ins like `sorted` and `sum` on large inputs is already running mostly compiled code. Python acts as the conductor, not the orchestra.
- **The data is small, or the code runs rarely.** A millisecond loop over 500 items, or a one-off script that runs once, is not worth a day of optimization work.

The useful question is never "is Python slow?" but "is *this* code too slow for *this* purpose, and where exactly does its time go?" Answering that requires measurement.

Meet the Stonebridge Daily Report

Let us measure the program we came to fix. The Stonebridge reference package, `stonebridge`, ships with the book; Chapter 2 walks through it file by file. For now we only need two of its pieces: the data generators in `stonebridge/data.py`, which create the day's 200,000 parcels and 1,000,000 GPS pings from fixed random seeds, and `build_daily_report` in `stonebridge/report.py`, the slow pure-Python report.

Here is the only timing code in this chapter. It uses `time.perf_counter()`, the right clock for measuring elapsed time (Chapter 3 explains why), and it times loading the data separately from building the report.

bench_ch01.py

```
import time

from stonebridge.data import make_parcels, make_pings
from stonebridge.report import build_daily_report
```

```
    t0 = time.perf_counter()
    parcels = make_parcels()
    pings = make_pings()
    t1 = time.perf_counter()
    report = build_daily_report(parcels, pings)
    t2 = time.perf_counter()

    print(f"load data : {t1 - t0:.2f} s")
    print(f"report    : {t2 - t1:.2f} s")
    print(f"revenue ${report.revenue_usd:,.2f}  busiest {report.busiest_depot}  "
          f"fleet {report.fleet_km:,.2f} km")
```

Terminal

```
python bench_ch01.py
```

Output

```
load data : 1.76 s
report    : 8.81 s
revenue $2,430,463.05  busiest D06  fleet 24,561.00 km
```

The business numbers are exactly what they should be: revenue of $2,430,463.05, the busiest depot is D06 (Kestrel Park), and the fleet drove 24,561.00 km. Those three values will be our anchor for the whole book. Any optimization that changes them is a bug, however fast it is.

The timing deserves a closer look. On our test machine (an AMD Ryzen 9 5950X running Windows 11 and Python 3.12) this particular run took 8.81 seconds for the report. When we ran the book's careful benchmark harness, which repeats each measurement and keeps the best result, on the same machine while it was otherwise quiet, the report took **4.78 seconds**. The run above was made while the machine was busy with other jobs, and two other runs of the same script gave 9.24 and 10.58 seconds. Same code, same data, same computer, and a factor of two between measurements. Your numbers will differ again. This is the first lesson of performance work: a single timing is an anecdote. Chapter 3 shows how to make measurements you can trust, and from here on every absolute time in this book is "on our test machine". The **ratios** between versions are what carry over to your machine.

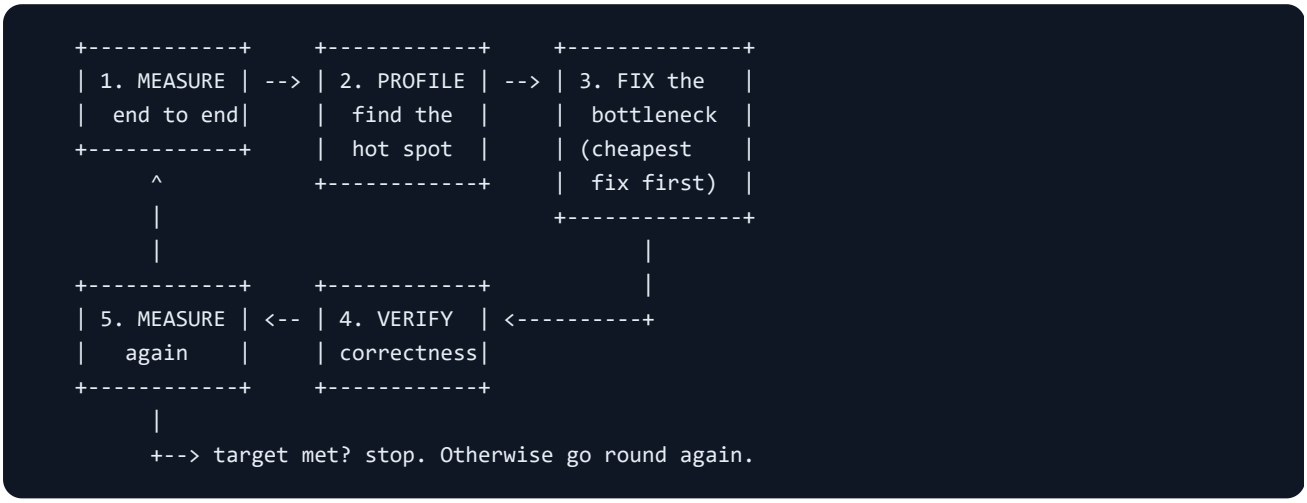
> **WARNING**
>
> Never compare a timing taken today with one taken last week on a different machine, under a different load, or with a different Python version, and conclude that your change made a difference. Compare versions side by side, in the same run, on the same data. Our 4.78 s reference and the 8.81 s above differ only because of background load.

Whichever number you look at, the report is too slow for an operations team that reruns it several times a day. By the end of the book it will run in a small fraction of a second, with exactly the same results. In the book's reference measurements the optimized report, `build_daily_report_fast`, took

about 146 milliseconds on our test machine, roughly 33 times faster than the baseline. Chapter 28 assembles that result step by step.

The Performance Loop

How do we get from 4.78 seconds to 146 milliseconds without breaking anything? Not by guessing. Experienced developers are notoriously bad at predicting where a program spends its time; the part that looks expensive is often cheap, and the real cost hides in a small helper function nobody suspected. The reliable method is a loop with five steps, repeated until the performance target is met.



1. **Measure** the whole task, end to end, with a repeatable benchmark. This number is your baseline. Chapter 3 builds the tools.
2. **Profile** to find out where the time actually goes. A profiler tells you which functions and lines consume the time. Chapters 4 and 5 cover CPU profiling, line-level timing, and memory profiling.
3. **Fix** the biggest bottleneck with the cheapest change that addresses it. "Cheapest" means least code, least risk, and fewest new dependencies, not least CPU time. Often the cheapest fix is a better algorithm or data structure; sometimes it is a cache; sometimes it is NumPy or Numba.
4. **Verify** that the results are still correct by comparing them against the baseline. For integers and strings, exact equality; for floating-point values, a tolerance with `math.isclose` or `np.allclose`. Chapter 2 sets up the checks we use throughout the book.
5. **Measure again**, the same way as in step 1, and record the result. If you have met your target, stop. Otherwise, profile again: fixing one bottleneck often reveals the next one.

Notice that verification comes *before* the second measurement. A fast wrong answer is worthless, and it is surprisingly easy to produce one. Off-by-one errors in vectorized code, a cache that returns stale values, a parallel reduction with a race condition, a float32 conversion that loses precision: every one of these makes code faster and wrong. In this book every optimization is shown next to its correctness check.

> **TIP**
>
> Keep a small performance log while you work: date, change, benchmark result, and whether the correctness check passed. A plain text file is enough. When a colleague asks why the code looks the way it does, or when a later change makes things slower again, that log is invaluable.

Amdahl's Law in Plain Words

Profiling tells you which part of a program is slow. Amdahl's law tells you how much the whole program can gain from speeding up that part. The idea is simple: **the part you do not optimize limits the total speedup**.

Suppose a step takes 60% of the total runtime and you make that step 10 times faster. The other 40% is unchanged. The new runtime, as a fraction of the old one, is the untouched 40% plus the optimized 60% divided by 10:

```
new time = 0.40 + 0.60 / 10 = 0.40 + 0.06 = 0.46 of the original
speedup  = 1 / 0.46 = about 2.17x overall
```

A tenfold improvement on the biggest step produced only about a twofold improvement overall. Even if you made that step infinitely fast, the program could never become more than 1 / 0.40 = 2.5 times faster, because the remaining 40% still has to run. In general, if a fraction *p* of the runtime is sped up by a factor *s*, the overall speedup is `1 / ((1 - p) + p / s)`. The table shows what that means in practice.

Share of runtime (p)	Speedup of that part (s)	Overall speedup
60%	10x	about 2.17x
60%	infinite	2.5x (the ceiling)
90%	10x	about 5.3x
90%	infinite	10x (the ceiling)
20%	100x	about 1.25x
99%	100x	about 50x

Three practical lessons follow from these numbers.

- **Optimize the biggest share first.** A 100x improvement on a step that takes 20% of the time gains you less than a modest improvement on the step that takes 90%. This is why profiling comes before fixing.
- **Large overall speedups require attacking almost everything.** To make the Stonebridge report about 33 times faster, we cannot just speed up one function; nearly all of its work has to move out of the interpreter.

- **Know when to stop.** Once the remaining time is spread thinly across many small pieces, each further gain costs more effort and yields less. That is a good moment to check your target.

Amdahl's law also applies to parallelism. If 90% of a job can run on many cores and 10% must run serially (reading input, combining results, starting worker processes), the job can never be more than 10 times faster, no matter how many cores you buy. Chapter 18 measures this effect, and it explains why our multiprocessing results there are far below the number of cores on the test machine.

Premature Optimization and Its Opposite

You have probably heard the phrase "premature optimization is the root of all evil", usually attributed to Donald Knuth, from a 1974 paper on programming style. The full passage is more balanced than the slogan: it argues that programmers waste enormous effort worrying about the speed of noncritical parts of their programs, and that those efforts hurt readability and maintenance, but it also insists that we should not pass up the opportunities in the small, critical portion of code where speed really matters.

In practice, premature optimization means rewriting readable code with clever tricks before anyone has measured whether it matters, or reaching for multiprocessing on a task that waits on a database. The result is harder to read and frequently no faster.

The opposite mistake is just as real: ignoring performance until it is a crisis. Some design decisions are hard to reverse later. If your data model stores every GPS ping as a separate Python object in a list, and three teams build code around that list, changing it to a compact array later is a large project. A little thought about data volumes and algorithmic complexity at design time (Chapter 6) is not premature; it is engineering.

A useful rule of thumb: **write clear code first, measure it against a realistic workload, and optimize only what the measurements point to**. Keep the clear version around as a baseline, as Stonebridge does with its `baseline.py`, so you always have something to verify against.

> **NOTE**
>
> Big-O complexity is the one kind of "early optimization" that is almost always worth thinking about. Choosing a set instead of a list for membership tests, or sorting once instead of searching repeatedly, costs nothing in readability and can turn hours into seconds. We cover this in Chapters 6 and 7.

A Quick Tour of the Toolbox

The rest of the book is organized around the performance loop. Part I teaches measurement; Parts II to V teach fixes, roughly from cheapest to most involved; Part VI turns everything into a repeatable engineering practice. The table below maps the main techniques to the kind of problem they solve and where we cover them.

Technique	Best for	Chapters
Timing and benchmarking	Knowing how slow, reliably	3, 25
Profiling (cProfile, pyinstrument, tracemalloc)	Finding where time and memory go	4, 5
Better algorithms and data structures	Work that grows too fast with input size	6, 7
Idioms and built-ins	Interpreter overhead in hot loops	8
Caching and precomputation	Repeated work with the same inputs	9
Generators and streaming	Memory pressure, large inputs	10, 23
NumPy vectorization	Numeric loops over many values	11–14
pandas and Polars	Tabular data	15
Numba JIT compilation	Loops NumPy cannot express	16, 17
Multiple processes	CPU-bound work on many cores	18
Threads and asyncio	Waiting on I/O	19
Compiled extensions	Last-resort hot spots, libraries	20
Compact memory layouts	Millions of small objects	21
Fast file formats	Slow loading and saving	22

To give you a sense of where we are heading, here are a few of the reference results the book works toward, all measured on our test machine with the book's benchmark harness. Each is explained, verified, and reproduced in its own chapter.

Stonebridge workload	Technique	Speedup vs baseline
Scan matching, 10,000 scans	list to set	about 600x
Zone ETA lookups, 20,000 calls	lru_cache	about 95x
Nearest depot, 200,000 parcels	NumPy broadcasting	about 22x
Route planning, 2,000 stops	Numba	about 38x
Daily report, end to end	NumPy, computed once	about 33x

The largest speedup comes from a one-line data-structure change, which is why the book teaches algorithms and data structures before compilers.

> **WARNING**
>
> Speedup ratios like these describe one workload on one machine. They are not general facts about the tools. "NumPy is 22 times faster" is false as a general statement; NumPy was about 22 times faster for this nearest-depot computation, with this data, on our test machine. Always measure your own case.

How to Read the Numbers in This Book

Every output block in this book was pasted from a real run on our test machine with CPython 3.12. When a tool was not available to us, such as a C compiler for Cython or mypyc, we say so and show no numbers for it. Headline timings are best-of-N, every speedup is shown next to its correctness check, and all Stonebridge data comes from fixed seeds, so your checksums will match ours exactly even though your timings will not. If your *ratios* differ wildly from ours, that is worth investigating: a busy machine, a different library version, or different data.

What You Need to Follow Along

You need Python 3.12 or newer, the Stonebridge reference package `stonebridge` that ships with the book, and NumPy; later parts add pandas, Polars, DuckDB, Numba, and pyinstrument, which you can install into a virtual environment when you reach them. We write commands the way you would type them (`python bench_ch01.py`, `python -m timeit ...`); on Windows you may need `py` instead of `python`. Chapter 2 shows the project layout.

> **TIP**
>
> Close heavy applications (video calls, browsers with many tabs, games, large builds) before you benchmark, and plug in your laptop. Laptops on battery often lower their CPU speed, which can make the same code run noticeably slower and turn a fair comparison into an unfair one.

Try It Yourself

1. Run `bench_ch01.py` three times in a row on your machine and write down the three report times. How much do they vary? Then close every other program you can and run it three more times. Did the spread shrink?
2. Use `dis.dis` on a short function of your own, for example one that sums the squares of a list in a `for` loop. Count the bytecode instructions executed per iteration, and compare with `dis.dis` of `sum(x * x for x in values)`.
3. Apply Amdahl's law: a program spends 30% of its time reading a file, 50% computing, and 20% writing a report. You make the computation 25 times faster. What is the overall speedup? What is the best possible speedup if you could make the computation take no time at all?