BEFORE

```
def p(d,t,x):
  if d:
    if t>24:
      r=x*.5
```

AFTER

```
def late_fee(
  price,
  hours):
  ...
```

FOR NEW DEVELOPERS

Clean Code for New Developers

Write code that is easy to read, change, and trust

Before-and-after examples, refactoring, and reviews —
with exercises and solutions in every chapter

NAMING · FUNCTIONS · REFACTORING · REVIEWS

THENEXBOOKS

Clean Code for New Developers

Write Code That Is Easy to Read, Change, and Trust
Practical Principles with Before-and-After Examples and Exercises

First Edition

THENEXBOOKS

Clean Code for New Developers
Write Code That Is Easy to Read, Change, and Trust

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 and exercise solutions may be used freely for learning and 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. Examples use Python 3.10 or later, with short comparisons in other languages; tools evolve, so consult the official documentation for the versions you use.

Python is a registered trademark of the Python Software Foundation. Java is a trademark of Oracle and/or its affiliates. JavaScript is a trademark of Oracle Corporation in the United States and other countries. Other product and tool names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation.

Maplewood Tutors, the running example in this book, and its people are fictional. Any resemblance to real organizations or people is coincidental.

Preface

Your code works. The tests pass, the feature ships, and you move on. A few weeks later someone — often you — has to change it, and suddenly nothing is obvious. What does `d2` mean? Why does this function take seven arguments? Which of these three nearly identical blocks is the one that actually runs? Working code is the starting line, not the finish. Code is read far more often than it is written, and code that is hard to read is hard to change safely.

Clean code is code that the next person can understand quickly and change with confidence. It is not about clever tricks or following rules for their own sake. It is a set of habits — good names, small focused functions, clear control flow, honest error handling, sensible structure, and tests — that make everyday work faster and less stressful, especially on a team.

This book is written for developers early in their careers. Every principle is shown with a realistic **before** (working but messy) and **after** (clean) version, with an honest explanation of the trade-offs. We follow one small application, written in a hurry by a former intern, and improve it chapter by chapter. Each chapter ends with exercises and worked solutions so you can practice the habits until they become your own.

Who This Book Is For

- **Students and bootcamp graduates** who can write working programs and want to write professional-quality code.
- **Junior developers** in their first years on a team, preparing for code reviews and shared codebases.
- **Self-taught developers** who want to understand the conventions experienced teams expect.
- **Mentors and team leads** looking for a practical, shared vocabulary to use in reviews.

What You Will Learn

- How to name things, write small functions, and use comments that help rather than hurt.
- Clear formatting, control flow, data handling, error handling, and avoiding duplication.
- How to organize modules and classes, manage dependencies and side effects, and keep designs simple.
- How to recognize code smells, refactor safely in small steps, and use tests as a safety net.
- How to work with legacy code, write clear commits and pull requests, and give and receive code reviews.
- How to review AI-generated code, balance performance with readability, and apply the same ideas in other languages.

How This Book Is Organized

Part I — Why Clean Code Matters (Chapters 1–5) covers reading code, naming, functions, and comments. **Part II — Writing Clear Code** (Chapters 6–10) covers style, control flow, data, errors, and duplication. **Part III — Structuring Code** (Chapters 11–15) covers modules, classes, dependencies, side effects, and simple design. **Part IV — Keeping Code Clean Over Time** (Chapters 16–20) covers smells, refactoring, tests, legacy code, and version control. **Part V — Clean Code in a Team** (Chapters 21–25) covers reviews, conventions, interfaces, AI assistants, and performance. **Part VI — Practice** (Chapters 26–28) applies everything across languages, in a workshop, and in a complete case study. The appendices provide a cheat sheet, templates, a checklist, and a glossary.

Each chapter ends with **Practice Exercises**, **Selected Solutions**, and a **Key Takeaways** box.

The Running Example: Maplewood Tutors

Throughout the book we clean up the in-house app of **Maplewood Tutors**, a fictional tutoring center. It books sessions, sends reminders, produces invoices, and reports on student progress — and it was written quickly, so it has vague names, long functions, magic numbers, copy-pasted logic, and silent errors. Priya, the lead developer, and Jonah, a junior developer, improve it step by step, and in the final chapter we complete the refactoring with tests.

Conventions

Code appears in `monospace`. Examples use Python 3.10 or later, with short JavaScript and Java snippets where conventions differ. "Before" and "after" versions are labeled. Exercises are marked **[Easy]**, **[Medium]**, or **[Challenge]**.

> **TIP**
> A recommended practice, or a "Try It" prompt to pause and experiment.

> **NOTE**
> Background information or a useful aside.

> **WARNING**
> A habit that makes code harder to read, change, or trust.

Contents

Preface

PART I — WHY CLEAN CODE MATTERS

1 What Clean Code Is (and Isn't): Readability, Change, and Teamwork

2 How Developers Read Code

3 Naming Things Well: Variables, Functions, Classes, and Files

4 Functions That Do One Thing: Size, Parameters, and Levels of Abstraction

5 Comments and Documentation: When They Help and When They Hurt

PART II — WRITING CLEAR CODE

6 Formatting and Style: Consistency, Linters, and Formatters

7 Clear Control Flow: Conditionals, Guard Clauses, and Loops

8 Data and Constants: Magic Numbers, Data Structures, and Types

9 Error Handling Done Right: Exceptions, Validation, and Failing Clearly

10 Avoiding Duplication: DRY and When Duplication Is Fine

PART III — STRUCTURING CODE

11 Organizing Files and Modules

12 Small, Focused Classes

13 Dependencies and Coupling: Keeping Parts Independent

14 Side Effects, State, and Immutability

15 Simple Design: KISS, YAGNI, and SOLID for Beginners

PART IV — KEEPING CODE CLEAN OVER TIME

16 Code Smells: Recognizing Problems Early

17 Refactoring Step by Step: Small, Safe Changes

18 Tests That Keep Code Clean

19 Working with Legacy Code You Didn't Write

20 Version Control Hygiene: Commits, Branches, and Pull Requests That Read Well

PART V — CLEAN CODE IN A TEAM

21 Code Reviews: Giving and Receiving Feedback

22 Team Conventions, Style Guides, and Automation

23 Designing Readable Interfaces and APIs

24 Clean Code with AI Assistants: Prompting, Reviewing, and Owning Generated Code

25 Performance vs Readability: Making Sensible Trade-Offs

PART VI — PRACTICE

26 Clean Code Across Languages: Python, JavaScript, and Java

27 Workshop: Cleaning Up Real Messy Code

28 Case Study: Refactoring the Maplewood Tutors App

APPENDICES

A Clean Code Cheat Sheet

B Templates and Reference

C Clean Code Checklist

D Glossary

PART I

Why Clean Code Matters

What clean code is and why it matters, how developers read code, and the everyday habits of naming, small functions, and useful comments.

1 · What Clean Code Is (and Isn't)
2 · How Developers Read Code
3 · Naming Things Well
4 · Functions That Do One Thing
5 · Comments and Documentation

CHAPTER 1 · PART I — WHY CLEAN CODE MATTERS

What Clean Code Is (and Isn't): Readability, Change, and Teamwork

Working code is only the starting line. In this chapter you will learn what makes code "clean," why it matters the moment a second person touches your work, and which popular ideas about clean code are myths that can make things worse.

A First Week at Maplewood Tutors

Meet Jonah. He has just joined Maplewood Tutors, a small tutoring center, as its second developer. The center runs on an in-house Python app that handles students, tutors, subjects, session bookings, monthly invoices, reminder messages, and progress reports. A former intern wrote most of it in a hurry, over one busy summer, and it has been running ever since.

On Monday morning, Ms. Alvarez, the center manager, stops by with a simple request: "Families keep asking whether the sibling discount applies to cancelled sessions. Can you check what the app does today?" Priya, the lead developer, points Jonah to the billing module. He opens `maplewood/billing.py` and finds this:

maplewood/billing.py (before)

```
def calc(s, t, l, sib, c):
    # calc price
    if l == 1:
        r = 40
    elif l == 2:
        r = 55
    else:
        r = 70
    p = r * t / 60
    if sib == True:
        p = p - p * 0.1
    if c == True:
        p = p * 0.5
    return round(p, 2)
```

The function works. It has been producing correct invoices for months. Yet Jonah needs twenty minutes and three questions to Priya before he can answer Ms. Alvarez. What is `s`? (A student name, it turns out, and it is never used.) What is `t`? (Minutes.) What does level `3` mean? (Exam preparation.) What is `c`? (Whether the session was cancelled less than 24 hours before it started.) The comment "calc price" repeats the function name and adds nothing.

Now look at the same logic after a short cleanup:

maplewood/billing.py (after)

```
HOURLY_RATES = {"primary": 40, "secondary": 55, "exam-prep": 70}
SIBLING_DISCOUNT = 0.10
LATE_CANCELLATION_SHARE = 0.50


def session_price(level, minutes, has_sibling=False, cancelled_late=False):
    """Return the price in dollars for one tutoring session."""
    hourly_rate = HOURLY_RATES[level]
    price = hourly_rate * minutes / 60
    if has_sibling:
        price = price * (1 - SIBLING_DISCOUNT)
    if cancelled_late:
        price = price * LATE_CANCELLATION_SHARE
    return round(price, 2)


print(session_price("primary", 60))                       # 40.0
print(session_price("secondary", 90, has_sibling=True))  # 74.25
print(session_price("exam-prep", 60, cancelled_late=True))  # 35.0
```

Jonah can now answer the manager's question by reading four lines: yes, a late-cancelled session for a family with siblings gets both the 10% discount and the 50% charge. The business rules are visible as named values at the top of the file. The parameters say what they are. The `True` / `False` flags have default values, so a normal booking is simply `session_price("primary", 60)`.

A few Python features appear here for the first time, so let us name them. A **dictionary** (the curly braces in `HOURLY_RATES`) maps keys such as `"primary"` to values such as `40`. A **default value** such as `has_sibling=False` lets callers leave out an argument. The triple-quoted string right under the `def` line is a **docstring**, a short description that tools and editors show when you hover over the function. And writing constants in `UPPER_CASE` is the Python convention for "this value is not meant to change while the program runs."

> **NOTE**
>
> The cleanup changed two behaviors, and it is important to be honest about them. First, callers must now pass `"secondary"` instead of `2`, so every call site changes. Second, the old code quietly charged $70 an hour for *any* unknown level, including a typo like `4`. The new code raises a `KeyError` for an unknown level. That is a deliberate bug fix: a loud error is better than a silently wrong invoice. For the three real levels and the two real session lengths, both versions return exactly the same prices.

A Working Definition of Clean Code

People have argued about the phrase "clean code" for decades, and you will find many definitions. In this book we use a practical one built from three questions you can ask about any piece of code:

1. **Can someone read it?** A developer who did not write the code can understand what it does, and why, without asking the original author.
2. **Can someone change it?** A new requirement can be added in one obvious place, without breaking unrelated features.
3. **Can someone trust it?** When it works, you can see why; when it fails, it fails loudly and clearly, and there are checks (such as tests) that catch mistakes.

Notice that all three questions mention *someone*. Clean code is not a property of code alone; it is a property of code in relation to the people who work with it. The intern's `calc` was perfectly clear to the intern on the day it was written. It stopped being clear the moment the intern left and Jonah arrived. That includes future you: the person who opens your code six months from now remembers much less than you think.

Let us look at each of the three questions more closely, using the Maplewood code.

Readability: Code Is Read Far More Than It Is Written

You write a line of code once. After that, it is read again and again: when you debug it, when a teammate reviews it, when someone adds a feature nearby, when someone searches the codebase for a bug, and when a new hire tries to learn how the system works. This is one of the oldest observations in software, and you will see it for yourself within weeks of joining any team.

Readable code lets a reader build a correct picture in their head quickly. Compare the two versions above line by line:

Before	After	What the reader gains
`calc`	`session_price`	What is calculated
`t`	`minutes`	The unit
`l == 3`	`"exam-prep"`	No code table to memorize
`0.1`	`SIBLING_DISCOUNT`	Why the number exists
`sib == True`	`has_sibling`	Reads like English

None of these changes makes the computer's job easier. Python does not care whether a variable is called `t` or `minutes`. Every change is for the human reader. That is the heart of clean code: **the computer is the easiest audience you will ever write for**; your teammates are the hard one.

Readability is also about what the reader does *not* have to do. In the BEFORE version, the reader must hold a small table in their head ("1 is primary, 2 is secondary, anything else is exam prep"). In the AFTER version, the table is written down once, at the top, with names. Chapter 2 explains why this matters so much: human working memory is small, and every detail you force a reader to remember pushes something else out.

> **TRY IT**
>
> Open a script you wrote at least a month ago. Before running it, read one function and write down in one sentence what it does. Then run it. Were you right? Which names or numbers slowed you down? Keep that list; you will fix those items in Chapter 3.

Change: Clean Code Is Cheap to Change

Requirements change constantly in real software. Suppose Maplewood decides to start a university-level program at $80 per hour. How much work is that?

In the BEFORE version, you must invent a new numeric code (`4`), add another `elif` branch, and then search the whole codebase for other places that turn codes into rates or names, because the numbers 1, 2 and 3 probably appear in reports and reminders too. Worse, until you add the branch, level `4` is silently charged the exam-prep rate, so a half-finished change produces wrong invoices instead of an error.

In the AFTER version, the change is one line:

```
HOURLY_RATES = {
    "primary": 40,
    "secondary": 55,
    "exam-prep": 70,
    "university": 80,
}

print(session_price("university", 90, has_sibling=True))  # 108.0
```

This is what people mean when they say clean code is "easy to change." It does not mean the code will never need editing. It means that each kind of change has one obvious home, and that making the change does not require understanding the whole program first.

The hidden cost of messy code

Messy code rarely causes a dramatic failure on the day it is written. Its cost appears later, as friction. Every change takes a little longer than it should. Developers become nervous about touching certain files. Bugs appear in places far from the change that caused them. Teams often describe this build-up with the metaphor of **technical debt**: taking a shortcut now is like borrowing time, and the "interest" is the extra effort every later change costs until the shortcut is cleaned up.

Debt is not always bad. A business sometimes has good reasons to borrow, and a team sometimes has good reasons to ship a quick version first. The problem is debt that nobody notices, nobody records, and nobody pays back. Part IV of this book, especially Chapter 16 and Chapter 17, shows how to recognize and pay down that debt in small, safe steps.

Trust: Clean Code Fails Clearly

The third question is the one new developers most often forget. Code you can trust tells you when something is wrong. Here is another function from the intern's codebase:

maplewood/utils.py (before)

```
RATES = {"primary": 40, "secondary": 55, "exam-prep": 70}


def load_rate(level):
    try:
        return RATES[level]
    except:
        return 0


print(load_rate("exam prep"))  # 0   (typo: space instead of hyphen)
```

A `try` / `except` block runs the code under `try`, and if an error (an **exception**) happens, runs the code under `except` instead. A bare `except:` with no exception type catches *every* error. Here, a simple typo in the level name produces a rate of zero, which produces a free session, which nobody notices until the monthly invoices go out and the numbers look strange. The code "never crashes," and that is exactly the problem.

maplewood/utils.py (after)

```
RATES = {"primary": 40, "secondary": 55, "exam-prep": 70}


def hourly_rate(level):
    if level not in RATES:
        raise ValueError(f"Unknown level: {level}")
    return RATES[level]


hourly_rate("exam prep")
# ValueError: Unknown level: exam prep
```

The `raise` statement creates an error on purpose, and the `f"..."` string (an **f-string**) inserts the value of `level` into the message. Now the same typo stops the program at the exact line where the mistake is, with a message that says what went wrong. This is a small change in code and a big change in trust. Chapter 9 covers error handling in depth; for now, remember that **silent failures are the enemy of trust**.

> **WARNING**
>
> "It never crashes" is not a compliment if it achieves that by hiding errors. Code that swallows exceptions with a bare `except:` and returns a default like `0`, `""`, or `None` turns an obvious bug into a subtle one. Subtle bugs reach customers; obvious bugs get fixed during development.

Trust also comes from tests. When the billing code has a handful of automated checks, such as "a 90-minute secondary session with a sibling costs $74.25," you can change the code and know within seconds whether you broke something. Chapter 18 is devoted to tests; you will write a tiny version of one in this chapter's exercises.

Teamwork: Clean Code Is a Team Sport

Most professional code is written by teams, read by other teams, and maintained long after its authors have moved on. That changes what "good" means. A brilliant solution that only its author can understand is a risk for the team. If that author is on holiday when the invoice job breaks, nobody else can fix it quickly.

Clean code helps teams in several concrete ways:

- **Reviews go faster.** When Priya reviews Jonah's pull request, she can focus on whether the logic is right instead of decoding what `p` and `r` mean. Chapter 21 covers code reviews.
- **Knowledge spreads.** Clear names and small functions let anyone on the team work on billing, not only the one person who "knows that file."
- **Onboarding is easier.** A new hire can learn the business rules by reading the code, because the rules are written down as named constants and well-named functions.
- **Conventions reduce arguments.** When a team agrees on formatting, naming and structure (Chapter 6 and Chapter 22), individual style stops being a debate in every review.

Teamwork also means clean code is partly about humility. The goal is not to show how clever you are; it is to make the next person's job easier. Very often, the next person is you.

How the Cleanup Actually Happened

It is tempting to think Jonah deleted `calc` and typed `session_price` from scratch. He did not, and neither should you. Rewriting from scratch is risky, because it is easy to drop a rule you did not notice. Instead, Priya showed him how to move in small steps, running the function after each one to confirm the prices had not changed:

1. **Rename the function and parameters.** `calc` became `session_price`, `t` became `minutes`, and so on. Nothing else changed yet.
2. **Remove the unused parameter.** The student name `s` was never read, so it went away, and every caller got one argument shorter.
3. **Name the numbers.** `0.1` and `0.5` became `SIBLING_DISCOUNT` and `LATE_CANCELLATION_SHARE`.
4. **Replace the level codes.** The `if`/`elif` chain became a dictionary lookup, and callers switched from `2` to `"secondary"`. This was the only step that touched other files.
5. **Simplify the flags.** `if sib == True:` became `if has_sibling:`, and both flags gained defaults of `False`.

6. **Add the docstring.** One sentence stating what the function returns and in which unit.

Each step was small enough to check in a minute and small enough to undo if something went wrong. That rhythm, change a little and check, is the core skill behind all refactoring. Chapter 17 turns it into a repeatable method, and Chapter 18 shows how automated tests make the "check" part fast.

What Clean Code Isn't

Because "clean code" is a popular phrase, it has collected a number of myths. Some of them can make code worse. Let us look at the most common ones.

Myth 1: Clean code means short code

Fewer lines are not automatically better. Here is a very short function that totals a list of sessions, where each session is a tuple of hourly rate, minutes, and a sibling flag:

```
def total(bs): return sum(r*m/60*(.9 if s else 1) for r,m,s in bs)


print(total([(40, 60, False), (55, 90, True)]))  # 114.25
```

It fits on one line, but a reader must unpack a generator expression, a conditional expression, and three one-letter names all at once. The longer version is easier to read, easier to debug (you can print `price` inside the loop), and easier to change:

```
def total_price(sessions):
    total = 0
    for hourly_rate, minutes, has_sibling in sessions:
        price = hourly_rate * minutes / 60
        if has_sibling:
            price = price * 0.9
        total = total + price
    return total


print(total_price([(40, 60, False), (55, 90, True)]))  # 114.25
```

The line `for hourly_rate, minutes, has_sibling in sessions:` uses **tuple unpacking**: each tuple in the list is split into three named variables. Clean code is code that is *as short as it can be while staying clear*, not shorter.

Myth 2: Clean code means lots of classes and patterns

The opposite mistake is building elaborate structure for a simple problem. Imagine a well-meaning developer "cleaning up" the sibling discount like this:

```
class DiscountStrategy:
    def apply(self, price):
        raise NotImplementedError


class SiblingDiscountStrategy(DiscountStrategy):
    def apply(self, price):
        return price * 0.9


class NoDiscountStrategy(DiscountStrategy):
    def apply(self, price):
        return price


class DiscountStrategyFactory:
    def create(self, has_sibling):
        if has_sibling:
            return SiblingDiscountStrategy()
        return NoDiscountStrategy()


print(DiscountStrategyFactory().create(True).apply(60))  # 54.0
```

Four classes to multiply by 0.9. Each class is small and tidy on its own, but a reader now has to jump between four definitions to understand one rule. For Maplewood's current needs, a single function is cleaner:

```
SIBLING_DISCOUNT = 0.10


def apply_sibling_discount(price, has_sibling):
    if has_sibling:
        return price * (1 - SIBLING_DISCOUNT)
    return price


print(apply_sibling_discount(60, True))  # 54.0
```

Patterns and classes are tools, and there are situations where a structure like the one above earns its keep, for example when there are many discount types that change often. Chapter 15 discusses how to decide, using the ideas known as KISS ("keep it simple") and YAGNI ("you aren't gonna need it").

Myth 3: Clean code is just formatting

Consistent indentation, spacing, and line length matter, and tools can handle most of it automatically (Chapter 6). But perfectly formatted code can still have meaningless names, 200-line functions, and silent errors. Formatting is the floor, not the ceiling.

Myth 4: Clean code must be perfect before it ships

Clean code is a direction, not a finish line. Real teams ship under deadlines, and a working feature today is often worth more than a beautiful one next month. The skill is knowing which shortcuts are cheap to fix later and which ones will spread. A common guideline, often called the **Boy Scout rule**, is to leave code a little cleaner than you found it: rename one confusing variable, extract one helper, delete one dead line, each time you pass through. Small improvements add up.

Myth 5: There is one correct way

Experienced developers disagree about many details: how long a function should be, whether a comment is helpful, how to name a module. This book gives you strong defaults, but it also shows the trade-offs. When your team has a convention that differs from ours, follow the team. Consistency within a codebase beats personal preference.

Clean code is...	Clean code is not...
Easy to read	As short as possible
Easy to change	Full of patterns
Honest about errors	"Never crashes"
Consistent	Only formatting
Improved over time	Perfect on day one

Clean Code and Trade-Offs

Clean code principles are guidelines, not laws. Every guideline in this book has situations where it does not apply, and part of growing as a developer is learning to recognize them. A few examples you will meet:

- **Prototypes and throwaway scripts.** A one-off script to rename 300 files on your laptop does not need the same care as the billing module. The risk is when a "throwaway" script quietly becomes part of the product, which is exactly how the Maplewood app was born.
- **Performance-critical code.** Occasionally the clearest code is too slow, and a less obvious version is justified. Chapter 25 shows how to make that trade-off sensibly, and how to keep the fast version as readable as possible.
- **Deadlines.** Sometimes you ship the quick version. When you do, leave a clear note (a ticket, or a comment with a reason) so the debt is visible.
- **Team conventions.** If the team writes code a certain way, a "cleaner" personal style that breaks consistency can make the codebase harder to read overall.

> **TIP**
>
> When you are unsure whether a change makes code cleaner, ask a concrete question: "Would a new teammate understand this faster?" or "Would the next likely change be easier?" If the answer is not clearly yes, the change may be a matter of taste rather than an improvement.

Meet the Maplewood Codebase

Throughout this book we will clean up the Maplewood Tutors app one piece at a time. Here is its layout as the intern left it:

```
maplewood/
    models.py      # Student, Tutor, Booking data (mostly plain dicts)
    bookings.py    # create, cancel and list session bookings
    billing.py     # prices, discounts and monthly invoices
    reminders.py   # reminder messages for students and tutors
    reports.py     # progress reports for parents
    utils.py       # a bit of everything
```

And here are the business rules the code must follow. Every chapter uses the same rules, so you can check examples against them:

Rule	Value
Session length	60 or 90 minutes
Primary rate	$40 per hour
Secondary rate	$55 per hour
Exam-prep rate	$70 per hour
Sibling discount	10%
Cancel under 24 hours	50% charged
Invoices	Monthly

The people are Priya (lead developer), Jonah (junior developer), and Ms. Alvarez (center manager). Students include Ava, Leo, Mina, and Omar; the tutors include Mr. Chen and Ms. Okafor. By Chapter 28, you will have seen the whole app transformed, one small and safe step at a time.

> **NOTE**
>
> Every BEFORE example in this book is code that *works*. That is deliberate. Clean code is rarely about fixing broken programs; it is about making working programs understandable. When a cleanup also fixes a bug, as with the unknown-level case above, we say so explicitly.

How to Read This Book

Each chapter follows the same rhythm. We show a piece of real-looking messy code, explain what makes it hard to work with, clean it up, and explain the reasoning and the trade-offs. Then you practice. The exercises at the end of each chapter use the same Maplewood codebase, and selected solutions follow so you can check your work.

You will get the most out of the book if you type the examples yourself and run them. Python 3.10 or newer and a plain text editor are all you need; everything uses the standard library. When you see a line like `# 74.25` after a `print` call, that comment shows the output you should see.

Finally, remember that clean code is a skill, not a talent. Nobody writes clean code on the first draft every time, including very experienced developers. What they do is *revise*: write something that works, then read it as a stranger would, and improve it. That revising habit is what this book teaches.

> **TRY IT**
>
> Copy the BEFORE `calc` function and the AFTER `session_price` function into one file. Call both with a 90-minute secondary session for a family with siblings. Then call `calc` with level `7` and `session_price` with level `"expert"`. Which behavior would you rather discover during testing?

Practice Exercises

1. **[Easy]** List every name in the BEFORE `calc` function that forced Jonah to guess its meaning. For each, suggest a clearer name.
2. **[Easy]** Without running it, predict what `calc("Omar", 60, 4, False, False)` returns. Then run it. Explain why this result is dangerous for Maplewood.
3. **[Easy]** Which of these statements are myths? (a) Clean code is as short as possible. (b) Clean code fails loudly when something is wrong. (c) Clean code uses design patterns everywhere. (d) Clean code follows the team's conventions. Explain each answer in one sentence.
4. **[Medium]** Clean up this reminder function from `maplewood/reminders.py`. Keep the output identical for valid bookings.

```
def rem(b):
    s = "Hi " + b[0] + ", reminder: " + b[1] + " with " + b[2] + " on " + b[3]
    if b[4] == 90:
        s = s + " (90 min)"
    else:
        s = s + " (60 min)"
    return s


print(rem(["Ava", "Math", "Mr. Chen", "Monday", 60]))
# Hi Ava, reminder: Math with Mr. Chen on Monday (60 min)
```

5. **[Medium]** Add a "university" level at $80 per hour to both `calc` and `session_price`. Count how many lines you changed in each, and note which version would give a wrong price if you forgot a step.
6. **[Medium]** This function hides errors. Rewrite it so an unknown tutor produces a clear error instead of an empty string.

```
def get_tutor_email(tutors, name):
    try:
        return tutors[name]["email"]
    except:
        return ""
```

7. **[Medium]** Rewrite the one-line `total` function from Myth 1 so that it uses the named constant `SIBLING_DISCOUNT` and descriptive names. Confirm it still returns `114.25` for the sample list.
8. **[Challenge]** Write a short script that proves `calc` and `session_price` return the same price for every valid combination: three levels, two session lengths, sibling yes or no, late cancellation yes or no. Use `assert` so the script stops on the first mismatch.
9. **[Challenge]** Ms. Alvarez asks for a rule change: the sibling discount should *not* apply to late-cancelled sessions. Make the change in `session_price`, and write two `print` checks showing the new behavior. Explain how you would tell the team that this is a behavior change.
10. **[Easy]** Pick one function you wrote recently. Ask the three questions from this chapter (read, change, trust) and write one sentence for each describing a concrete improvement.

Selected Solutions

Exercise 2

`calc("Omar", 60, 4, False, False)` returns `70.0`. Level `4` does not match `1` or `2`, so it falls into the `else` branch and gets the exam-prep rate. There is no level 4, so this is a mistake in the calling code, but the function hides it by producing a plausible-looking price. The invoice will be wrong, and nothing will tell anyone. A clean version rejects unknown levels.

Exercise 4 (Medium)

```
def reminder_text(student, subject, tutor, day, minutes):
    return f"Hi {student}, reminder: {subject} with {tutor} on {day} ({minutes} min)"


print(reminder_text("Ava", "Math", "Mr. Chen", "Monday", 60))
# Hi Ava, reminder: Math with Mr. Chen on Monday (60 min)
print(reminder_text("Leo", "Physics", "Ms. Okafor", "Friday", 90))
# Hi Leo, reminder: Physics with Ms. Okafor on Friday (90 min)
```