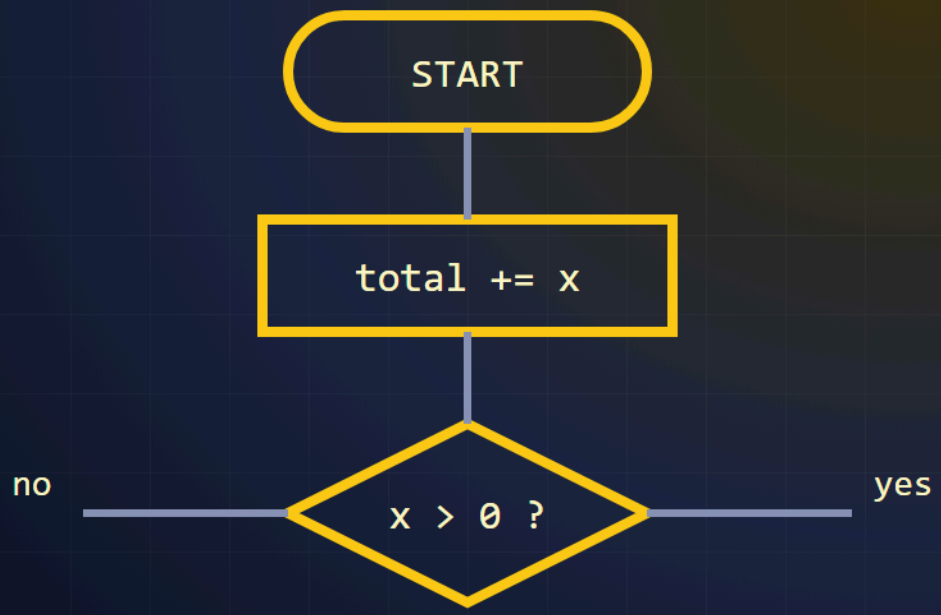


THINK LIKE A PROGRAMMER

Learn Programming Logic with Practical Exercises

Solve problems step by step — from pseudocode to Python

Pseudocode, flowcharts, and Python —
with exercises and solutions in every chapter

PSEUDOCODE · FLOWCHARTS · PYTHON · EXERCISES

THENEXBOOKS

Learn Programming Logic with Practical Exercises

Think Like a Programmer, Solve Problems Step by Step
From Pseudocode and Flowcharts to Working Python Programs

First Edition

THENEXBOOKS

Learn Programming Logic with Practical Exercises
Think Like a Programmer, Solve Problems Step by Step

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; always consult the official documentation for the version you install.

Python is a registered trademark of the Python Software Foundation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation.

Sparrow Street Library, the running example in this book, and its members are fictional. Any resemblance to real organizations or people is coincidental.

Preface

Learning to program is not really about memorizing a language. It is about learning to think in a particular way: breaking a problem into small steps, deciding what the computer should do in each situation, repeating work without repeating yourself, and checking that the result is right. That way of thinking is called **programming logic**, and once you have it, every programming language becomes much easier to learn.

Many beginners start by copying code they do not fully understand. It works until the problem changes a little, and then they are stuck. This book takes the opposite approach. We start with the logic itself — in plain language, pseudocode, flowcharts, and hand-traced tables — and only then express it in Python, a friendly language that reads almost like English. Every idea is followed by practice, because logic is a skill you build by doing.

Each chapter ends with graded exercises, from quick warm-ups to real challenges, and fully worked solutions for a selection of them. Work through them with a pencil and a Python prompt beside you. By the end of the book you will be able to take a problem you have never seen before and turn it into a program that works, handles bad input, and is easy to test.

Who This Book Is For

- **Complete beginners** who have never written a program and want a solid foundation.
- **Students** taking a first programming course who want more practice and clearer explanations.
- **Career changers and self-taught learners** who can follow tutorials but struggle to solve problems on their own.
- **Anyone preparing for a coding course or interview** who wants to strengthen problem-solving skills.

What You Will Learn

- How to break problems down and describe solutions with pseudocode and flowcharts.
- Variables, operators, decisions, Boolean logic, and loops — and how to trace them by hand.
- Functions, strings, lists, dictionaries, and sets, and how to choose the right structure.
- Classic patterns: accumulators, searching, sorting, recursion, and working with grids.
- How to validate input, debug logic errors, write test cases, and reason about efficiency.
- Problem-solving strategies, mini projects, and a complete case study that ties everything together.

How This Book Is Organized

Part I — Thinking Like a Programmer (Chapters 1–5) introduces problem decomposition, algorithms, tracing, and variables. **Part II — Core Logic** (Chapters 6–10) covers operators, decisions, Boolean logic, and loops. **Part III — Organizing Logic** (Chapters 11–15) introduces functions and data structures. **Part IV — Classic Problem-Solving Patterns** (Chapters 16–20) teaches the patterns you will use again and again. **Part V — Writing Reliable Logic** (Chapters 21–25) covers validation, debugging, testing, efficiency, and state. **Part VI — Putting It All Together** (Chapters 26–28) brings everything together in strategies, projects, and a case study. The appendices provide a cheat sheet, pattern templates, a debugging checklist, and a glossary.

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

The Running Example: Sparrow Street Library

Throughout the book we write small programs for **Sparrow Street Library**, a fictional community lending library run by volunteers. Late fees, borrowing limits, book searches, shelf sorting, reading-challenge statistics, and room bookings give every concept a practical job. In the final chapter these pieces come together as a small working library system.

Conventions

Code appears in `monospace`, with program output shown in comments or in a separate block. Exercises are labeled **[Easy]**, **[Medium]**, or **[Challenge]**. Try each one before reading the solution — the struggle is where the learning happens.

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

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

> **WARNING**
> A common mistake that causes wrong results or confusing errors.

Contents

Preface

PART I — THINKING LIKE A PROGRAMMER

1 What Programming Logic Is and Why It Matters

2 Breaking Problems Down: Decomposition, Patterns, and Abstraction

3 Algorithms in Plain Language: Pseudocode and Flowcharts

4 Getting Set Up: Running Python and Tracing Code by Hand

5 Values, Variables, and Types

PART II — CORE LOGIC

6 Expressions and Operators: Arithmetic, Comparison, and Logic

7 Making Decisions: if, elif, else, and Nested Conditions

8 Boolean Logic in Depth: Truth Tables, De Morgan's Laws, and Simplifying Conditions

9 Repetition with while Loops: Designing Loops That End

10 Counting Loops: for, range, and Nested Loops

PART III — ORGANIZING LOGIC

11 Functions: Breaking Programs into Reusable Pieces

12 Parameters, Return Values, and Scope

13 Working with Text: Strings and Character Logic

14 Lists: Storing and Processing Collections

15 Dictionaries, Sets, and Choosing the Right Structure

PART IV — CLASSIC PROBLEM-SOLVING PATTERNS

16 Accumulators: Counting, Summing, Averages, Minimum, and Maximum

17 Searching: Linear Search, Binary Search, and Finding Things

18 Sorting: How Sorting Algorithms Work

19 Recursion: Solving Problems with Smaller Versions of Themselves

20 Grids and Tables: Working with Two-Dimensional Data

PART V — WRITING RELIABLE LOGIC

21 Validating Input and Handling Bad Data

22 Debugging: Finding and Fixing Logic Errors

23 Testing Your Logic: Test Cases, Edge Cases, and Simple Automated Tests

24 Efficiency: An Intuitive Introduction to Big-O

25 State and Simulation: Step-by-Step Processes and State Machines

PART VI — PUTTING IT ALL TOGETHER

26 Problem-Solving Strategies: From Puzzle to Program

27 Mini Projects: Games and Everyday Tools

28 Case Study: Building the Sparrow Street Library System

APPENDICES

A Logic and Python Cheat Sheet

B Problem-Solving Patterns and Templates

C Debugging and Self-Review Checklist

D Glossary

PART I

Thinking Like a Programmer

What programming logic is, how to break problems down, how to describe solutions before coding, and the first building blocks: values and variables.

1 · What Programming Logic Is and Why It Matters
2 · Breaking Problems Down
3 · Algorithms in Plain Language
4 · Getting Set Up
5 · Values, Variables, and Types

CHAPTER 1 · PART I — THINKING LIKE A PROGRAMMER

What Programming Logic Is and Why It Matters

Before you learn any programming language, you need a way of thinking: breaking a task into precise, ordered steps that a machine can follow without guessing. In this chapter you will discover what programming logic is, meet the three building blocks every program is made from, and solve your first real problem for the Sparrow Street Library.

A Tuesday Afternoon at Sparrow Street Library

Sparrow Street Library is a small community lending library run entirely by volunteers. It has a few thousand books, a front desk with one old laptop, and a notebook where volunteers write down who borrowed what. Every Tuesday afternoon, a volunteer named Dara sits at the desk and works out late fees for books that came back after their due date.

The library's rules are simple and are written on a card taped to the desk:

- A book may be borrowed for **14 days**.
- If it comes back late, the fee is **$0.25 per day late**.
- The fee for a single book is **capped at $5.00**, no matter how late it is.
- Members **under 12** may borrow at most **3 books** at a time; everyone else may borrow **5**.

Dara is careful, but on a busy afternoon mistakes creep in. Sometimes a fee is calculated from the wrong date. Sometimes the $5.00 cap is forgotten and a member is asked to pay $7.50 for a book that was a month late. Sometimes Ben, another volunteer, applies the rules slightly differently from Dara, and members notice that the fee depends on who is at the desk.

None of these problems are really about arithmetic. Everyone at the library can multiply by 0.25. The problems come from **steps that are unclear, skipped, or done in a different order**. That is exactly the kind of problem programming logic solves. If we can describe the late-fee process so precisely that a computer can follow it, then the process will be followed the same way every single time, by every volunteer, on every busy Tuesday.

Throughout this book we will keep coming back to Sparrow Street Library. We will write a late-fee calculator, check member cards, search for books, enforce the borrowing limit, organize shelves, compute reading-challenge statistics, and finally put everything together into a small library system in Chapter 28. Each program is small, but together they show how real software grows out of clear thinking.

What Is a Program?

A **program** is a list of instructions that a computer follows, one after another, to get something done. That is all. A calculator app, a video game, and the software that runs a bank are all, underneath, very long lists of very precise instructions.

The important word is **precise**. When you give instructions to a person, you can be vague because the person fills in the gaps with common sense. If you tell a friend "grab me a coffee on your way," they know to pay for it, to bring it in a cup, and not to bring you a bag of coffee beans. A computer has no common sense at all. It does exactly what it is told, in exactly the order it is told, and nothing more.

A classic classroom exercise shows this nicely. A teacher asks the students to write instructions for making a jam sandwich, then follows the instructions *literally*. "Put jam on the bread" results in the whole jar being placed on top of the loaf, because nobody said to open the jar, pick up a knife, or take out a single slice. The students laugh, but they have learned the central lesson of programming: **the machine only knows what you tell it**.

Instructions, data, and results

Almost every program you will ever write has the same basic shape:



For the late-fee problem, the **input** is the number of days a book was kept. The **processing** is the set of rules on the card. The **output** is the fee to charge. Programming logic lives in the middle box: it is the reasoning that turns inputs into correct outputs.

So What Exactly Is Programming Logic?

Programming logic is the skill of planning the steps a program must take, in the right order, with the right decisions and repetitions, so that it produces the correct result for every possible input. It is the thinking that happens *before* and *behind* the code.

It helps to separate two ideas that beginners often mix up:

Idea	What it means	Example
Logic	What steps to take and why	"If the fee is over $5.00, use $5.00"
Syntax	How a language spells those steps	`if fee > 5.00:`

Syntax is the grammar of a particular programming language: where the colons go, which words are special, how to indent. Syntax matters, and the computer will refuse to run your program if you get it wrong. But syntax is the easy part to fix, because the computer usually tells you exactly where the mistake is.

Logic is harder. A program with perfect syntax can still be completely wrong. If you write the late-fee rule so that the cap is checked *before* the fee is calculated, the program runs happily and charges the wrong amount. The computer cannot warn you, because it has no idea what you *meant*. Only clear thinking prevents that kind of mistake.

> **NOTE**
>
> This book teaches logic first and uses Python as the language for expressing it. Python was chosen because its syntax is short and readable, so it gets out of the way. The thinking skills you build here carry over to Java, C#, JavaScript, and nearly every other language you may learn later.

Computers Are Literal: The Problem of Ambiguity

Let us try to state the late-fee rule in ordinary English and see how many questions a computer would need answered.

"Charge the member a fee if the book is late."

This sentence feels clear to a person. Now read it as a computer would:

- **Late compared to what?** The due date. How is the due date decided? 14 days after the borrowing date.
- **Is a book returned on the due date late?** The rule says the loan period is 14 days, so returning on day 14 is on time. Late starts on day 15.
- **How much is the fee?** $0.25 for each day late.
- **Is there a limit?** Yes, $5.00 per book.
- **What if the member returns three late books?** The cap is per book, so each book is capped separately.
- **What if the book is returned early?** No fee. The number of days late must never be negative.

One short sentence turned into six questions. This is completely normal. A large part of a programmer's job is asking these questions *before* writing code. Experienced programmers are not people who type fast; they are people who notice what is unclear.

> **WARNING**
>
> Boundary values, like "exactly 14 days" or "exactly $5.00," are where most logic mistakes hide. Whenever a rule contains a number, ask yourself: "What happens exactly at that number, one below it, and one above it?" We will return to this habit again and again, especially in Chapter 23 on testing.

Solving Our First Problem: The Late-Fee Rule

Now that the questions are answered, we can write the late-fee process as numbered steps in plain English. We will assume we already know how many days the member kept the book.

1. Get the number of days the book was kept.
2. Subtract the 14-day loan period to find the days late.
3. If the days late is less than zero, set it to zero (the book came back early).
4. Multiply the days late by $0.25 to get the fee.
5. If the fee is greater than $5.00, set the fee to $5.00.
6. Report the fee.

Notice that each step does one small thing, and the order matters. Step 5 must come after step 4, because you cannot cap a fee that has not been calculated yet. Step 3 must come before step 4, or an early return would produce a negative fee, and the library would end up paying the member.

Tracing the steps by hand

A plan is only useful if it gives the right answer. The best way to check a plan without a computer is to **trace** it: pretend to be the computer, follow each step exactly, and write down the values as you go. A **trace table** has one column for each value we care about and one row for each test case.

Days kept	Days late	Fee before cap	Final fee
10	0 (was -4)	$0.00	$0.00
14	0	$0.00	$0.00
15	1	$0.25	$0.25
21	7	$1.75	$1.75
34	20	$5.00	$5.00
50	36	$9.00	$5.00

Look carefully at how the test cases were chosen. We did not just pick random numbers. We picked an early return (10), the exact due date (14), the first late day (15), an ordinary late case (21), the exact point where the cap is reached (34 days kept means 20 days late, and 20 times $0.25 is exactly $5.00),

and a case well past the cap (50). Each row checks a different part of the logic. If every row matches what the rules on the card say, we can be confident in the plan.

> **TRY IT**
>
> Add three more rows to the trace table: 0 days kept, 33 days kept, and 100 days kept. Work out each column by hand. Which of your rows tests the cap, and which tests the "early return" step?

The same logic in Python

Here is the late-fee plan written in Python. You do not need to run it yet; Chapter 4 shows you how to install Python and run programs. For now, read it line by line and compare it with the numbered steps above.

late_fee.py

```
# Sparrow Street Library - late fee for one book
days_kept = 21                  # step 1: the input

days_late = days_kept - 14      # step 2: subtract the loan period
if days_late < 0:               # step 3: returned early?
    days_late = 0

fee = days_late * 0.25          # step 4: $0.25 per day late
if fee > 5.00:                  # step 5: apply the cap
    fee = 5.00

print("Days late:", days_late)  # step 6: report
print("Fee: $", fee)

# Output:
# Days late: 7
# Fee: $ 1.75
```

Even without knowing any Python, you can probably follow this. A few details are worth pointing out now, and each one will get a full explanation later in the book:

- Lines starting with `#` are **comments**. Python ignores them; they are notes for humans.
- `days_kept = 21` creates a **variable**, a named box that holds a value. Chapter 5 covers variables in detail.
- `if days_late < 0:` is a **decision**. The indented line below it runs only when the condition is true. Chapter 7 is all about decisions.
- `print(...)` shows a result on the screen.

The Python version is a direct translation of the plain-English steps. That is the whole approach of this book: **think in steps first, then translate**. When the thinking is right, the translation is usually the easy part.

The Three Building Blocks of Every Program

It may sound surprising, but every program ever written, from the late-fee calculator to a web browser, is built from just three kinds of control structures. Computer scientists showed long ago that these three are enough to express any algorithm. Learning to recognize them is the first real step toward thinking like a programmer.

1. Sequence: do this, then this, then this

Sequence means steps run one after another, in order, from top to bottom. Most of the late-fee program is sequence: subtract, then multiply, then print.

```
# Sequence: each line runs once, in order
title = "The Time Machine"
copies = 3
copies = copies - 1          # one copy is borrowed
print(title, "copies left:", copies)

# Output:
# The Time Machine copies left: 2
```

Order matters in a sequence. If you swapped the third and fourth lines, the program would print 3 instead of 2, because it would report the count before the borrowed copy was subtracted.

2. Selection: choose between paths

Selection means the program makes a decision and takes a different path depending on a condition. The borrowing limit rule is a perfect example: the limit depends on the member's age.

```
# Selection: pick a path based on a condition
age = 10
if age < 12:
    limit = 3
else:
    limit = 5
print("Borrowing limit:", limit)

# Output:
# Borrowing limit: 3
```

Only one of the two paths runs. For Ben, who is 10, the limit is 3. For Ada, who is 34, the limit is 5. The word `else` means "otherwise": if the condition is not true, take this path instead.

3. Repetition: do something many times

Repetition, also called **looping**, means repeating a set of steps. Dara does not calculate one late fee on Tuesday; she calculates one for every returned book. Instead of writing the same steps twenty times, a program writes them once and repeats them.

```
# Repetition: do the same steps for every value in a list
days_kept_list = [10, 21, 50]
for days_kept in days_kept_list:
    days_late = days_kept - 14
    if days_late < 0:
        days_late = 0
    fee = days_late * 0.25
    if fee > 5.00:
        fee = 5.00
    print(days_kept, "days kept -> fee", fee)

# Output:
# 10 days kept -> fee 0.0
# 21 days kept -> fee 1.75
# 50 days kept -> fee 5.0
```

The `for` line says "for each number in this list, do the indented steps." Notice how this example combines all three building blocks: repetition on the outside, selection inside it, and sequence inside that. Real programs are always combinations like this. Chapters 9 and 10 explain loops in depth.

> **NOTE**
>
> Python shows `0.0` and `5.0` rather than `$0.00` and `$5.00`. Formatting money neatly is a presentation detail we will handle later in the book. For now, focus on whether the *value* is correct.

Spotting the building blocks in everyday life

Once you know the three building blocks, you start seeing them everywhere:

Everyday task	Building block
Follow a recipe step by step	Sequence
Take an umbrella if it is raining	Selection
Stir until the sauce thickens	Repetition
Wash each plate in the sink	Repetition
Stop at a red light, go on green	Selection

Notice the two kinds of repetition in the table. "Wash each plate" repeats a known number of times, once per plate. "Stir until the sauce thickens" repeats until a condition changes, and you do not know in advance how many times that will be. Programs use both kinds, and the difference will matter a lot in Chapters 9 and 10.

Why Programming Logic Matters

You might wonder why a book would spend so much time on logic instead of jumping straight into a language. There are several good reasons.

Logic outlives any single language

Programming languages come and go, and even popular ones change from version to version. Logic does not. The late-fee rule is the same whether it is written in Python, Java, or JavaScript. Here is the heart of it in Java:

```
// Java
int daysLate = daysKept - 14;
if (daysLate < 0) {
    daysLate = 0;
}
double fee = daysLate * 0.25;
if (fee > 5.00) {
    fee = 5.00;
}
```

And in JavaScript:

```
// JavaScript
let daysLate = daysKept - 14;
if (daysLate < 0) {
  daysLate = 0;
}
let fee = daysLate * 0.25;
if (fee > 5.00) {
  fee = 5.00;
}
```

The punctuation is different. Java and JavaScript use curly braces and semicolons; Python uses a colon and indentation. But the *steps* are identical. Someone who understands the logic can read all three versions, and can learn a new language in weeks rather than months, because they only need to learn new spelling, not new thinking.

Logic makes you faster, not slower

Beginners often feel that planning is a waste of time and that "real programmers" start typing immediately. In practice, the opposite is true. A few minutes spent writing clear steps and tracing them saves hours of confused debugging later. When you know exactly what each part of your program should do, you can find mistakes quickly, because you can compare what the program *does* against what it *should* do.

Logic is useful even if you never become a programmer

Breaking a problem into steps, spotting unclear rules, checking boundary cases, and testing your reasoning are valuable in almost any job. A volunteer writing a clear procedure for the library's front desk is using the same skills as a software developer. So is a student planning a science experiment or a manager designing a process for handling customer complaints.

Logic helps you work with tools, including AI tools

Many people now use AI assistants, such as a `<chat-model>`, to suggest code. These tools can be helpful, but they can also produce code that looks right and is subtly wrong, for example a late-fee function that forgets the cap. If you cannot read the code and trace it, you cannot tell the difference. Strong logic skills let you use any tool with confidence, because you can check its work.

A Four-Step Problem-Solving Process

Mathematicians and teachers have long recommended a simple four-step approach to solving problems. It fits programming very well, and we will use it throughout the book:

1. **Understand the problem.** What are the inputs? What is the output? What are the rules? What is unclear?
2. **Make a plan.** Write the steps in plain language, pseudocode, or a flowchart (Chapter 3).
3. **Carry out the plan.** Translate the steps into code.
4. **Look back and check.** Trace the plan and test the code with carefully chosen cases, including boundaries.

Let us apply it to a new library problem: *"Tell a member whether they can borrow another book."*

Step 1, understand. The inputs are the member's age and how many books they currently have out. The output is "yes" or "no." The rule is: under 12 may have at most 3 books; everyone else at most 5. One question to settle: "at most 3" means a child with 3 books out cannot borrow a fourth, but a child with 2 books out can borrow one more.

Step 2, plan.

1. Get the member's age and the number of books they currently have.
2. If the age is less than 12, the limit is 3; otherwise the limit is 5.
3. If the number of books currently out is less than the limit, answer "yes."
4. Otherwise answer "no."

Step 3, carry out.

can_borrow.py

```
age = 10
books_out = 3
```

```
if age < 12:
    limit = 3
else:
    limit = 5

if books_out < limit:
    print("Yes, the member can borrow another book.")
else:
    print("No, the member has reached the limit of", limit)

# Output:
# No, the member has reached the limit of 3
```

Step 4, look back. We trace a handful of cases, choosing boundaries on purpose:

Member	Age	Books out	Limit	Answer
Ben	10	2	3	Yes
Ben	10	3	3	No
Chen	12	3	5	Yes
Ada	34	4	5	Yes
Ada	34	5	5	No

The row for Chen is especially important. Chen is exactly 12. The rule says "under 12," so 12 is *not* under 12, and Chen gets the adult limit of 5. If we had accidentally written `age <= 12`, Chen would wrongly be limited to 3 books. This is the boundary habit from earlier paying off.

> **TRY IT**
>
> Suppose the library decides that members aged 65 and over may borrow 8 books. Before reading ahead, write the new plan in plain-English steps. How many different limits are there now, and which ages are the boundaries you would test?

Common Beginner Mistakes in Logic

Every programmer makes logic mistakes, and making them is part of learning. Knowing the most common ones helps you catch them sooner.

Jumping straight to code

It is tempting to open an editor and start typing as soon as you read a problem. Without a plan, you end up solving the problem and learning the language at the same time, which is twice as hard. Write the steps first, even if it is just four lines on a scrap of paper.

Leaving rules ambiguous

If you cannot answer "what happens exactly at 14 days?" before coding, the program will answer it for you, and probably not the way the library wanted. When a rule is unclear, ask the person who owns the rule. In a real job that might be a customer or a manager; for Sparrow Street Library it is the volunteers' rule card.

Getting the order wrong

Capping the fee before calculating it, printing a result before updating it, or subtracting the loan period after multiplying are all ordering mistakes. The program runs, but the answer is wrong. Tracing catches these quickly.

Off-by-one errors at boundaries

Using "less than" when you meant "less than or equal to" is so common that it has its own name: an **off-by-one error**. The Chen example above is exactly this kind of mistake. Always test the boundary value itself, plus one on each side.

Testing only the "happy path"

Checking that a book returned 7 days late gets a $1.75 fee is good, but it only tests the most ordinary case. Early returns, returns on the exact due date, and very late returns are where things break. A good set of test cases deliberately includes the unusual ones.

Assuming the computer "knows what you mean"

If you forget to write a step, the computer will not fill it in. Reading your plan as if you were a very literal machine, like the jam-sandwich teacher, is one of the best habits you can develop.

How to Get the Most from This Book

This is a hands-on book. Each chapter explains an idea, traces it on paper, shows it in Python, and then gives you practice exercises with difficulty labels. Selected exercises have worked solutions at the end of the chapter. To learn well:

- **Do the exercises before reading the solutions.** Struggling a little is how the ideas stick.
- **Trace on paper.** A pencil and a trace table will teach you more than rereading a page.
- **Type the code yourself** once you have Python set up in Chapter 4. Copying by hand forces you to notice every detail.
- **Change things and predict.** Before running a modified program, predict its output. Then run it and compare.

In the next chapter we will look at how to break big problems into small ones, how to spot patterns, and how to ignore details that do not matter, which are the core thinking tools behind every plan.

Practice Exercises

You do not need a computer for these exercises. Pencil and paper are enough. Where a Python version is shown in a solution, you will be able to run it after Chapter 4.

1. **[Easy]** For the task "count how many books a member currently has overdue," name the input, the processing, and the output.
2. **[Easy]** Label each activity as sequence, selection, or repetition: (a) brushing each tooth; (b) wearing a coat if the temperature is below 10 degrees; (c) unlocking the door, then opening it, then walking in; (d) knocking until someone answers.
3. **[Easy]** Use the late-fee steps from this chapter to trace the fee for books kept 16, 30, and 40 days. Show days late, fee before cap, and final fee.
4. **[Easy]** The library wants to "send a reminder when a book is almost due." Write at least four questions a programmer would need answered before writing this program.
5. **[Medium]** Write plain-English steps that convert a temperature from Celsius to Fahrenheit using the formula F = C × 9 / 5 + 32. Trace your steps for 0, 100, and -40 degrees Celsius.
6. **[Medium]** Ben wrote these steps: (1) get days kept; (2) if the fee is greater than $5.00, set it to $5.00; (3) subtract 14 to get days late; (4) multiply days late by $0.25 to get the fee; (5) report the fee. Find every problem with the plan and write a corrected version.
7. **[Medium]** Extend the borrowing rule so that members aged 65 and over may borrow 8 books. Write the steps and a trace table with at least six rows that includes every boundary age.
8. **[Medium]** Ada returns three books kept 20, 40, and 15 days. Remembering that the cap is per book, work out each fee and the total she owes. Then write plain steps for computing the total for any number of books.
9. **[Challenge]** FizzBuzz: for each number from 1 to 15, say "Fizz" if it is divisible by 3, "Buzz" if it is divisible by 5, "FizzBuzz" if it is divisible by both, and the number itself otherwise. Write the steps, and explain why the order of the checks matters.
10. **[Challenge]** A book was borrowed on day 40 of the year and returned on day 61. Write steps that work out the days kept from these two day numbers and then the fee. What unclear question arises if a book is borrowed in December and returned in January?

Selected Solutions

Exercise 3

Days kept	Days late	Fee before cap	Final fee
16	2	$0.50	$0.50
30	16	$4.00	$4.00
40	26	$6.50	$5.00