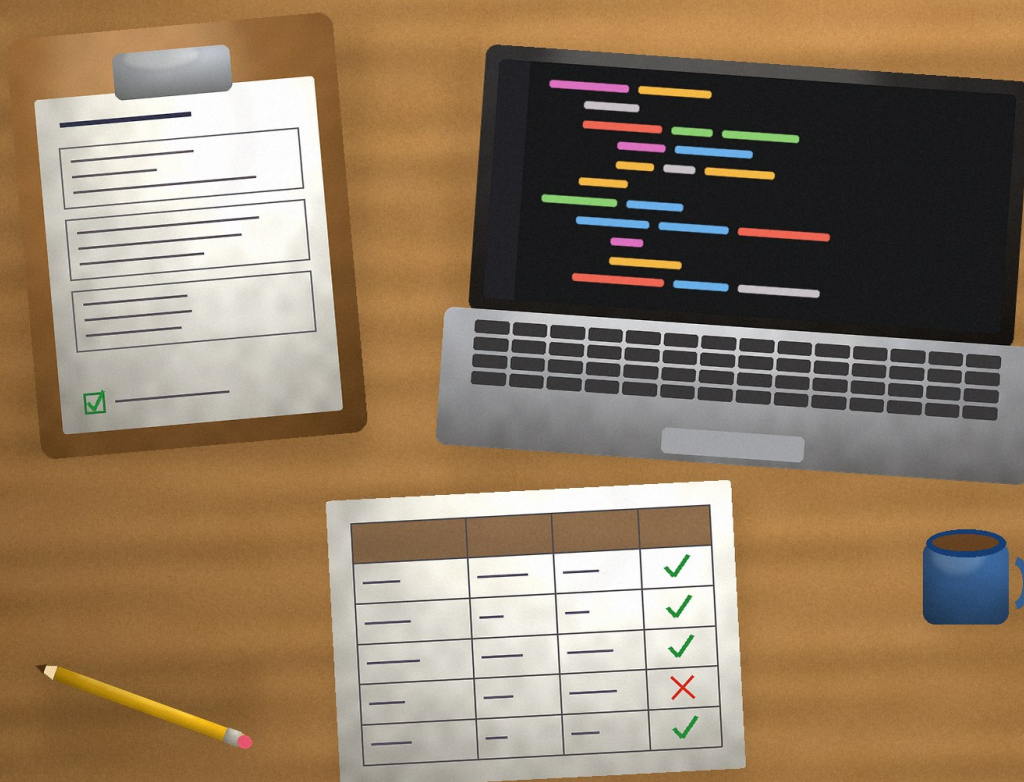


# COMPUTING REFRESH FOR ADULTS

Michael Gannotti



# Computing Refresh for Adults

Michael Gannotti

*A guide for adults learning computing for themselves and for parents refreshing to teach or co-learn with their kids: specs, Python, testing, data and SQL, the web, the command line and Git, home security, AI, and honest records. Book 4 of the SMF WisdomForge Computing series. Illustrated edition, 13 October 2026. Notes run in one series at the back; the superscript numbers link to them. The images are original still lifes, with no people and no brands.*

# Contents

- Welcome
- How to Use This Book This Week
- If You Only Remember Five Things
- The Computing Hour
- Chapter 1 — Coming Back to Computing: Specs, Algorithms, and a Plan You'll Keep
- Chapter 2 — Python as a Working Tool
- Chapter 3 — Testing and Debugging: Proof Before Pride
- Chapter 4 — Data: Spreadsheets, Questions, and SQL
- Chapter 5 — Building for People: The Web
- Chapter 6 — The Command Line and Git: Save Points and Grown-Up Tools
- Chapter 7 — Systems and Security at Home: Backups, Passwords, and the Family Rule
- Chapter 8 — Working With AI as an Adult Learner: Where It's Wrong and Whose Work It Is
- Chapter 9 — Records: Your Learning Log and Portfolio—and Your Teen's Transcript
- Chapter 10 — Resources
- A Note on Sources
- Notes

# Welcome

*An illustration, not a reported family:* Picture a parent at the kitchen table after the kids are asleep, a notebook open, trying to guess what four lines of Python will print. The guess is written in pencil before the laptop is opened. When the screen disagrees, the parent doesn't sigh; they lean in, because now there's a clue.

That scene is the whole book in small. Not the late hour—you'll find your own time—but the order of things. A guess first. Then the computer. Then a calm look at the difference.

## Who this book is for

This book is for two kinds of readers, and some of you are both.

The first is a parent who wants to teach computing at home, or learn it alongside a child, and who would like to feel steady doing it. Maybe your seven-year-old has discovered block coding and keeps asking what the blocks “really” do. Maybe your thirteen-year-old has moved to typed Python and you'd like to read over a shoulder without bluffing. Maybe your sixteen-year-old wants a transcript line that means something, and you're the one who has to sign it.

The second is an adult learning for yourself. You might be curious about a career change, or retired with time at last, or simply tired of not knowing what a program does when it runs. You may never have written a line of code. You may have taken a programming class long ago and kept nothing but the memory of a semicolon.

Both of you are welcome here, and neither of you is behind. You'll read the same chapters. Each teaching chapter teaches you first, as an adult learner, and then offers a short section called “Coaching your kids at every age” that turns the same skill into something you can do with a child of five, a twelve-year-old, or a teen. If there are no children in your life right now, that section is optional.

## Rust is ordinary

Many adults feel rusty. That's the normal starting point, not a verdict.

Here is what the evidence on learning to program actually suggests: the difference between beginners who make progress and beginners who stall is mostly a difference in strategy—planning before typing, tracing what the code will do, testing it, and changing one thing at a time—rather than some

rare gift.<sup>1</sup> Strategies can be taught. They can be practiced in twenty minutes on a Tuesday. This book is built from them.

## The line the whole book runs on

**Specify it small; predict before you run; debug before you rewrite; explain it before you call it yours.**

- **Specify it small.** Say what you want before you build it, in writing, in a size you can finish this week.
- **Predict before you run.** Write your guess before the computer answers. A wrong guess is the most useful thing that can happen in a session; it shows you exactly where your picture and the program part ways.
- **Debug before you rewrite.** A bug is a clue. Starting over throws the clue away.
- **Explain it before you call it yours.** If you can't explain every line, it isn't yours yet—whoever or whatever wrote it.

The last three clauses are the same ones the earlier books in this series teach to children and teens, word for word. That's deliberate. When your child hears “predict before you run,” you'll know it from the inside.

## What a good session looks like

A good session is short and complete. You write a few lines saying what you'll build today and how you'll know it's done. You write down what you expect to happen. You build a little and test it. When something surprises you, you write down what you expected, what happened, and the one change you'll try. Then you say what you did, name any help you used, and save your work in two places.

That's **The Computing Hour**, and it has its own section right after the front matter. The name is the series' word; your sessions will often run fifteen or twenty-five minutes, and that counts.

## What this book won't do

This book won't hand you a 180-day script. It gives you this week's sessions and a menu of activities, and trusts you to choose.

It won't sell you a course, and it doesn't rank products. Tools are named plainly when they fit, with a reminder to check the current version.

It won't promise you a job. Chapter 9 talks honestly about work, as a possibility with real costs in time, not as a guarantee.

It won't print you a certificate. Your records will be real: a dated learning log, a portfolio, and an honest note on every project about what help you used.

And it won't promise that programming will make you or your children cleverer in general. Learning to program teaches programming, which is plenty. Other benefits are possibilities, and we'll treat them that way.

## Where this book sits

This is the fourth and final book in the SMF WisdomForge Computing series. *Computing for Little Thinkers* (ages 5–10) asks the parent to be a literal robot and a patient question-asker: “What will I do if you tell me that?” *Computing for Young Minds* (ages 11–14) asks the parent to check predictions and read the Debug Log. *Computing for Emerging Adults* (ages 15–18) asks the parent to be first user, safety-keeper, and records-keeper while the teen runs the work. This book is the one for you. It teaches the same habits at an adult's pace, so that whichever rung your child is on, you can stand beside them—and so that, if you're learning for yourself, you have a complete path of your own.

You don't need to be ahead of your child at any rung. You need to ask good questions, wait, and say “I don't know yet—let's find out.” This book will help you do all three.

Turn the page, read How to Use This Book This Week, and plan three short sessions. By Sunday you'll have a dated baseline page, a reason written on a card, and your first prediction checked against a real program.

— Michael Gannotti

# How to Use This Book This Week

You can start this book tonight with a pencil. Nothing in the first week needs more than a notebook, a deck of index cards, and—on the third day—a computer you’re allowed to install a program on.

## Two roads through the book

**If you’re learning for yourself**, read Chapters 1 through 8 in order. Each one builds on the last: specs and algorithms, then Python, then testing, data, the web, the command line and Git, household security, and working with AI. Chapter 9 shows you how to keep records as you go, and Chapter 10 helps you find tools and people. If you’re learning for yourself, the coaching sections are optional—some readers use them with a nephew, a grandchild, or a friend.

**If you’re refreshing to teach or co-learn**, take the same road in the same order, and read each chapter’s “Coaching your kids at every age” section too. Then keep the sibling book for your child’s rung nearby: *Computing for Little Thinkers* for ages 5–10, *Computing for Young Minds* for 11–14, *Computing for Emerging Adults* for 15–18. You don’t need to finish this book before your child starts theirs. Most parents run a few weeks ahead on the habits—predicting, logging, explaining—and learn the content alongside.

## Place yourself by what you can do today

Start with START-01 The Baseline Page in Chapter 1. It asks you to write, from memory and with no help, what you can already read, trace, and build. Date it. That page—not a class you took years ago, and not what you think you ought to know—tells you where to begin. If your baseline shows you can already write a small Python function and test it, you can move quickly through the early parts of Chapter 2. If it shows a blank page, you’re in exactly the right place to start at Chapter 1’s first try-it.

Each chapter ends with “What ‘done enough’ looks like,” a short checklist. Done enough means ready to move on, not finished forever.

## This week

Three sessions and one optional extra. Each uses cards from the activity bank; the codes and names match the bank exactly, so you can find them again.

**Day 1 (desk, about 30 minutes).** START-01 The Baseline Page, then START-02 Why Card: one sentence each on why you're learning, for whom, and what you'll build first. Tape the card inside the front cover of your notebook.

**Day 2 (micro, about 15 minutes).** START-03 Cue and Kit: choose a fixed cue (after the morning coffee, after school drop-off, after the dishes) and set out a notebook, a pencil, and one saved file. Then PLAN-01 The Week on One Card: next week's two or three sessions, each with a day, a length, and the one thing you'll build.

**Day 3 (desk, about 40 minutes).** START-04 Set Up a Clean Workspace: install a beginner-friendly Python editor, make one learning folder, and save a first file in two places. Then PY-01 Predict the Output—write what a short program will print, run it, and compare. Finish with LOG-01 The Learning Log Page: the date, the length, what you specified, predicted, and built, and the help you used.

**Weekend (optional micro).** SPEC-03 The Literal Reader with a partner or your child: they follow your written steps for making a sandwich with total literalness, and you repair every ambiguity. It's the funniest fifteen minutes in the book, and it teaches the first chapter's biggest idea.

**Parents:** add one coaching move from the chapter your child is working in. COACH-02 Ask for the Prediction is always safe: before anything runs, ask your child to write what they think it will do.

## The activity bank is a menu

The book's 100 activities are coded cards—START, SPEC, PY, TEST, and so on—and every chapter points to the ones that belong to it. Each card names its context, its time, and which rung of the ladder it helps you support: yourself, all ages, or a specific age band. You won't do all of them. Pick the ones that fit the week you actually have.

The contexts are the same everywhere in the book:

- **Micro** (15–25 minutes): a lunch break, a wait at practice, a ride where someone else is driving. Paper cards fit here.
- **Desk** (30–50 minutes): an evening at the table.
- **Deep** (60–90 minutes): a weekend block, for projects.

Two or three sessions a week make a full week. On a hard week, one micro session on paper keeps the habit alive.

## How long a pass takes

This is a planning estimate, not a research finding. At two or three sessions a week, most adults can expect roughly three weeks for Chapter 1, four or five for Chapter 2, three for Chapter 3, four for Chapter 4, three or four for Chapter 5, two or three each for Chapters 6 and 7, and about three for Chapter 8. That adds up to roughly six to nine months for a first full pass, with the record-keeping practices of Chapter 9 running all along. Faster or slower is fine.

## Learning alongside your kids

COACH-06 Co-Learning Hour is a deep session you run side by side with a child or teen on the same small spec, each with your own file and your own log. It's the best single thing a parent can do with this book, and it works even when your teen is ahead of you: you become the spec reader and the first user, and they get the strongest possible check of their own understanding—teaching it to you.

For the whole household, FAM-05 Screens on Purpose is a short family media plan: screens open for a job and close when it's done, and sleep and meals are protected. Your learning sessions follow the same rule.

## Special situations

- **Shift work or caregiving.** Micro sessions on paper count. In a hard season, the log *is* the week.
- **Low vision or other access needs.** Larger editor fonts, a screen reader, and keyboard navigation all work with the tools in this book. Chapter 5's accessibility pass applies to your own setup as much as to a web page.
- **One shared family computer.** Give each person a separate user account, and keep a teen's portfolio on a family drive. **The Computing Hour's** AI box has one more caution for shared computers.
- **A teen who's ahead of you.** Wonderful. Ask them to teach you one thing a week.
- **Learning through work.** Follow your employer's rules on tools and data, especially before using AI tools with work code.

Now turn to If You Only Remember Five Things. It's short on purpose.

# If You Only Remember Five Things

If a busy month swallows the rest of this book, keep these five. They're the habits underneath every chapter, and they're the same ones you'll ask of your kids.

## **1. Say what you want before you build it—small enough to finish.**

A few written lines—what it's for, what goes in, what comes out, and what “done” means—will save you more time than any shortcut you'll ever learn. Small is the adult's secret: a program you finish this week teaches more than a grand one you abandon in March. If the spec won't fit on one page, the project is too big for now; cut it until it does.

## **2. Guess what will happen before you run it, and write the guess down.**

The guess turns a run into an experiment. When the computer agrees, you've confirmed your picture; when it disagrees, you've found the exact spot where your understanding needs work, which is the most valuable thing a session can produce. A prediction can be one line in pencil. It still counts.

## **3. When it breaks, change one thing and log it.**

Expected, actual, your guess, one change, the result. Five lines on a page will calm a frustrating evening and, a month later, show you how much you've learned. Three changes at once only tells you that something, somewhere, did something. And when you're tempted to delete it all and start fresh, log the bug first; the clue is worth more than the clean page.

## **4. Call it yours only when you can explain every line; name the help you used.**

A tutorial, a friend, the official documentation, an AI tool—all of it is fine to use, and none of it is shameful. Write “Help I used: ...” every time, even when the answer is “none,” and keep only

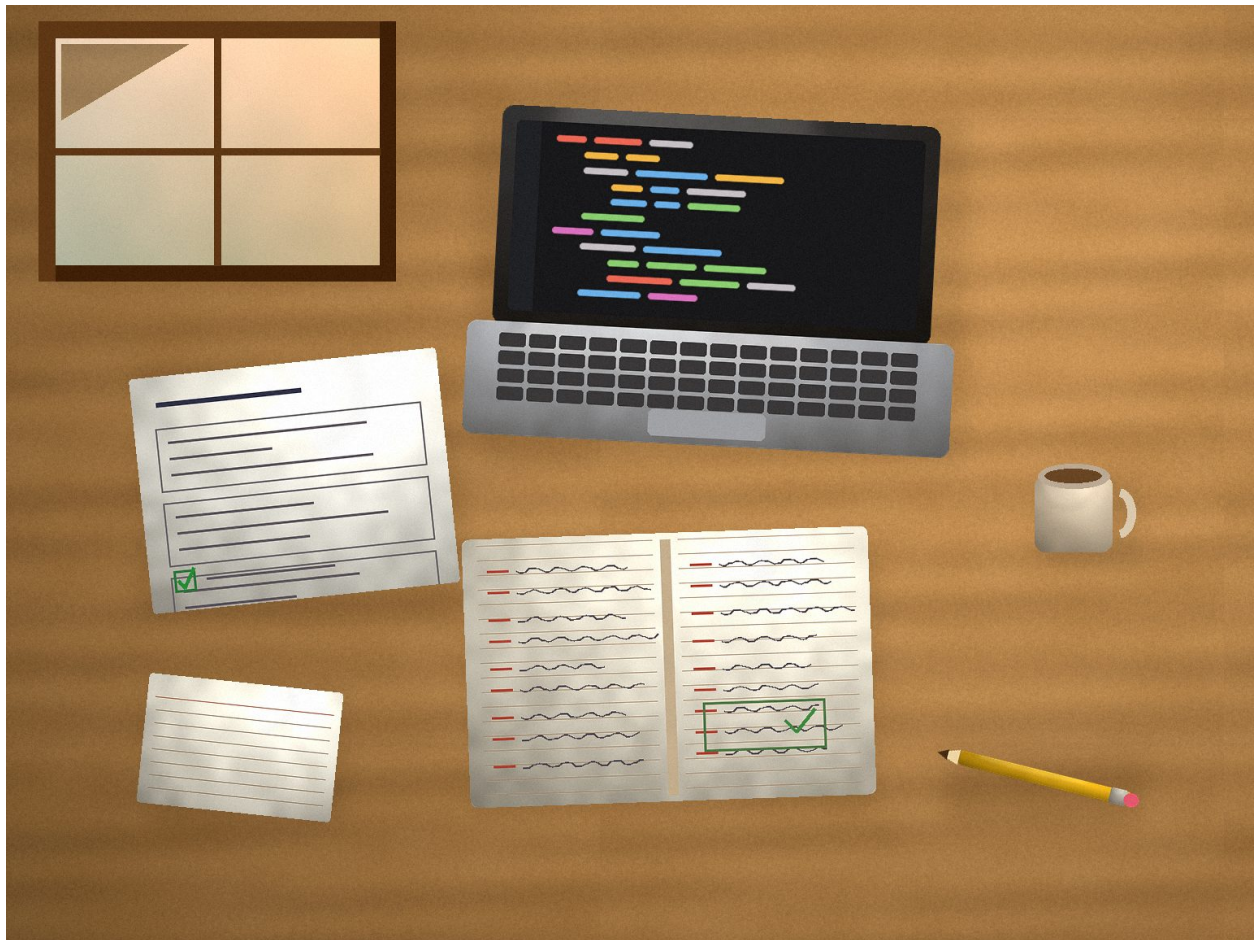
the code you can explain line by line. That rule protects your learning, and it's the same rule you'll hold your teen to—so it helps to have kept it yourself.

## **5. Your kids need your questions more than your answers.**

“What do you think it'll do?” “Show me where it surprised you.” “I don't know either—let's find out.” You don't have to be ahead of your child at any age. You have to ask, wait, and hand the work back.

Everything else in this book—Python, data, the web, Git, security, AI—is these five habits applied to a new kind of problem. When a chapter feels heavy, come back to this page and ask which of the five you skipped.

# The Computing Hour



**The Computing Hour** is the session this whole book runs on. It has five steps, the same five every time, and you run all of them yourself. “Hour” is the series’ word, not a clock promise: your sessions will often be fifteen or twenty-five minutes, sometimes fifty, now and then ninety. Learn the shape this week, and every chapter after this will slot into it.

**Specify it small; predict before you run; debug before you rewrite; explain it before you call it yours.**

## The five steps

| Step                    | What you do                                              | Micro (15–25) | Desk (30–50) | Deep (60–90) |
|-------------------------|----------------------------------------------------------|---------------|--------------|--------------|
| <b>Specify</b>          | Write the goal, the input, the output, and the done line | 2–3 min       | 5 min        | 5–10 min     |
| <b>Predict</b>          | Write what you expect to happen before running           | 1–2 min       | 2–3 min      | 3–5 min      |
| <b>Build and Test</b>   | Write a little, run it, check it against your examples   | 8–15 min      | 15–30 min    | 35–60 min    |
| <b>Debug and Log</b>    | Expected / Actual / Guess / One change / Result          | 2–3 min       | 5–8 min      | 8–12 min     |
| <b>Explain and File</b> | Say or write the exit line; save in two places           | 1–2 min       | 3–4 min      | 4–5 min      |

**Specify.** A spec is a short written description of what a program must do, with examples and a line saying what “done” means. For a micro session it can be three lines on an index card: “Today I’m building a tip splitter for our dinner group. It takes the bill and the number of people and gives back each share. It’s done when 60 split three ways gives 20.” Later in a project, Specify is a two-minute reread and one updated line. Chapter 1 teaches specs in full.

**Predict.** Before you run anything, write one line: “When I run it with , *I expect* .” This feels slow for about a week. Then it starts to feel like the point. A wrong prediction isn’t a failure; it’s a map of the exact place where your picture of the program and the program itself disagree.

**Build and Test.** Write a few lines, run them, and check the result against the examples in your spec. Then a few more. Small steps keep surprises small. From Chapter 3 on, “test” means a written check—code that compares what the program does with what it should do—but on day one, a test can be you looking at the output and your spec side by side.

**Debug and Log.** When something surprises you, reach for the pencil before the keyboard. Write what you expected, what actually happened, your best guess at why, the one change you’ll try, and what happened when you tried it. That’s a Debug Log entry, and it gets its own section below.

**Explain and File.** End by saying or writing the exit line, then save your work in two places—your learning folder and a backup. Filing is part of the step, not an afterthought; Chapter 9 turns these files into a portfolio without any extra work.

## The exit line

Every session ends with the same five-part sentence, said aloud or written in your log:

“I specified... I predicted... It did... Next I’d change... Help I used: ...”

For example: “I specified a tip splitter for three people. I predicted 60 would give 20. It did, but 50 gave 16.666666666666668. Next I’d change it to round to cents. Help I used: none.”

“Help I used” gets filled in every time, even when the answer is “none.” A tutorial, a friend, the official documentation, an AI tool, your teenager—all of it counts, and none of it is shameful. The habit matters because you’ll ask the same of your kids, and because six months from now you’ll want to know what you could do alone.

## The Debug Log

The Debug Log is a notebook (or a file) with one page per bug, dated. Each page has five fields, always in this order:

- **Expected** — what you thought would happen
- **Actual** — what did happen, copied exactly
- **Guess** — your best reason for the difference
- **One change** — the single thing you’ll try
- **Result** — what happened after that change

Here’s a filled page from a micro session (an illustration):

**Tue, 7:10 p.m. (micro)** **Expected:** 20 for each of 3 people from a 60 bill. **Actual:** 20.0 **Guess:** dividing always gives a decimal number in Python. **One change:** none needed—20.0 equals 20. Updated my spec to say “20.0 is fine.” **Result:** spec now matches; test passes.

Notice that the change was to the spec, not the code. That happens more than you’d think, and it’s a good sign: it means you’re checking your intentions, not just your typing. BUG-02 The Debug Log Page is the bank card for this habit; Chapter 3 builds it into a full testing practice.

## Contexts, not ages

Your sessions come in three sizes, chosen by the time you have, not by how advanced you are.

- **Micro (15–25 minutes)** fits a lunch break, a wait at swim practice, or a train ride. Paper work fits here beautifully: a trace, a test table, a spec, a log review, PY-01 Predict the Output.
- **Desk (30–50 minutes)** fits an evening at the table. Most worked sessions in this book are desk sessions: one new idea, practiced until it works.
- **Deep (60–90 minutes)** fits a weekend morning. Use it for projects, and take a short break in the middle—stand up, refill the mug, look at something far away.

Two or three sessions a week make a full week. As a planning estimate, that's about two to three hours a week for most adults. Less is still progress. PLAN-01 The Week on One Card is the two-minute way to plan them: day, length, and the one thing you'll build.

## The rescue order

Everyone gets stuck. The rescue order is what you do next, in order, so that being stuck turns into learning rather than flailing.

**For yourself:** 1. **Reread your spec.** Half of all stuck moments are a fuzzy spec. Is the done line actually written? 2. **Wait.** Set a five-minute timer and trace the code by hand, line by line, on paper. Most bugs surrender here. 3. **Hint.** Check your own notes and the official documentation. 4. **Worked example.** Build a short example of the idea in a separate scratch file—a few lines, nothing else—until it does what you expect. 5. **Back to your own file.** Bring what you learned home.

Ask a person or a tool after the timer, not before, and log that you did.

**With a child** (the rule the whole series uses): 1. **Ask:** “What did you think it would do?” 2. **Wait.** Really wait. Count to ten in your head if it helps. 3. **Hint:** “Look at line three again.” 4. **Short demo on your own copy**—never on theirs. 5. **Hand it back.**

COACH-03 The Rescue Order is the bank card for rehearsing the child version. It's harder than it looks, mostly because the urge to grab the mouse is strong.

## When the Hour goes wrong

- **No time.** Do a micro session on paper: one trace, one prediction, one log line. It counts.
- **Stuck the whole session.** Log it and stop. Write Expected, Actual, and your best Guess, and leave One change blank for next time. A logged stuck session is progress; you'll start the next one exactly where the problem is.
- **Too tired.** Do AS-01 Cold Trace only—trace a short program and predict its output—or simply reread last session's log.
- **The session turned into tutorial-following.** You typed what a video typed and it worked, but you can't say why. Next time, change one thing in the example and predict what will happen before you run it.
- **You missed a week.** Pick up at your cue. One missed session doesn't undo a habit; research on habit formation found that a single missed opportunity didn't materially set people back.<sup>2</sup>

## A session, start to finish

*An illustration, not a reported family:* A father has twenty minutes at the kitchen counter while pasta water heats.

**Specify (2 minutes).** On an index card: “Count how many items on a grocery list file. Input: a file with one item per line. Output: a number. Done when my five-line test file gives 5.”

**Predict (1 minute).** “I expect 5.”

**Build and Test (12 minutes).** He writes four lines that open the file and count lines. It prints 6.

**Debug and Log (3 minutes).** Expected 5. Actual 6. Guess: there’s an empty line at the end of the file. One change: open the file and look. Result: there is a blank last line. He updates the spec: “Blank lines don’t count.” Next session, he’ll skip them.

**Explain and File (2 minutes).** “I specified a list counter. I predicted 5. It did 6 because of a blank line. Next I’d change it to skip blank lines. Help I used: none.” He saves to his learning folder and his backup drive, and the water boils.

## The try-it shape

Every activity in this book appears in the same box, so you can read one at a glance:

### Out-of-home — OUT-02 Library Computer Session

- **Time:** 45 minutes (desk)
- **Materials:** your notebook; a USB drive or your own cloud storage; your library card
- **Safety and privacy:** your own accounts only; save to your own drive; log out of everything and close the browser before you leave
- **The fun:** a change of scene, and the quiet of a library table
- **The skill:** running a full session on a public computer, and leaving it clean
- **The crumb:** a file saved in two places

**How it goes.** Library computers vary. Some let you install software; many let you use only a browser. Check what your library allows. Where you can’t install anything, a website that runs short Python programs in the browser, such as Python Tutor, usually still works.<sup>3</sup> Then run one ordinary Hour there—spec, prediction, a little building, a log line. Save the file to your own drive and to one other place. Before you stand up, sign out of every account, close the browser, and check that nothing is left on the desktop. Write in your log that you did. It’s a small rehearsal of a habit you’ll teach your teen in Chapter 7: on a shared machine, you leave no trace.

The fields are always in this order: **Time, Materials, Safety and privacy, The fun, The skill, The crumb**, then **How it goes**. The crumb is the small thing you keep—a page, a file, a card—so your records build themselves.

## The talk-to-yourself shape

Each teaching chapter has one box for coaching yourself through a stuck moment. The questions always come in the same order. Here’s the Hour’s version:

## Talk to yourself (the Computing Hour)

**Opening:** “Before I change anything, ask anyone, or ask a tool: what did I expect, what actually happened, and what’s the one change I’ll try?”

**Follow-ups:** 1. **Prediction** — What did I write down before I ran it? 2. **Evidence** — What exactly did the screen or the trace show? 3. **Location** — Where is the first place reality and my expectation part ways? 4. **Change** — What’s the one change I’ll make, and what do I expect it to do? 5. **Revision** — Did my spec or my test need to change instead?

**How to wait:** set a five-minute timer; trace by hand; read your spec aloud; stand up and come back.

**What a stuck silence usually means:** the spec is fuzzy; you’re tired; or the bug is in a line you’re sure is fine—go there first.

**Don’t say / say instead:** Don’t say “I’m just bad at this.” Say instead “I haven’t found it yet; here’s what I know so far.”

## Household safety basics

Chapter 7 covers security at home in detail. Three basics belong here, because they apply from the very first session.

- **Your accounts, your rules.** Use your own accounts for your own learning, and give each family member their own login on any shared computer.
- **Practice only on your own systems.** Everything in this book that pokes at a network, a password, or a device is for systems you own, or have written permission to test. Never a workplace’s, a school’s, a library’s, or a neighbor’s.
- **Keep family photos out.** Practice files use made-up or public data, and family photos stay out of AI tools and off shared computers.

## Screens on purpose

Open the laptop for a job and close it when the job is done. Protect sleep and meals—yours and your household’s. FAM-05 Screens on Purpose turns this into a short family plan; if you like a starting template, the American Academy of Pediatrics offers a Family Media Plan tool.<sup>4</sup> A good session often ends with the screen closed and the notebook still open.

## The AI box

### AI and your own learning

You hold your own accounts, so the rules here are the ones you choose—and the ones you’ll ask your kids to keep. Try first. Write your prediction and your first attempt

before you ask any tool (AI-01 Attempt, Then Ask). Then a tool can explain an error message, suggest a test you missed, or walk through a line you don't understand.

What a tool makes is made by a tool. A suggested function becomes part of your work only after you can explain every line and it passes your tests. Write "Help I used: ..." on the log every time, and keep an AI-04 Whose Work Is It? Card on every project you file.

Tools are often fluent and wrong: they invent options and library names, use outdated ways of doing things, and cite sources that don't say what they claim. Run it, test it, and open the source.

A careful study of high-school math students found that open access to a chatbot raised practice scores but lowered later unaided scores, while a version built to give hints protected learning.<sup>5</sup> That was teenagers and math, not adults and code—but the lesson about attempting first travels well.

Keep private messages, family names, addresses, and photos of your children and family out of AI tools. At work, check your employer's rules before pasting any work code or data into a consumer tool.

If you use Grok Bot, remember that all of your bots share one computer. Keep photos of your children and family off it, and keep your student's portfolio on a family drive.

Terms change. Check the current age and privacy terms of any tool before your kids use it (Chapter 8).

Every chapter's "Tools, including AI" section points back here and adds only what's different for that chapter.

## How much is enough

The session lengths, the weekly budget, and the six-to-nine-month estimate in How to Use are planning estimates, not research findings. What the research does support is the shape: practice that asks you to retrieve and predict, spread across days rather than crammed into one long sitting, tends to stick better for most people than rereading.<sup>6</sup> That's why a twenty-minute Tuesday and a twenty-minute Thursday are worth more than they look.

## What to say, step by step

Keep these on a card until they're automatic.

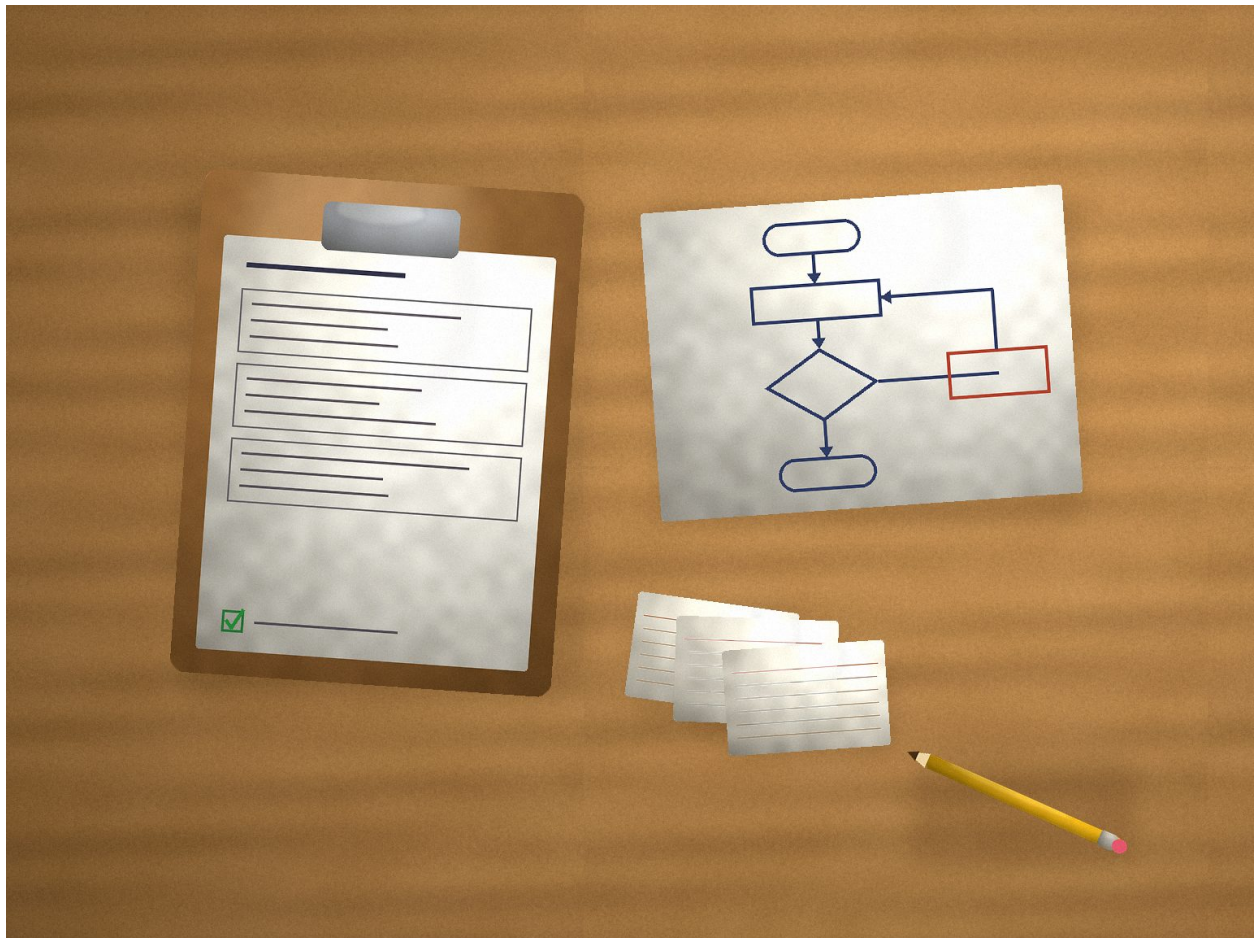
- **Specify:** "Today I'm building \_\_\_\_ for . *It takes* and gives back . *It's done when* ."
- **Predict:** "When I run it with , *I expect* ."
- **Debug and Log:** "Expected . *Actual* . My guess . *One change* . Result \_\_\_\_."
- **Explain and File:** "I specified... I predicted... It did... Next I'd change... Help I used: ..."

- **With a child:** “What do you think it’ll do?” → wait → “Show me on your page.” → “What’s one thing you could change?”

Finish every session with LOG-01 The Learning Log Page: date, length, what you specified, predicted, and built, and the help you used. Five lines. Then close the laptop.

# Chapter 1

## Coming Back to Computing: Specs, Algorithms, and a Plan You'll Keep



### Why this matters

There's a popular story about programming. It says programming belongs to the young and the gifted, that you either "get it" quickly or you never will, and that the people who are good at it

were somehow born that way. It's a common story, and it's easy to believe if your last memory of a programming class is a screen full of errors and a classmate who seemed to understand everything.

The better picture is plainer and much more encouraging. Researchers who reviewed decades of work on how beginners learn to program found that what separates the novices who make progress from the ones who stall is mostly *strategy*: planning before typing, tracing what the code will do, testing it, and fixing one thing at a time.<sup>1</sup> Those are habits, and habits can be learned at any age. **The Computing Hour** you just read is nothing more than those strategies, put in order.

Adults learn this in every decade of life. In one large survey of more than five hundred people aged 60 to 85 who were learning to program, the most common reason given was making up for chances they'd missed when young; the most common frustrations were finding the time and having nobody to learn with.<sup>7</sup> (The respondents were a self-selected and highly educated group, so treat this as a picture of some learners, not all.) Missed chances, not enough time, no one to learn with: if any of that sounds familiar, this chapter is built for it.

Here's the other thing the popular story gets wrong. It treats typing code as the hard part. In practice, the hard part is saying exactly what you want. A computer does what you tell it, with no common sense and no mercy, and most programs that "don't work" are doing precisely what they were told. So this chapter starts on paper, with the skill that makes every later chapter easier: writing a short, clear spec, and turning it into exact steps.

By the end of the chapter you'll have a dated baseline page, a why card, a fixed cue for your sessions, a one-page spec for a first small project, and a trace table you've checked by hand. You'll also have a plan for the next month that fits the life you actually have.

## Understand it yourself

### The plain picture

An algorithm is a recipe written for a cook who does exactly what it says. Not what you meant—what it says. Tell this cook "add salt" and they'll stand there, because you didn't say how much, or when, or which bowl. Tell them "stir until smooth" and they might stir forever, because they don't know what smooth looks like. Every ambiguity in your recipe becomes a bug.

A spec comes before the recipe. It's a short written description of what a program must do, with examples and a line saying what "done" means. If the algorithm is the recipe, the spec is the note on the fridge that says: "Dinner for six, one vegetarian, on the table by 6:30, nothing with nuts."

### The precise picture

An algorithm is a finite list of unambiguous steps that takes some input and produces some output. Each word in that definition is doing work. *Finite*: it must stop. *Unambiguous*: each step has exactly one meaning. *Input and output*: it starts from something and gives back something.

A good spec has six parts, and SPEC-01 The One-Page Spec uses all of them:

1. **Purpose**—one sentence on what it's for.

2. **User**—who will use it (often you).
3. **Inputs**—what goes in.
4. **Outputs**—what comes out.
5. **Rules**—what it must always or never do.
6. **Done line**—how you’ll know it’s finished, written so you can check it.

The done line is the part people skip, and it’s the part that saves you. “Make it work” is not a done line. “Given four people and six chores, it prints six lines, each with one chore and one name, and nobody gets three chores in a row” is a done line. You can check it.

Examples belong in the spec too. SPEC-02 Examples Before Code asks you to write pairs—this input, that expected output—including edge cases. An edge case is an input at the boundary, or an odd one a careless user might type: an empty list, one person instead of four, a zero, a word where a number should be. Writing the edge cases before you write any code is the single cheapest way to avoid a whole category of bugs.

## A name for all this, once

Educators have a name for the habits underneath all of this: **computational thinking**—breaking a problem into parts, spotting patterns, leaving out what doesn’t matter, and writing steps a machine can follow. That’s the only time this book will use the phrase. From here on, we’ll just name the habits.

## Searching and sorting as counting work

Two ideas from computer science explain a surprising amount of the world, and you can learn both with a deck of cards.

**Searching.** Suppose you’re looking for the seven of hearts in a shuffled pile. The only reliable way is to look at each card in turn until you find it. That’s *linear search*. If the pile has 52 cards, the worst case is 52 looks. Double the pile, double the looks.

Now suppose the pile is sorted, lowest to highest. You can open it in the middle, see whether you’ve gone too far or not far enough, and throw away half the pile with one look. Then do it again with the half that’s left. That’s *binary search*. With 52 sorted cards, you need at most six looks. Double the pile, and you need just one more look. ALG-01 Two Ways to Search has you count this by hand, and the counts are memorable.

The catch: binary search only works on a sorted pile. Using it on a shuffled one is a classic wrong answer, and it fails quietly—it just doesn’t find things that are there.

**Sorting.** To sort a pile by hand, one way is to look through the whole pile for the smallest card, set it aside, then look through what’s left for the next smallest, and so on. That’s *selection sort*. It’s easy to understand and slow on big piles, because you keep re-scanning. Another way is to split the pile in half, and halves into halves, until every pile has one card, then merge pairs of piles back together in order. That’s *merge sort*. It feels fussier with real cards, but it does far fewer comparisons on big piles. ALG-02 Sort by Hand, Count the Work has you tally the comparisons both ways.

You don't need to memorize either algorithm. What matters is the idea underneath: different correct algorithms can do wildly different amounts of work, and you can count that work on paper before you ever touch a computer.

## Trace tables

A trace table is how you run a program in your head without fooling yourself. You make one column for each variable and one row for each step, and you fill in the values as the program would. Here's a three-line loop:

```
total = 0
for n in [4, 7, 2]:
    total = total + n
print(total)
```

| Step   | n | total |
|--------|---|-------|
| start  | — | 0     |
| loop 1 | 4 | 4     |
| loop 2 | 7 | 11    |
| loop 3 | 2 | 13    |
| print  | — | 13    |

The program prints 13. The trace table caught nothing here, but it's in the habit of catching things that you won't see by squinting. ALG-03 Pseudocode to Trace Table is the card; you'll use trace tables in every chapter after this one.

## Wrong answers you should be able to hear

These come up in your own work and in your child's, and each has a gentle reply.

- **A spec with no done line.** "It should pick chores fairly." Say: "What exactly should happen when it's done? How would you check it?"
- **"Make it work" as a requirement.** It means the spec isn't written yet. Say: "Work how? Give me one example: this goes in, that comes out."
- **A binary search on an unsorted pile.** It fails without complaint. Say: "What has to be true about the pile before we can split it?"
- **A trace table that skips the loop's last pass.** This is the most common tracing slip there is. Say: "How many items are in the list? How many rows do we have?"
- **"It should just work."** Usually there's no spec at all. Say: "Let's write down what 'work' means first."

## A five-minute warm-up

Write the steps for brushing your teeth, as if for a robot that has never seen a toothbrush. Then circle every ambiguous word. "Some" toothpaste. "Brush" for a while. "Rinse." Most people circle six

or seven words in a ten-step list. Each circle is a spec bug you’ve just learned to see.

## How to run it this week

This week’s three sessions use the Hour’s steps from the start. You won’t open a code editor until Chapter 2—this is all paper—but you’ll Specify, Predict, Build and Test, Debug and Log, and Explain and File just the same. The “build” is a page instead of a program.

### Session 1 (desk, about 30 minutes): Baseline and why

#### In-home — START-01 The Baseline Page

- **Time:** 30 minutes (desk)
- **Materials:** your notebook, a pencil, a timer
- **Safety and privacy:** your own notebook; nothing online
- **The fun:** finding out you know more than you thought, or getting to start completely fresh
- **The skill:** writing, from memory and with no help, what you can already read, trace, and build
- **The crumb:** a dated baseline page

**How it goes.** At the top of a fresh page, write today’s date and “Baseline.” Then answer three questions in plain words, with no looking anything up: What can I read? (A recipe? A spreadsheet formula? A few lines of any programming language?) What can I trace? (Could I say what a short loop does?) What can I build? (Anything—a spreadsheet that totals a column counts.) Then copy the three-line loop from the trace table section without looking at the answer, and predict what it prints. Write the prediction down. Check it against this chapter. The page is your starting line, and in a few months it will be the most encouraging page in your notebook.

Then do START-02 Why Card: on an index card, one sentence each for why you’re learning, for whom, and what you’ll build first. “To help my daughter with Python, and for me, because I’ve always wondered. First: a chore picker for our house.” Tape it inside the notebook’s front cover.

**Self-talk for Specify, this session:** “What exactly will be on this page when I’m done, and how will I know I’m finished?” The done line here is easy: a dated page with three answers and one checked prediction.

### Session 2 (micro, about 15 minutes): Cue and kit

START-03 Cue and Kit asks you to choose a fixed cue—an event already in your day that will mean “session now.” After the morning coffee. After school drop-off. After the dishes on Tuesday and Thursday. A cue tied to something that already happens is far sturdier than a time on a calendar. Then set out the kit: notebook, pencil, index cards, and (once you have it) one saved file. Leave the kit where you’ll see it at the cue.

Habits form over weeks to months, not days. In one well-known study of everyday habits, the median time for a new behavior to become automatic was about 66 days, with a wide range—and

one missed day didn't materially set anyone back.<sup>2</sup> That's a useful study, not a promise that every reader will see the same timeline. The practical message: expect it to take a while, and don't treat a missed session as a broken streak.

Finish with PLAN-01 The Week on One Card for next week.

### Session 3 (desk, about 40 minutes): Your first spec

Use SPEC-01 The One-Page Spec for a small program you'd actually like to have. The worked session below shows one start to finish.

#### Stopping when the timer ends

Stop when the timer ends, even mid-thought. Write one line in your log saying exactly where you stopped—"Halfway through the examples table; next is the empty-list case"—and close the notebook. Stopping on time is what makes it possible to start again on Thursday. Unfinished work with a clear next step is a gift to your future self.

#### Worked session: a chore-rotation picker

*An illustration, not a reported family:* A mother of three has forty minutes at the dining table on a Wednesday evening.

**Specify (8 minutes).** She writes SPEC-01 for a chore-rotation picker:

**Purpose:** Each week, assign each chore to one person, rotating so nobody gets the same chore two weeks running. **User:** Me, every Sunday night. **Inputs:** a list of chores, a list of people, the week number. **Outputs:** one line per chore: the chore and who has it. **Rules:** every chore gets exactly one person; the assignments shift by one each week. **Done line:** for three chores, two people, and week 1, it prints three lines, and week 2 gives each chore a different person than week 1 did.

Then she adds SPEC-02 examples, including the edge cases:

| Chores                 | People | Week | Expected                     |
|------------------------|--------|------|------------------------------|
| dishes, trash, laundry | A, B   | 1    | dishes→B, trash→A, laundry→B |
| dishes, trash, laundry | A, B   | 2    | dishes→A, trash→B, laundry→A |
| (none)                 | A, B   | 1    | nothing printed              |
| dishes                 | (none) | 1    | ???                          |

She stops at the last row. What *should* happen with no people? She doesn't know, and that's the point: the spec found a question before any code existed.

**Predict (2 minutes).** She sketches the steps in plain words—"for each chore, count along the people list, starting further along each week, wrapping around at the end"—and predicts the first example will match the table.

**Build and Test (20 minutes).** Her “build” is the algorithm, in pseudocode, and a trace table for the first example. You won’t write Python until Chapter 2, but here is what her plan becomes in a few weeks, so you can see where it’s going:

```
def pick(chores, people, week):
    for i in range(len(chores)):
        person = people[(i + week) % len(people)]
        print(chores[i], "->", person)

pick(["dishes", "trash", "laundry"], ["A", "B"], 1)
# dishes -> B
# trash -> A
# laundry -> B
```

The % sign means “the remainder after dividing,” which is how the counting wraps around the list. Her hand trace of week 1 matched the expected row exactly.

**Debug and Log (6 minutes).** She traces the empty-people row. Dividing by the length of an empty list means dividing by zero. On paper, the step can’t be done at all. Her log:

**Expected:** something sensible when there are no people. **Actual (on paper):** the step “count along the people list” has nothing to count; it divides by zero. **Guess:** the spec never said what to do with no people. **One change:** add a rule to the spec. **Result:** new rule: “If there are no people, print ‘Add someone first’ and stop.”

(When she ran the Python version weeks later, Python agreed: it stopped with `ZeroDivisionError: integer modulo by zero`. The paper trace had found the same bug, a day earlier, with a pencil.)

**Explain and File (4 minutes).** “I specified a chore picker for two people and three chores. I predicted week 1 would match my table, and on paper it does. The empty-people case divides by zero. Next I’d change the spec to say ‘Add someone first.’ Help I used: none.” She files the spec and the trace in her notebook and photographs both pages to her backup folder.

## Practice that actually builds it

### Blocked, then mixed

Start with **blocked practice**: three specs in a row, one each session or all in one deep session. Pick three small, real things—a tip splitter, a reading log that totals pages, a grocery list that flags anything you bought last week. Write all six parts and at least four examples each, including two edge cases. By the third, you’ll write the done line first without being told.

Then switch to **mixed practice**: in each session, do one spec, one trace, and one sort or search by hand. Mixing feels harder, and that’s partly why it works: you have to recognize which tool the problem needs, which is the real skill.

## Diagnose this

Here's a spec someone handed in. What's missing?

**Purpose:** A program for tracking my reading. **User:** me. **Inputs:** books. **Outputs:** my reading stats. **Rules:** should be easy to use. **Done line:** when it works.

Nearly everything. “Books” isn’t an input you can type—a title? pages? a date finished? “Reading stats” could mean anything. “Easy to use” can’t be checked. And “when it works” is the classic non-done line. A repaired version: “Inputs: title, pages, and date finished for each book. Output: total pages this month and the number of books. Done when three sample books from the examples table give 812 pages and 3 books.” Now you could build it—and know when to stop.

## In-home — SPEC-03 The Literal Reader

- **Time:** 15 minutes (micro)
- **Materials:** bread, a spread, a plate, a butter knife; a written list of steps; two colored pens
- **Safety and privacy:** a butter knife only; at home, with people who’ve agreed to play
- **The fun:** watching a grown adult (or a delighted child) refuse to open the bread bag because you didn’t say to
- **The skill:** repairing every ambiguity in written steps
- **The crumb:** marked-up steps in two colors

**How it goes.** Write the steps for making a jam sandwich. Hand them to a partner or your child, who follows them *exactly* as written—and no more. “Put jam on the bread” may produce a jar set on top of a closed loaf. Every wrong turn is a spec bug. Mark each one in the first color; write the repair in the second. Then run it again. Most people need three rounds. With a young child as the literal reader, expect laughing; with a teen, expect a certain rigorous joy in being unhelpful. Either way, you’ll never write “add salt” without a quantity again.

## Out-of-home — OUT-01 Algorithms in Public

- **Time:** 20 minutes (micro)
- **Materials:** your notebook, a pencil
- **Safety and privacy:** observe only; no photos of people or screens; don’t hold up a line
- **The fun:** seeing the rules behind ordinary machines
- **The skill:** writing a spec from observation
- **The crumb:** a spec page for a machine you’ve watched

**How it goes.** Pick a self-checkout, an elevator, or a crosswalk signal. Watch it work for a few minutes, then write its spec as if you were its programmer. What are the inputs (a scanned item, a weight on the bagging area, a button press)? What are the outputs? What are the rules? The best part is the edge cases: what happens when you put an item down without scanning it, or press the elevator button for the floor you’re already on, or press the crosswalk button twice? Write your prediction, then—if it’s safe and polite—watch for the answer. You’ll start noticing that every machine you use was once a spec somebody wrote, and that some of them were missing a row.

## More from the bank for this chapter

- **ALG-04 Recipe or Algorithm?** (micro): rewrite a household routine—laundry is ideal—as precise steps with one loop (“repeat until the basket is empty”) and one decision (“if it’s wool, then...”).
- **SPEC-05 Spec Diff** (micro): after any small project, compare what the spec said with what you actually built, and write a two-line change note. It’s the habit that keeps specs honest.
- **PLAN-03 Scope It Down** (micro): take a project idea that’s too big (“an app for the whole family’s schedule”) and cut it to a version you could finish in two weeks of sessions (“print this week’s pickups from a list I type in”).

## Talk to yourself (this chapter)

**Opening:** “What exactly should happen when this is done, and how small can I make it?”

**Follow-ups:** 1. **Prediction** — What did I expect this step, or this example, to produce? 2. **Evidence** — What does my trace table actually show, row by row? 3. **Location** — Which row, or which word in my spec, is the first place it goes wrong? 4. **Change** — What’s the one rule or example I’ll add, and what will it fix? 5. **Revision** — Is the problem in my steps, or did my spec never say what I wanted?

**How to wait:** set a five-minute timer; read your spec aloud as if to a stranger; trace one more row by hand.

**What a stuck silence usually means:** the done line is missing. Go write it before anything else.

**Don’t say / say instead:** Don’t say “I’ll figure it out as I go.” Say instead “Done means ; *I’ll check it by* .”

## Coaching your kids at every age

You don’t need to be ahead of your child to coach this chapter. You need the habits you just practiced: a done line, a prediction, a trace. Here’s how each translates.

### Ages 5–10

**The move:** be the literal robot. COACH-01 Literal Robot for a Five-Year-Old turns your child into the programmer and you into a machine that does exactly what it’s told. Put a stuffed animal across the room. Your child gives you steps—“walk forward,” “turn”—and you do *exactly* that. Walk forward until you bump into the couch, because they didn’t say how many steps. Turn in a full circle, because they didn’t say which way or how far.

**Exact words:** “What will I do if you tell me that?” and “How many steps?” Then do what they said, cheerfully, even when it’s wrong.

**One thing to avoid:** fixing their instructions by doing what they meant. The whole lesson is in the gap between what they said and what they meant; if you quietly close the gap, the lesson vanishes.

*Computing for Little Thinkers*, Chapter 2, “Steps and Algorithms,” has a full set of these games if you want more.

## Ages 11–14

**The move:** ask for a written prediction before anything runs (COACH-02 Ask for the Prediction). At this age, kids often want to press Run and see. That’s fine—after the prediction. One line on their page: “I predict it will print...”

**Exact words:** “What do you think it’ll do? Write it down first.” And when it surprises them: “Ooh. Where did your guess and the computer part ways?”

**One thing to avoid:** treating a wrong prediction as a mistake. A wrong prediction is the most useful event of the session. If you say “oops,” they’ll stop writing predictions they’re unsure of, which are exactly the ones that teach.

## Ages 15–18

**The move:** read their spec for a done line. Teens can build real things, and they often skip straight to building. Your job is the question that makes a project finishable.

**Exact words:** “How will you know it’s finished?” If the answer is “when it works,” ask for one example: “This goes in, what comes out?”

**One thing to avoid:** writing the done line for them. Ask, wait, and let them write it. If they ask you what *you* think done looks like, that’s a great conversation—and they still hold the pencil.

## At every age: “I don’t know yet”

COACH-09 What I Don’t Know Yet is the move that makes all the others possible. When your child asks something you can’t answer, say “I don’t know—let’s find out,” and then show them how you look. Children learn far more from watching a calm adult search than from an adult who always has the answer.

## If it isn’t clicking

**Your specs keep ballooning.** Every spec turns into a plan for the whole family’s digital life. Use PLAN-03 Scope It Down: write the big version, then cross out everything that isn’t needed for the first useful version. If you can’t finish it in two weeks of sessions, cut again.

**Your traces skip steps.** You trace four rows when the loop has five, or you jump to the answer. Slow down to one row per line of code, even when it feels silly. Put your finger on the line you’re tracing. The skipped row is almost always the last pass through a loop.

**Your algorithms are vague in the middle.** The first step and the last step are clear, and everything between them says something like “sort out the rest.” That middle is where the program lives. Take the vague line and ask of it what a literal reader would ask: which item first? compared with what?

what happens when two are equal? when do I stop? Write each answer as its own numbered step. If a step still needs judgment, split it again. You're done when a stranger could follow the steps with a pencil and get the same answer you would, without asking you a single question.

**The habit won't stick.** Three weeks in, sessions keep not happening. Shrink every session to micro, and move the cue (START-03). A cue that depends on a quiet evening will fail in a busy week; a cue tied to the morning coffee will usually survive. Also check your why card—if it no longer sounds like you, rewrite it.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. If you've logged two sessions stuck on the same thing, ask a human. OUT-03 Find Your People is the card for it: visit a library coding group, a meetup, or a community-college open house. Learning with someone else helps; in programming courses, working in pairs has shown positive effects on learning across many studies.<sup>8</sup> That evidence comes from classroom courses, not home study, but a friend at the next chair seems to work the same way for many adults. And having nobody to learn with is one of the most common frustrations adults report—so this one's worth an evening.

## Out-of-home — OUT-03 Find Your People

- **Time:** 45 minutes (desk)
- **Materials:** your notebook; one question you'd like answered
- **Safety and privacy:** a public meeting; share only your first name if you prefer
- **The fun:** meeting people who are also figuring it out
- **The skill:** finding a place to learn with others
- **The crumb:** notes and the date of the next meeting

**How it goes.** Look for a coding group at your public library, a local meetup for beginners, or an open house at a community college's continuing-education program—check what's near you. Go once, with one question written down. You don't have to say anything; listening counts. Before you leave, write the date of the next meeting in your notebook. Most adults say the second visit is much easier than the first.

## Tools, including AI

This chapter's tools are a pencil, index cards, a deck of playing cards, and a notebook. That's on purpose. Specs and traces are thinking work, and paper keeps the thinking yours.

For AI tools, everything in the AI box in **The Computing Hour** applies. One thing is specific to this chapter: write your first spec yourself, without asking a tool to write it for you. A tool can produce a tidy-looking spec in seconds, but it can't know what you actually want, and the act of discovering what you want—like the mother who found she didn't know what should happen with no people—is the skill. Once your spec is written, it's fine to ask a tool “What edge cases might I be missing?” and then decide for yourself which ones matter. Log it.

You may hear claims that learning to program makes people better at thinking in general. The

evidence for that kind of broad transfer is modest: skills near programming tend to benefit more than distant ones.<sup>9</sup> This is a useful study, not a promise that every reader will see the same result. Learn programming for what it is. That's plenty.

## What “done enough” looks like

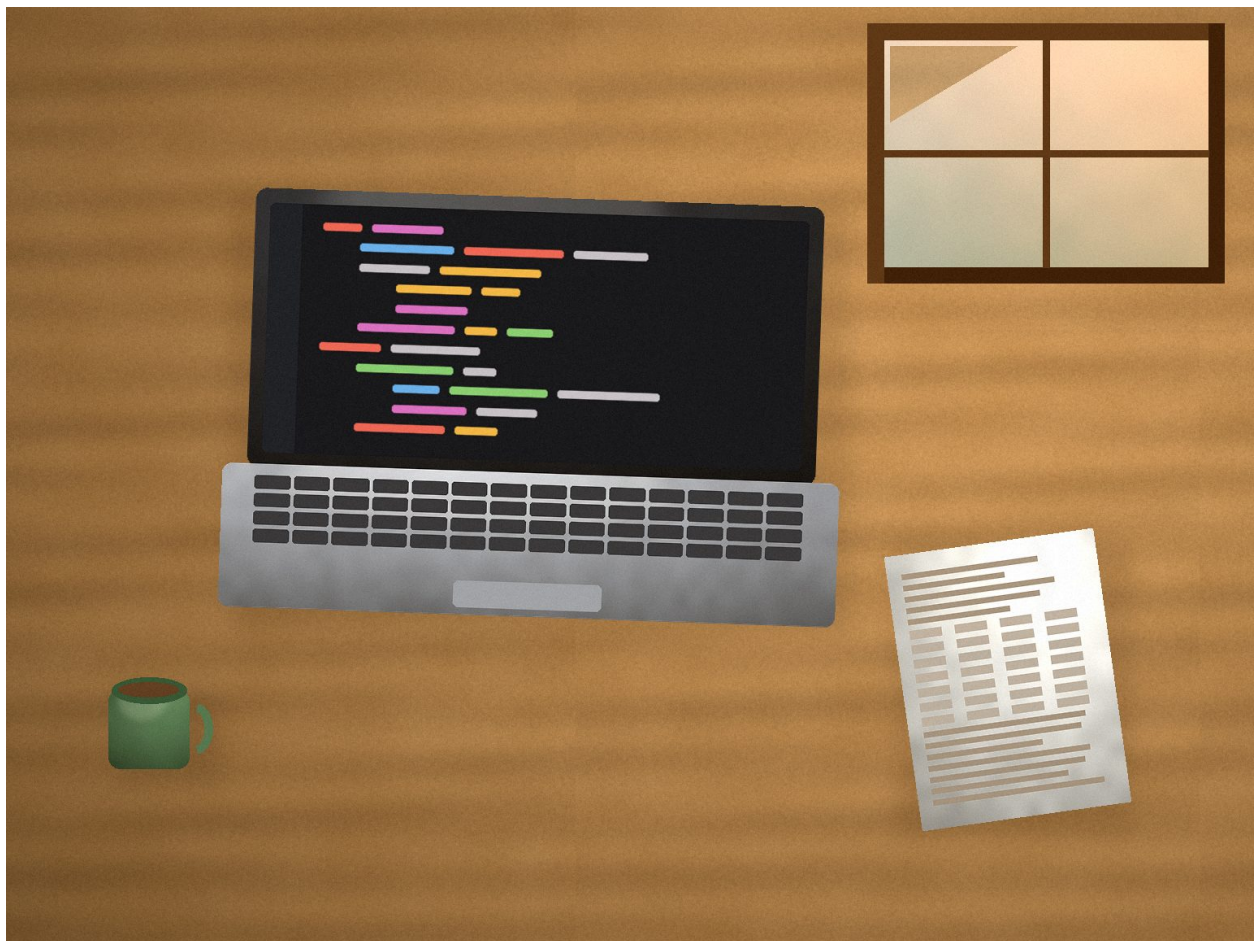
You're ready to move on to Chapter 2 when you have:

- ☐ a dated baseline page (START-01)
- ☐ a why card taped inside your notebook (START-02)
- ☐ a cue and a kit you've used at least three times (START-03)
- ☐ one SPEC-01 spec with all six parts and a checkable done line, plus a SPEC-02 examples table with at least two edge cases
- ☐ a correct trace table for a short loop, including the last pass (ALG-03)
- ☐ step counts for linear and binary search on at least two pile sizes (ALG-01)
- ☐ AS-01 Cold Trace once: trace a short program you haven't seen, unaided, and predict its output
- ☐ AS-02 Debug on the Spot once: find a planted mistake in a short set of steps while explaining aloud (a partner can plant it)
- ☐ AS-04 Explain Your Code once: walk a partner, or a voice recorder, through your spec and trace line by line

Place yourself by these skills, not by how long you've been at it. If you can already do all of them, move on today. If two weeks in you can do most of them, that's done enough; the rest will come with practice in Chapter 2.

# Chapter 2

## Python as a Working Tool



### Why this matters

A program is a tool you can keep. Write a recipe scaler once and it scales every recipe for the rest of your life. Write a little script that totals the miles in a log and it totals next year's log too. That's the grown-up reason to learn Python: not to become a different kind of person, but to own a few small tools that do real jobs in your house, and to understand the ones your kids are building.

Python is a programming language—a precise way of writing steps a computer can follow. This book uses Python because it reads close to plain English, because its error messages are more helpful than most, and because it’s the typed language the later books in this series use for children. When your twelve-year-old moves from blocks to text, it will very likely be Python on the screen, and you’ll recognize it. This isn’t a claim that Python is the best language; it’s a good first one, and the habits you build here carry to any other.

The chapter teaches a small set of ideas well: variables and input, functions that hand back a value, reading and writing files, lists and dictionaries, finding answers in the official documentation, and one small class designed on index cards first. By the end you’ll have a program that reads a file, does something useful with it, and writes a report—and you’ll be able to explain every line.

Beginners learn programming best from guided examples, not from being dropped in front of a blank screen and told to discover it.<sup>10</sup> So this chapter works in a particular shape, borrowed from a well-tested approach to teaching programming in schools: **predict** what a short example will do, **run** it, **investigate** how it works, **modify** one thing, and then **make** something of your own.<sup>11</sup> You’ll see that shape in every session.

## Understand it yourself

### The plain picture

A **variable** is a labeled jar. `cups = 2` means “put 2 in a jar labeled *cups*.” Later you can look in the jar, or empty it and put something else in.

A **function** is a recipe card you can call by name. You write the card once—“to scale a recipe, take the amount, multiply by the new servings, divide by the old servings”—and after that you just say “scale” with the numbers, and the card does the work and hands you the answer.

A **list** is a shopping list: things in order, numbered from the top. A **dictionary** is a set of labeled drawers: you find things by their label, not their position.

A **file** is a page the program can read from or write on, which stays there after the program stops.

### The precise picture

A variable is a name bound to a value. Every value has a **type**—what kind of value it is: a whole number (`int`), a decimal number (`float`), text (`str`, short for “string”), a list, and so on. Types matter, because the same symbol does different things to different types. `3 * 2` is 6. `"3" * 2` is `"33"`—text repeated twice.

`input()` asks the person at the keyboard to type something and gives it back—always as text, even if they type a number. This is the single most common surprise for adult beginners, and the worked session below is built around it.

A function is defined with `def`, takes **parameters** (the names for its inputs), and **returns** a value with `return`. The **return value** is what it hands back to whoever called it. `print()` is different:

it shows something on the screen and hands back nothing useful. A function that prints its answer instead of returning it looks fine on screen and is useless to the rest of your program.

A **docstring** is a short description in quotes on the first line inside a function, saying what it does. It's a spec, built into the code.

Files are opened with `with open(...) as f:`. The `with` means Python closes the file for you when the indented block ends, even if something goes wrong.

Lists are accessed by position, starting at zero: in `["milk", "eggs"]`, "milk" is item 0. Dictionaries are accessed by key: in `{"milk": 3}`, the key "milk" gives you 3.

A **class** is a blueprint; an **object** is one thing built from it. You'll meet one small class near the end of the chapter, on index cards first.

## Wrong answers you should be able to hear

- **input() treated as a number.** "I typed 2, why did it print 222?" Say: "What type does `input()` give back? Let's print it."
- **A function that prints instead of returns.** The answer appears, but the next line gets `None`. Say: "What does this function hand back? Not show—hand back."
- **A dictionary used where order matters to the reader.** A shopping list stored as a dictionary loses the meaning of "first" and "last" for whoever reads the code, even if the program happens to keep it. Say: "Do you need to find things by name, or keep them in order?"
- **Reading a file without closing it.** It often works, until it doesn't. Say: "Let's use `with`, so Python closes it for us."
- **A class with data but no actions.** A "Book" class that only stores a title is just a labeled box. Say: "What does a book *do* in your program? Check out? Renew?"

## A five-minute warm-up

Predict what each of these prints, on paper, before reading on.

```
print(3 * 2)
print("3" * 2)
print(10 / 4)
print(10 // 4)
```

They print 6, 33, 2.5, and 2. The second surprises most people (text repeated), and the fourth surprises nearly everyone: `//` divides and throws away the remainder. If you got all four, wonderful. If you missed one, even better—you've just found a corner of your picture to repair, and that's exactly what PY-01 Predict the Output is for.

## How to run it this week

### Session 1 (desk, about 40 minutes): Set up once

#### In-home — START-04 Set Up a Clean Workspace

- **Time:** 40 minutes (desk)
- **Materials:** a computer you're allowed to install software on; your notebook
- **Safety and privacy:** download only from the official site for the editor or Python itself; practice folder only
- **The fun:** the first time a computer says hello because you told it to
- **The skill:** installing a Python editor, making one learning folder, and saving a first file in two places
- **The crumb:** a folder with a hello file and a backup copy

**How it goes.** Install a beginner-friendly Python editor. Thonny is a good choice: it comes with Python built in, and its debugger lets you step through a program one line at a time.<sup>12</sup> Any plain editor plus Python 3 from the official site works too; check the current version of whichever you choose rather than trusting a screenshot from an old tutorial. Then make one folder called something like `learning`, and inside it a file called `hello.py` with one line: `print("Hello from my kitchen table")`. Before you run it, write your prediction. Run it. Then copy the folder to a second place—a USB drive or your own cloud storage. Every file in this book gets saved in two places, starting tonight.

### Session 2 (micro, about 15 minutes): Predict the Output

PY-01 Predict the Output: take the warm-up above, or any three short lines from this chapter, write your predictions, run them, and write the result next to each prediction. Log any surprise in the Debug Log. Fifteen minutes, three lines of code, and you've practiced the most important habit in the book.

### Session 3 (desk, about 35 minutes): A calculator you need

PY-02 Variables, Input, and a Calculator You Need. The worked session below is one version of it; your version should solve a calculation that turns up in your own house. A mileage log. A tip splitter. Paint coverage for a room. Pick one you'll actually use.

**Self-talk for Predict, this chapter:** "What will this print, line by line, before I run it?" Say each line aloud with its predicted output. If you can't predict a line, that's the line to investigate.

### Stopping when the timer ends

Stop mid-program if the timer says so. Write one comment at the top of the file—`# NEXT: handle the case where someone types a word`—and save it in two places. Tomorrow-you will open the file and know exactly where to start.

## Worked session: the recipe scaler

*An illustration, not a reported family:* A father who bakes on weekends has thirty-five minutes on a Thursday night.

**Specify (5 minutes).** On an index card:

Recipe scaler. Input: cups of an ingredient for 4 servings; the servings I want. Output: cups for the new servings. Done when 2 cups for 4 gives 3 cups for 6, and 1.5 cups for 4 gives 0.75 for 2.

**Predict (2 minutes).** He starts small: just ask for the cups and multiply, to see the input working. “If I type 2 and multiply by 3, I’ll see 6.”

**Build and Test (15 minutes).**

```
cups = input("Cups for 4 servings: ")    # he types 2
print(cups * 3)
```

The screen shows 222.

**Debug and Log (8 minutes).** He doesn’t change anything yet. He writes:

**Expected:** 6 **Actual:** 222 **Guess:** it’s treating 2 as text and repeating it, like “3” \* 2 in the warm-up. **One change:** turn the input into a number with `float()`. **Result:** see below.

```
cups = input("Cups for 4 servings: ")    # types 2
cups = float(cups)
print(cups * 6 / 4)                      # prints 3.0
```

It prints 3.0. That’s 3 cups—his spec said 3, and 3.0 is Python’s way of writing a decimal 3. He updates the spec: “3.0 is fine.” Then he turns the calculation into a function so he can reuse it:

```
def scale(amount, old_servings, new_servings):
    """Return amount scaled from old_servings to new_servings."""
    return amount * new_servings / old_servings

print(scale(2, 4, 6))    # 3.0
print(scale(1.5, 4, 2))  # 0.75
```

Both examples from his done line pass.

**Explain and File (5 minutes).** “I specified a recipe scaler. I predicted 6 and got 222, because `input()` gives text. I changed it with `float()` and got 3.0, then wrote a `scale` function and both examples pass. Next I’d change it to ask for the servings instead of typing them into the code. Help I used: none.” He saves it in his learning folder and his backup, and he can explain every line, including the docstring.

## Modify: when someone types a word

PRIMM's fourth step is *modify*, and the recipe scaler offers a perfect one. Run it again and, when it asks for cups, type two instead of 2. Predict first. Most people guess “it'll print something strange.” Instead, the program stops, and the last line of the red text says:

```
ValueError: could not convert string to float: 'two'
```

Read it slowly, because it's being helpful. *ValueError*: the type was right (text) but the value was one Python couldn't use. *Could not convert string to float*: it tried to turn text into a decimal number. 'two': here's the exact text it choked on. (Stray spaces, by the way, are fine: `float(" 2 ")` quietly gives `2.0`. It's letters that stop it.)

You don't have to fix this today. A good move for now is to add a line to your spec—“If someone types a word, the program may stop; that's acceptable for version one”—and a note for later. Chapter 3 shows how to test for it, and how to decide, in the spec, what *should* happen. Deciding is the part that's yours. Python can tell you what went wrong; only you can say what you wanted instead.

That's the rhythm of the whole chapter in small: predict, run, investigate the surprise, change one thing, and write down what you learned. None of it requires you to know more Python than you did ten minutes ago. It requires you to look closely, which you already know how to do.

## Practice that actually builds it

### Blocked: three small functions

Write three functions in a row, each with a docstring and a return value, each tested with two examples you write *before* the code (PY-03 Functions With Return Values):

1. `tip(bill, percent)` returns the tip.
2. `split(total, people)` returns each person's share.
3. `total_with_tip(bill, percent, people)` *calls the first two* and returns each person's share including tip.

The third is the real lesson: a function calling another function. Predict what `total_with_tip(60, 20, 3)` returns before you run it. (The tip is `12.0`, the total is `72.0`, and each share is `24.0`. Division in Python always gives a decimal number, which is why the `.0` appears.)

### Diagnose this: print instead of return

Here's a function someone wrote. Predict the two lines of output before reading on.

```
def total(prices):
    s = 0
    for p in prices:
        s = s + p
    print(s)

result = total([2, 3, 5])
```

```
print(result)
```

It prints 10 and then None. The first line comes from the print inside the function. The second shows what the function handed back: nothing, which Python calls None. The fix is one word: change `print(s)` to `return s`. After the fix, the program prints one line, 10, and `result` holds a number the rest of your program can use.

## Read a file, count something

PY-04 Read a File, Count Something is the first program in this book that feels like a real tool. Make a text file called `list.txt` in your learning folder with one grocery item per line:

```
milk
eggs
milk
bread
milk
```

Then this program counts how many times each item appears and writes a small report:

```
counts = {}
with open("list.txt") as f:
    for line in f:
        item = line.strip()
        counts[item] = counts.get(item, 0) + 1
print(counts)

with open("report.txt", "w") as out:
    for item in sorted(counts):
        out.write(item + ": " + str(counts[item]) + "\n")
```

The screen shows `{'milk': 3, 'eggs': 1, 'bread': 1}`, and a new file, `report.txt`, appears in your folder with three lines: `bread: 1`, `eggs: 1`, `milk: 3`. Before you run it, predict both. Then investigate: what does `strip()` do? (It removes the invisible end-of-line character and any stray spaces; without it, "milk" and "milk " would be different items.) What does `counts.get(item, 0)` do? (It looks in the drawer labeled `item`, and gives 0 if there's no such drawer yet.) Then modify one thing—count only items that appear more than once—and predict again.

## Mixed: list or dictionary?

PY-05 Lists, Dictionaries, and When is a micro session on paper. For each job below, choose a list or a dictionary and write one sentence saying why:

- the order you'll visit stores on Saturday (a list—order matters)
- how many of each item you need (a dictionary—you look things up by name)
- this week's chore rota from Chapter 1 (try it both ways and decide)
- the books each child has checked out (a dictionary whose values are lists)

There's no single right answer for the third; the reason is the skill.

## Reading the official documentation

PY-06 Use a Library, Read Its Docs. Python comes with a large standard library—ready-made modules for dates, files, random numbers, and much more. The official documentation at [docs.python.org](https://docs.python.org) is the authority, and it’s written for people who already know a little, which makes it intimidating at first. The trick: read the examples first, and only then the explanation around them.

Try it with dates. Suppose you want to know the due date of a library book three weeks from a given day. Search the documentation for `timedelta`, find the example, and adapt it:

```
import datetime

borrowed = datetime.date(2026, 10, 13)
due = borrowed + datetime.timedelta(days=21)
print(due)    # 2026-11-03
```

Predict the date by counting on a calendar first. Then, in your notebook, copy the one line from the documentation that answered your question. That docs note is the crumb, and it’s also a habit you’ll ask your teen to show you.

## In-home — PY-07 A Small Class on Index Cards

- **Time:** 20 minutes (micro)
- **Materials:** three index cards, a pencil
- **Safety and privacy:** made-up titles only
- **The fun:** designing a tiny piece of software without touching a keyboard
- **The skill:** designing one class: what it knows and what it does
- **The crumb:** a card set

**How it goes.** On the first card, write the class’s name: *LibraryBook*. On the second, what every library book knows: a title, a due date, how many times it’s been renewed. On the third, what it can do: renew (push the due date later and add one to the renewal count). Now pretend to be a book. Have someone tell you “renew by 7 days,” and update your cards. Only then write it in Python:

```
class LibraryBook:
    def __init__(self, title, due):
        self.title = title
        self.due = due
        self.renewals = 0

    def renew(self, days):
        self.due = self.due + days
        self.renewals = self.renewals + 1

book = LibraryBook("Gardening Basics", 14)
book.renew(7)
print(book.title, book.due, book.renewals)
```

Here, the due date is just “day 14” to keep it simple. Predict the output before you run it: Gardening Basics 21 1. The class is the blueprint; book is one object built from it. `__init__` is the part that runs when a new book is made, and `self` means “this particular book.”

## Out-of-home — OUT-02 Library Computer Session

- **Time:** 45 minutes (desk)
- **Materials:** your notebook; your files on a USB drive or in your own cloud storage
- **Safety and privacy:** your own accounts; save to your own drive; sign out and close the browser before you leave
- **The fun:** a change of scene for a stuck project
- **The skill:** running a full session on a public computer and leaving it clean
- **The crumb:** a file saved in two places

**How it goes.** You met this card in **The Computing Hour**; this time, take this chapter’s file-counting program to a library session. Check what your library’s computers allow; if you can’t install anything, a website that runs Python in the browser, such as Python Tutor, may work for a short program (a file-reading program may need its sample lines pasted in as a list instead). Run one Hour—spec, prediction, a small change, a log line. Save to your own drive, then sign out of everything, close the browser, and log it. A change of chair is surprisingly good for a stuck bug.

## Talk to yourself (this chapter)

**Opening:** “What will this print, line by line, before I run it?”

**Follow-ups:** 1. **Prediction** — What did I write down for each line? 2. **Evidence** — What exactly did the screen show? Copy it, red text and all. 3. **Location** — Which line is the first one whose output surprised me? 4. **Change** — What’s the one change I’ll make—often, printing the value and its type—and what do I expect it to show? 5. **Revision** — Was my spec wrong about what the output should look like?

**How to wait:** set a five-minute timer; trace the code by hand in a trace table, one row per line; read the last line of the error aloud.

**What a stuck silence usually means:** a value’s type isn’t what you think. Text that looks like a number is still text.

**Don’t say / say instead:** Don’t say “Python is being weird.” Say instead “Let me print the value and its type.”

That last move is worth seeing once:

```
x = "5"
print(x + "5")      # 55
print(type(x))      # <class 'str'>
```

55, because `x` is text, and adding text joins it. `type()` tells you so.

## Coaching your kids at every age

### Ages 5–10

**The move:** COACH-07 Scratch Night. Scratch is a block-based programming environment for children; ScratchJr is a simpler version for the youngest.<sup>13</sup> With an eight-to-ten-year-old, build one tiny project together—a cat that walks across the screen and says hello—where *they* drive the mouse and *you* ask the questions. CodeAI (formerly Code.org) offers block-based courses for this age too. Use an offline editor or an account you manage as the parent.

**Exact words:** “What do you think the cat will do when you click the flag?” and, when it doesn’t: “What did it do instead? Which block do you think did that?”

**One thing to avoid:** taking the mouse. Sit on your hands if you must. Your role is the question-asker; their hand on the mouse is the whole point.

### Ages 11–14

**The move:** co-learn a function side by side (COACH-06 Co-Learning Hour). Pick a small spec together—“a function that tells you whether a number is even”—and each write your own version in your own file. Then trade, predict what the other’s will do on three inputs, and run them.

**Exact words:** “Let’s both write it, then guess what each other’s does.” And when yours is wrong—it will be, sometimes, and that’s a gift—“Ha. Mine broke on zero. Let me log it.”

**One thing to avoid:** writing yours first and then “helping.” Write at the same time, or let them go first. Being visibly wrong in front of your child, calmly, is one of the most valuable things you can model.

### Ages 15–18

**The move:** ask them to show you the docs line. When your teen solves something, ask: “Where did you find that? Show me the line in the documentation.” This is PY-06 in reverse, and it does two jobs: it nudges them toward the official source, and it tells you, without any spying, whether they understood the answer or just copied it.

**Exact words:** “That’s neat—show me the line in the docs that told you how.” If they used an AI tool: “Great—what did it say, and how did you check it?”

**One thing to avoid:** treating “I looked it up” as cheating. Looking it up is what working programmers do all day. What matters is that they can explain what they found.

If your teen is ready for more, *Computing for Emerging Adults*, Chapter 4, “Objects and Classes: The Bridge to Java,” takes this chapter’s index-card class much further, including into Java. It makes a fine side-by-side project: you design the cards, they write the code, and you each explain the other’s half.

## If it isn't clicking

**Syntax errors swamp you.** Every run is a wall of red text. Type less and run more: one new line, then run; one more line, then run. When an error appears, read only its last line first. Here's one:

```
TypeError: can only concatenate str (not "int") to str
```

That came from `print("2" + 3)`. In plain words: you tried to join text and a number with `+`, and Python won't guess which you meant. The line number above that last line tells you where. Chapter 3 teaches reading these reports—they're called *tracebacks*—from the bottom up.

**Functions confuse you.** Write the *call* before the function. Write `print(scale(2, 4, 6))` and the answer you expect (3.0) first, as a comment. Then write the function to make that line true. The call is the spec.

**The documentation feels impenetrable.** Read only the examples. Copy one, run it, change one thing, and run it again. Come back to the prose later; it makes much more sense once you've seen the example work.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. If two logged sessions are stuck on the same thing, bring the log to a person—a library coding group, a meetup, a friend who codes. Show them your Debug Log page. Most people who code are delighted to be asked a precise question, and your log makes your question precise.

**Indentation errors keep appearing.** Python reads the spaces at the start of a line as meaning: they say which lines belong inside a function, a loop, or an `if`. An `IndentationError` almost always means two lines that should line up don't, often because one was indented with a tab and the other with spaces after a paste. Set your editor to insert four spaces when you press Tab, then retype the leading spaces on the line Python names and the line above it. Read the block aloud as you go: "this line is inside the loop; this one runs after it." If you can't say which block a line belongs to, Python can't either.

**Names keep tripping you up.** A `NameError` means Python met a name it has never been told about. Nine times in ten it's a spelling or capitalization difference (`total` and `Total` are different names to Python) or a variable used before the line that creates it. Circle every name in the function on paper, and draw an arrow from each use back to the line where it was first given a value. A name with no arrow is your bug.

## Tools, including AI

**Editors.** Thonny is gentle for beginners and has a step-through debugger. Any plain-text editor works with Python 3 installed. Check the current version of whatever you install.

**Seeing a trace.** Python Tutor is a website that shows a short program running one step at a time, with every variable drawn as a box—an animated trace table.<sup>3</sup> It's excellent for the recipe scaler and the counting program.

**Official answers.** docs.python.org: the tutorial for learning, the library reference for looking things up.

**AI.** Everything in the AI box in **The Computing Hour** applies. What’s specific to this chapter: AI tools are useful for *explaining* an error message you’ve already read yourself, and risky for *writing* the functions this chapter asks you to write, because the whole point of these exercises is the struggle that builds your picture. Try first, with the timer. If you then ask a tool to explain `TypeError: can only concatenate str (not "int") to str`, log it. If a tool suggests code, keep it only after you can explain every line and it passes your examples.

## In-home — CAREER-03 Automate One Chore

- **Time:** 75 minutes (deep)
- **Materials:** your learning folder; a stopwatch; a copy of the files the chore uses
- **Safety and privacy:** work on copies, never the originals; at work, only within your employer’s rules on tools and data
- **The fun:** watching a computer do in seconds a job that used to eat your Sunday evening
- **The skill:** turning a real chore into a spec, a small script, and a before-and-after time note
- **The crumb:** the script and a before/after time note

**How it goes.** When you’re comfortable with files and functions, this is the chapter’s capstone. Choose one small, boring, repeated chore: totaling a monthly mileage log, writing next week’s chore list to a file from Chapter 1’s picker, counting how many books each child read this month from a reading list. Time yourself doing it by hand once, and write the time down. Then write a SPEC-01 spec for the script, with a done line (“given last month’s file, it prints the same total I got by hand”). Build it in small steps, predicting before every run. Work on a *copy* of the real file until the script has passed your examples at least twice; a script that changes files can make a mess faster than any person. Finally, time the scripted version, and write both times on one line in your log.

Expect the first version to take longer than the chore itself. That’s normal and it isn’t the point: the script runs next month, and the month after, and you can explain every line of it. That before-and-after note is one of the most satisfying entries you’ll ever put in a portfolio.

## Blocked, round two: a mileage log

If you’d like one more round of blocked practice before the capstone, build a mileage log from a made-up file with one trip’s miles per line. Three functions, each with a docstring and two examples written first: one that reads the file into a list of numbers, one that returns the total, and one that returns the longest trip. For a file holding 12.5, 30, and 7.5, predict the total and the longest trip before you run anything. (They’re 50.0 and 30.0; if your longest trip prints as 30 without the decimal point, look at how you turned each line into a number—`float()` and `int()` give different types, and noticing which one you used is exactly the kind of detail this chapter trains.)

## What “done enough” looks like

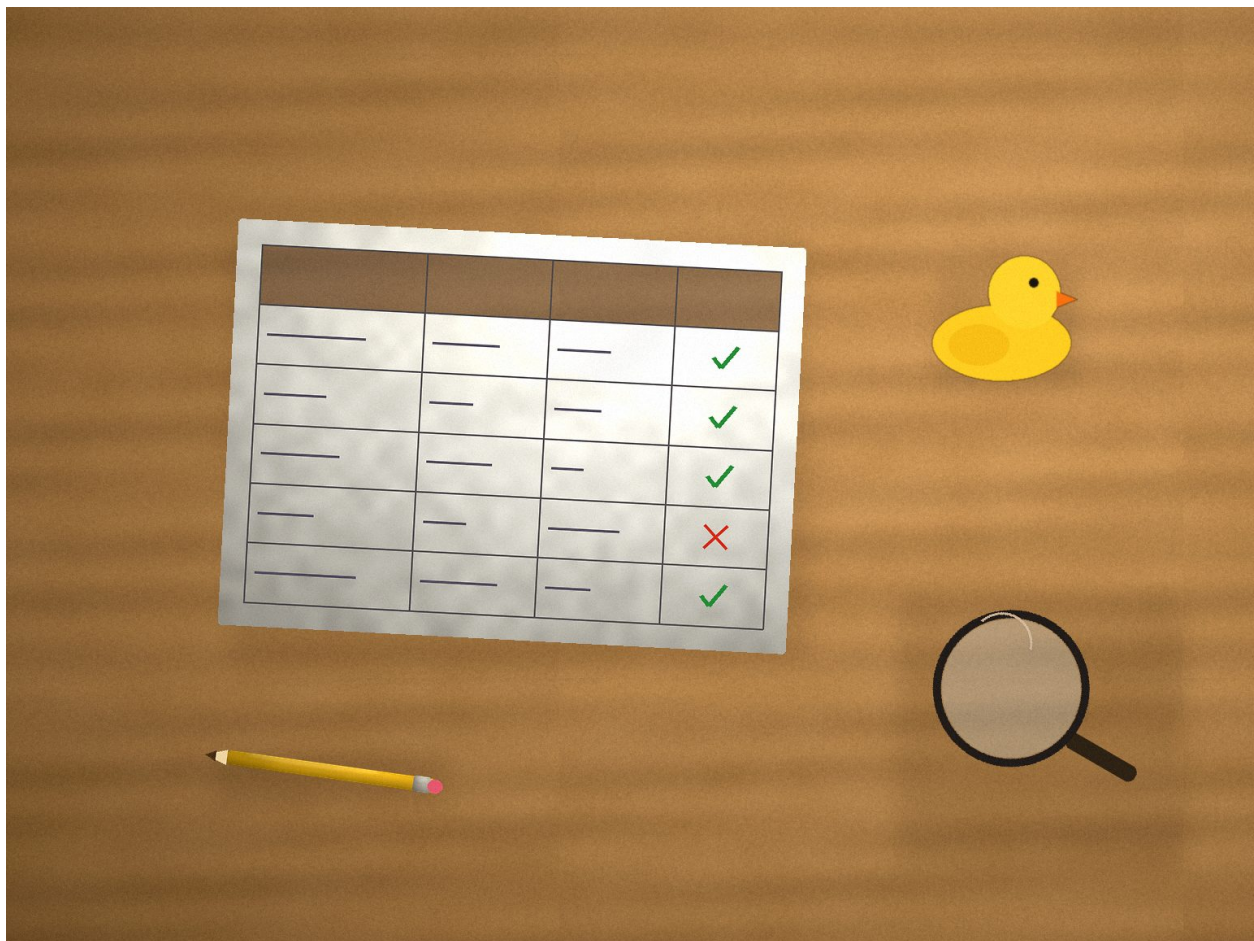
You’re ready for Chapter 3 when you have:

- ☐ a clean learning folder with a backup copy (START-04)
- ☐ a program that reads a file and writes a report (PY-04)
- ☐ at least one function with a docstring and a return value, tested with examples you wrote first (PY-03)
- ☐ a list-or-dictionary choice card with reasons (PY-05)
- ☐ one docs note: the line from the official documentation that answered a question (PY-06)
- ☐ a class designed on index cards and then written (PY-07)
- ☐ at least three Debug Log pages
- ☐ AS-01 Cold Trace once: trace a short program you haven’t seen and predict its output, unaided
- ☐ AS-02 Debug on the Spot once: a partner plants a bug (print instead of return is a good one), and you fix it while explaining aloud
- ☐ AS-04 Explain Your Code once: walk a partner through your file-counting program, line by line

Place yourself by these skills, not by the calendar. If the file program and one function feel solid, move on; you’ll use everything here again in Chapters 3 and 4.

# Chapter 3

## Testing and Debugging: Proof Before Pride



### Why this matters

Up to now you've written programs and watched them run. From here on, every program you write comes with a promise attached: a short table that says, before you type a line, what the program should do for each kind of input. Then you make the computer check the promise for you. That's testing. When the check fails, you read the evidence, write down one honest guess, and try one

change. That’s debugging. This chapter teaches both, and it teaches them as a calm habit rather than a crisis.

Here’s why it matters for you, as an adult with limited evenings. Without tests, you finish a session not knowing whether your program works. You think it does, because it printed something plausible. Next week you come back, change one line, and something else breaks quietly. You lose the evening hunting. With tests, you finish a session knowing exactly which promises hold. You come back next week, change a line, run the tests, and know in two seconds what you broke. Tests are how a busy person keeps their progress.

There’s a deeper reason too. Researchers who reviewed decades of work on how people learn to program found that the difference between novices who make progress and novices who stall is mostly strategy, not talent.<sup>1</sup> The effective ones test, trace, and work from evidence. The stalled ones change things at random and hope. That’s good news, because strategy can be learned in an evening and practiced in fifteen minutes. You don’t need a gift for this. You need a table, a pencil, and the patience to read the last line of an error message.

This chapter also carries something the earlier chapters only touched: the feeling of being wrong. Many adults learning to program catch themselves thinking some version of “I’m just bad at this.” The program fails, the red text appears, and it feels like a verdict on you. It isn’t. A bug is information about the program, not about the person who wrote it. Working programmers meet programs that don’t work yet all the time. The skill isn’t avoiding bugs. The skill is meeting them with curiosity. That’s the voice of **The Computing Hour**—*oh, a clue*—and this is the chapter where it earns its keep.

And if you’re teaching or learning beside a child, this is the chapter you’ll use most. Children meet bugs constantly, and how the adult nearby reacts teaches them more than any lesson. If you can say “Huh, it surprised us—let’s write down what we expected” in a steady voice, you’ve given them the most useful habit in computing. You’ll find exact words for each age in the coaching section.

One honest limit before we start. Testing doesn’t make a program perfect. A test only checks the cases you thought to write down. What tests give you is something more modest and more useful: proof that the promises you wrote are kept, and a fast alarm when one breaks. Proof before pride—that’s the title, and it’s the whole idea.

## Understand it yourself

### The plain picture

A test table is a promise written down before you cook. Picture a recipe card that says “bake until done.” Two cooks, two cakes. Now the card says “bake about 35 minutes; it’s done when a toothpick comes out clean.” That last line is a test. It’s written before anyone bakes, anyone can run it, and it gives the same answer no matter who holds the toothpick. A test table is a page of toothpicks for a program: for this input, it should give this output.

A **test** is a written check comparing what the program does with what it should do. You write a handful of them in a table, then turn each row into one line of code that the computer runs.

## The precise picture

Good test tables cover three kinds of input:

- **Normal** inputs—the ordinary cases the program exists for.
- **Boundary** inputs—the values right at the edge of a rule. If the rule is “50 or more,” the boundary is 50, and its neighbors 49.99 and 50.01 are worth a look too.
- **Invalid** inputs—things that shouldn’t happen but will: a negative number, an empty list, text where a number belongs.

An **edge case** is an input at the boundary, or an odd one a careless user might type. Most real bugs live at edges, which is why the table makes you write them down before you’re tired.

Here’s a test table for a small household rule. Picture a family that lets the kids order a book from a catalog, with free shipping on orders of 50 or more. (The threshold is made up for practice.)

| Input | Kind     | Expected | Why                        |
|-------|----------|----------|----------------------------|
| 60    | normal   | True     | well above                 |
| 40    | normal   | False    | well below                 |
| 50    | boundary | True     | “50 or more” includes 50   |
| 49.99 | boundary | False    | just under                 |
| −5    | invalid  | an error | an order can’t be negative |

Notice the last column. “Why” is where you discover what your spec really says. Writing “50 or more includes 50” forces the question every boundary hides: is it *more than* or *at least*?

## Asserts: making the table run

Python has a one-word way to turn a row into a check: `assert`. It takes something that should be true. If it is, nothing happens. If it isn’t, the program stops loudly. Here is the shipping rule, written the way many people first write it. The bug is deliberate, and it’s labeled:

```
def free_shipping(order_total):
    """True when the order ships free: 50 or more."""
    return order_total > 50  # DELIBERATE BUG: 50 itself is left out

assert free_shipping(60) == True
assert free_shipping(40) == False
assert free_shipping(50) == True
print("all shipping tests passed")
```

Predict before you run. If you read carefully, you’ll guess that the third assert fails, and it does. Python stops and prints a traceback whose last line is just `AssertionError`, pointing at line 7, the `free_shipping(50)` row. The words “all shipping tests passed” never appear. One change—> becomes >=—and the run prints `all shipping tests passed`.

That’s the whole mechanism. You’ll learn fancier testing tools later if you want them, but plain asserts in a file are enough for everything in this book.

## Three kinds of error

Programs go wrong in three ways, and naming them calms everyone down:

- A **syntax error** means Python can't read the code at all—a missing colon or bracket—so nothing runs.
- A **runtime error** means the program starts, reaches a line it can't carry out, and stops with a message.
- A **logic error** means the program runs, finishes, and gives the wrong answer. No red text. Only a prediction or a test catches it.

The shipping bug is a logic error. Without the test, the function would have run happily for years, quietly charging shipping on exactly-50 orders. Logic errors are why tests exist.

## How to read a traceback

A **traceback** is Python's report of where an error happened, read from the bottom up. Here's a runtime error from a function that averages a list of scores:

```
def average(scores):
    """Return the mean of a list of numbers."""
    return sum(scores) / len(scores)

print(average([8, 9, 10]))
print(average([]))
```

The first call prints 9.0. The second stops the program. The traceback looks like this (your file path will match your own folder):

```
Traceback (most recent call last):
  File "avg.py", line 6, in <module>
    print(average([]))
  File "avg.py", line 3, in average
    return sum(scores) / len(scores)
ZeroDivisionError: division by zero
```

Recent versions of Python also print small carets and tildes under the part of each line that failed; we've left them out so the shape is easy to see. Read it in this order:

1. **The last line** says what happened: the error's name and a message. `ZeroDivisionError: division by zero`.
2. **The line just above it** says where: line 3, inside `average`.
3. **Further up** says how the program got there: line 6 called `average` with an empty list.

The official Python tutorial puts it plainly: the last line of the message tells you what happened.<sup>14</sup> The place Python reports isn't always the place that needs fixing, though. Here the division on line 3 is where it broke, but the real question is a spec question: what *should* the average of no scores be? Nothing? Zero? An error with a friendly message? The traceback found the edge; your spec has to decide it.

## The Debug Log, used for real

You met the Debug Log in **The Computing Hour**: one page per bug, dated, with **Expected** / **Actual** / **Guess** / **One change** / **Result**. This chapter is where it becomes a habit. Here's the shipping bug as a log page.

*An illustration, not a reported family:* a filled Debug Log page.

**Expected:** `free_shipping(50)` gives True (spec: "50 or more") **Actual:** Assertion-Error on line 7 **Guess:** I used `>` and the spec means "at least" **One change:** `>` to `>=`  
**Result:** all three asserts pass; "all shipping tests passed" printed

Five lines, two minutes. The page does three things for you. It slows you down at the exact moment panic wants to speed you up. It forces one change instead of three. And it leaves a record, so next month when a similar bug appears, you can flip back and find your own past solution.

## Wrong answers you should be able to hear

These are the five you'll hear most, in yourself or in your child. Each has a calm reply.

- **A test that always passes.** If a test can't fail, it proves nothing. Say: "Can I make this test fail on purpose?"
- **Changing three things at once.** If it starts working, you don't know which change mattered; if it gets worse, you don't know which one hurt. Say: "Which one change will I try first?"
- **Reading the top of the traceback and stopping.** The top is the oldest part of the story. Say: "Let's read it from the bottom."
- **"It works on my machine."** Without a written test, "works" means "it printed something once." Say: "Which test shows that?"
- **"I'll just start over."** Rewriting from scratch to escape a bug usually rebuilds the same bug, because the misunderstanding comes with you. Say: "Log what I expected and what happened first."

## A five-minute warm-up

Print the average traceback above, or copy it by hand. With a pencil, number the lines from the bottom: 1 for the error line, 2 for the line above, and so on. Next to line 1 write what happened in your own words. Next to line 2 write where. Next to the top write how the program got there. That's the whole skill of reading red text, and you've just done it on paper.

## How to run it this week

From this week on, every Computing Hour has a test table in it. That's the one new rule, and it slots into the steps you already know:

**Specify.** Write the spec as before, then add a test table under it: two normal rows, one or two boundary rows, one invalid row. If you can't fill the "Expected" column, your spec isn't finished. That's not a failure—that's the table doing its job early.

**Predict.** Before running, mark each row with your prediction: pass or fail. On a first run, most of them should fail, because you haven't written the function yet.

**Build and Test.** Turn each row into an assert. Write the function a little at a time. Run after every few lines. Watch the asserts go from failing to passing.

**Debug and Log.** Any surprise gets a Debug Log page—every one, even the ones you fix in thirty seconds. Five quick pages teach you more than one heroic evening.

**Explain and File.** Say which tests pass, which you'd add next, and what help you used. File the test file next to the program, in both places you keep your work.

## Session 1 (micro, about 15 minutes): Test Table Before Code

Write the table before any code. The card below is the whole session.

### In-home — TEST-01 Test Table Before Code

- **Time:** 15 minutes (micro)
- **Materials:** Notebook, pencil, a spec you wrote in Chapter 1 or 2
- **Safety and privacy:** Made-up data only.
- **The fun:** Catching a hole in your own spec before the computer does
- **The skill:** Normal, boundary, and bad inputs with expected results, written first
- **The crumb:** Test table

**How it goes.** Pick a small spec you've already written—the recipe scaler, a chore picker, a word counter. Draw five rows with four columns: Input, Kind, Expected, Why. Fill two normal rows first; they're easy and they warm you up. Then find the boundary: what number, length, or count changes the answer? Write the row right at it. Last, write one invalid row: an empty list, a negative number, text where a number belongs.

What usually happens is that one row's Expected column won't fill. You'll sit with the pencil over it, and realize you don't know what the program should do when the list is empty. Good. Write the question in the Why column and decide it—that decision goes back into the spec. Date the page. Next session, the table becomes asserts.

## Session 2 (micro, about 20 minutes): Asserts and the boundary

Turn the table into asserts and let the computer check your promises. The worked session below is one version of it, using the shipping rule.

## Session 3 (micro, about 15 minutes): Read the Red Text

BUG-01 Read the Red Text: take one traceback from this week—yours, or the average example above—and annotate it bottom-up on paper: what happened, where, and how the program got there. Then file it with your Debug Log pages.

## The words to say to yourself

When a test fails, the Hour’s opening question still leads: “*Before I change anything, ask anyone, or ask a tool: what did I expect, what actually happened, and what’s the one change I’ll try?*” For this chapter, sharpen it: “Which test failed, and what did it expect?” That question points you at a line number and a value, which is where the bug is. “Why is this broken?” points you at the whole universe.

## Worked session: the free-shipping boundary

*An illustration, not a reported family:* A learner has about twenty minutes, split between a train ride home (as a passenger) and the kitchen table.

**Specify (6 minutes).** In the notebook, on the train:

Free shipping. Input: an order total. Output: True if it ships free. Done when orders of 50 or more ship free, under 50 don’t, and a negative total is an error.

Underneath, the five-row test table from above.

**Predict (1 minute).** At home, before running: “All three asserts pass.”

**Build and Test (5 minutes).** They type the function with > and the first three asserts, exactly as in the shipping example above. Run. AssertionError, line 7.

**Debug and Log (4 minutes).** They don’t touch the code. They read the last line, then the line number, then look at line 7: it’s the 50 row. The guess comes fast once the line is in view.

**Expected:** free\_shipping(50) gives True (spec: “50 or more”) **Actual:** AssertionError on line 7 **Guess:** I used > and the spec means “at least” **One change:** > to >=  
**Result:** all three asserts pass; “all shipping tests passed” printed

There’s time for one more row. In the spec, a negative total is an error, so the function needs to say so:

```
def free_shipping(order_total):
    if order_total < 0:
        raise ValueError("order total can't be negative")
    return order_total >= 50

print(free_shipping(50))    # True
free_shipping(-5)           # stops: ValueError: order total can't be negative
```

The last line of the traceback now reads ValueError: order total can't be negative—a message the learner wrote, in plain words. The red text is on their side now.

**Explain and File (3 minutes).** “I specified free shipping at 50 or more. I predicted all three tests would pass. Test three failed at the boundary; >= fixed it. Next I’d add the 49.99 row as a real assert. Help I used: none.” Test file and program saved in two places.

## Stopping when the timer ends

Stop on a passing test if you can, and on a written log page if you can't. If the timer goes off mid-bug, finish the Expected and Actual lines, write your current guess, and close the laptop. You've left yourself a perfect starting place. A logged stuck session is progress, not a failed evening.

## Practice that actually builds it

Practice comes in two shapes. Blocked practice repeats one move until it's smooth; mixed practice asks you to choose the right move from several. You need both, in that order.

### Blocked: boundaries

Spend two micro sessions writing nothing but boundary rows. Take five rules from daily life—"kids under 12 eat free," "the library allows up to 10 holds," "a password needs at least 15 characters"—and for each, write the boundary value, its two neighbors, and the expected answer. No code. You're training your eye to find the edge.

### Mixed: tests, then traces

Once boundaries feel easy, mix them with tracing. Take a short function, write its test table, then trace it by hand for the boundary row before running anything. If your trace and your table disagree, one of them is wrong—and finding out which is the best kind of practice.

### Diagnose this: the test that can't fail

Here's a test file someone handed in proudly. Every test passes. Find the one that proves nothing.

```
def double(n):  
    return n + n  
  
assert double(2) == 2 + 2  
assert double(0) == double(0)  
print("done")
```

It prints done. The second assert compares the function with itself. It would pass if double returned 0, or 99, or nonsense—it can't fail. The first is weak too: it checks the function against its own arithmetic. Rewrite them as plain answers you worked out by hand: `assert double(0) == 0` and `assert double(-3) == -6`. Both pass with the real function, and both would fail with a broken one. That's the test of a test: could it ever fail?

## Out-of-home — TEST-03 The Test That Can't Fail

- **Time:** 10 minutes (micro)
- **Materials:** The double example above, printed or copied; a pen

- **Safety and privacy:** Made-up data only. Paper only—fine for a waiting room or a library table.
- **The fun:** Playing detective on a test that looks fine
- **The skill:** Spotting tests that prove nothing (always true, testing the test)
- **The crumb:** Annotated example

**How it goes.** Take the printed example somewhere you’re waiting anyway—a practice, an appointment, a library table. For each assert, write one sentence in the margin: “This would fail if the function...” If you can’t finish the sentence, the test can’t fail. Circle it. Then write the replacement assert with a number you worked out yourself.

Most people find the second assert quickly and miss the first. The first one is the sneaky kind: it passes, it looks reasonable, and it uses the same arithmetic as the code, so a mistake in your thinking would appear in both. Write “use a worked-out answer, not the formula again” at the bottom. That’s the crumb.

### In-home — TEST-04 Break It on Purpose

- **Time:** 20 minutes (micro)
- **Materials:** Your shipping file (or any file with passing asserts), your Debug Log
- **Safety and privacy:** Practice folder only.
- **The fun:** Being the one who breaks it, for once
- **The skill:** Changing one line to make a test fail, then restoring it
- **The crumb:** Before/after note

**How it goes.** Open a file whose asserts all pass. Before you touch anything, predict: “If I change `>=` back to `>`, which test will fail?” Write the row number. Make the change. Run. Did the test you named fail—and only that one? Put the line back and run again to watch everything pass.

Then try a second break: change 50 to 5. Predict first. This time the 40 row fails, because 40 now counts as free shipping. You’ve learned which tests guard which lines. If a change breaks nothing, that’s important news: no test is guarding that line. Write a new row for it. The crumb is a two-line note: “Changed X, expected test N to fail, test N failed.”

### In-home — TEST-02 Asserts That Fail Loudly

- **Time:** 30 minutes (desk)
- **Materials:** A test table from TEST-01; your editor
- **Safety and privacy:** Made-up data only.
- **The fun:** Watching the failures turn into passes, one at a time
- **The skill:** An assert for each test-table row; reading a failing assert
- **The crumb:** Test file

**How it goes.** Turn every row of your table into one assert line. Put them under the function, as in the shipping example. Write the function in small steps and run after each. You’ll see the first failing assert stop the file; fix that one, run, and see the next. Resist fixing ahead of the evidence.

One detail helps when there are many asserts: you can add a short message after a comma, and Python prints it when the assert fails—`assert free_shipping(50) == True, "50 should ship free"`. When the file is green, file it beside the program with a dated name. That file is the crumb, and it's the proof you'll show in your portfolio.

## In-home — TEST-05 Someone Else's Spec

- **Time:** 15 minutes (micro)
- **Materials:** A short spec someone else wrote—a partner, a friend, a teen, or a practice exercise from the official Python tutorial
- **Safety and privacy:** Their permission to use it; made-up data only.
- **The fun:** Finding the question the writer didn't think to answer
- **The skill:** Writing tests for a spec you didn't write
- **The crumb:** Dated test table

**How it goes.** Read the spec once without a pencil. Then write the test table. You'll find it harder than writing tests for your own spec, because you can't fill gaps from memory. Every row where you hesitate is a gap in their spec, and it's worth a polite question: "When the list is empty, what should it do?"

If the spec came from your teen, this is one of the most respectful things you can do with their work. You're not grading it; you're using it seriously. Hand back the table with the questions circled.

## In-home — BUG-04 Print to See

- **Time:** 20 minutes (micro)
- **Materials:** The deliberate-bug example below; your editor
- **Safety and privacy:** Practice folder only.
- **The fun:** Watching a variable change in front of you
- **The skill:** Adding prints to watch a variable change, then removing them
- **The crumb:** Before/after file

**How it goes.** Here's a function that should count the words longer than a given length. The bug is deliberate and labeled:

```
def count_long(words, n):
    """Count words longer than n letters."""
    count = 0
    for w in words:
        if len(w) > n:
            count = count + 1
    return count    # DELIBERATE BUG: indented one level too far

print(count_long(["oak", "maple", "birch", "elm"], 3))
```

Predict: "maple" and "birch" are longer than three letters, so it should print 2. It prints 0. Add one line inside the loop, at the top: `print("checking", w, "count is", count)`. Run it. You'll see a single line—checking oak count is 0—and then the answer. The loop ran once. That's your

clue: the function returned during the first trip around the loop. Move `return` count back four spaces so it sits under `for`, `run`, and it prints 2.

Now remove the print. Debugging prints are scaffolding, not part of the building. Save a before and after copy; that's the crumb. Some editors have a debugger that steps through a line at a time and shows every variable—Thonny's is gentle for beginners. Print first, because it's clear; try the debugger when you're curious.<sup>12</sup>

## In-home — BUG-03 Rubber Duck, Out Loud

- **Time:** 10 minutes (micro)
- **Materials:** Any buggy file; any object—a mug, a toy, a patient person
- **Safety and privacy:** Practice folder only.
- **The fun:** Solving the bug in the middle of your own sentence
- **The skill:** Explaining code line by line to an object (or a patient person) until the bug shows
- **The crumb:** One-line note of where it showed

**How it goes.** Put the object on the table. Explain the code to it, line by line, out loud: “This line sets count to zero. This line goes through each word. This line checks if it's long. This line... returns. Oh.” It sounds silly. It works far more often than it should, because saying a line forces you to say what it really does, not what you meant.

Write one line in your log: “It showed at line 7, when I said ‘returns’ out loud inside the loop.” If you're doing this with a teen, you be the duck. Your only job is to listen and say “And then?”

## The rescue order, with tests in hand

This is the chapter where you'll use the rescue order from **The Computing Hour** most, both versions. Use the order for yourself as written there; with tests, its “worked example in a separate scratch file” step becomes especially powerful—a three-line program that reproduces the same failing assert often makes the bug obvious. With a child, use the Hour's child version, and let the failing test be the hint: “Which test does the red text name?” COACH-03 The Rescue Order (in the Hour) is the card for rehearsing it.

## Talk to yourself (this chapter)

**Opening:** “Which test failed, and what did it expect?”

**Follow-ups:** 1. **Prediction** — What did I write in the Expected column before I ran it? 2. **Evidence** — What does the last line of the traceback say, and which line number does it name? 3. **Location** — On that line or the one above it, where do the value I expected and the value I got first part ways? 4. **Change** — What's the one change I'll make, and which test do I expect it to fix? 5. **Revision** — Was the test wrong, or the spec fuzzy, instead of the code?

**How to wait:** Set a five-minute timer. Trace the failing row by hand on paper, one line at a time, writing each variable's value. Read the spec aloud. If the timer rings and you're still lost, stand up, get water, and come back to the log—not the code.

**What a stuck silence usually means:** You’re looking at the wrong line. Go to the line number in the traceback and read it slowly, then the line above it. If it’s a logic error with no traceback, the bug is often in a line you’re sure is fine—go there first.

**Don’t say / say instead:** Don’t say “It’s broken.” Say instead “Test three expected True and got False.”

## Coaching your kids at every age

You don’t need to be ahead of your child to coach debugging. You need to be calmer than the bug. Each age below has one move, the exact words, and one thing to avoid.

### Ages 5–10

**The move:** a bug hunt on paper, with your hands in your lap. Young children’s programs are often paper strips—arrows for “step forward,” “turn left”—or a few blocks on a screen. Plant one mistake in a strip you wrote: an extra turn, a missing step. Hand it over and say, “This robot was supposed to reach the cookie, but it bumped the wall. Can you find the step that went wrong?” Then wait. Let them walk it with a finger or a toy.

**Exact words:** “What did you think it would do? What did it do? Where did they stop matching?” And when they find it: “You found the clue. What would you change?”

**One thing to avoid:** grabbing the pencil. Even a gentle “here, let me show you” teaches that bugs are fixed by grown-ups. Sit on your hands if you need to.

### Out-of-home — BUG-05 Bug Hunt With a Child

- **Time:** 15 minutes (micro)
- **Materials:** Two paper strips of arrow steps (one correct, one with a planted mistake), a toy or coin to move, a grid drawn on paper
- **Safety and privacy:** Paper only; no screens, no photos of the child posted anywhere.
- **The fun:** The child gets to catch the grown-up’s mistake
- **The skill:** Finding a planted mistake in a young child’s paper program without grabbing the pencil
- **The crumb:** Child’s corrected strip

**How it goes.** This one travels well—to a grandparent’s table, a library children’s room, a waiting area with a little space. Draw a small grid with a start square and a goal. Give the child your strip with the planted mistake and the toy. “Let’s run my program.” They move the toy step by step. When it goes wrong, they’ll usually laugh. Ask: “Which step was the bug?” Let them circle it and write the fix in their own way—a new arrow, a scribble, a word.

Keep the strip. Write the date on the back. If you’re doing it with two children, let the older one plant the bug for the younger one next time. The corrected strip is the crumb, and children are often proud to see it saved.

## Ages 11–14

**The move:** Expected and Actual on the log before you help. At this age, kids write real code and meet real tracebacks. When they call you over, don't look at the screen first. Look at the log. "Before I look—what did you expect, and what happened?" Have them write those two lines. Often they find the bug while writing them, and you never needed to look at all.

**Exact words:** "Read me the last line of the red text." · "Which line number does it name?" · "What's the one change you want to try?"

**One thing to avoid:** reading the traceback for them. Even if you can see the answer, the skill they need is reading it themselves. Your job is to point at the bottom line and ask.

## Ages 15–18

**The move:** read the log, not the screen. Teens working on longer projects keep Debug Logs of their own. When you sit with them, ask for the log page, not the laptop. Read one page and write one question on it—just one, in pencil, in the margin: "What made you guess the boundary?" or "Did this test exist before the fix?" That question is your contribution. It respects their ownership and shows them their log is real evidence, worth reading.

If your teen is using it, *Computing for Emerging Adults*, Chapter 3, "Testing and Debugging: Proof Before Pride," runs this skill from the teen's side.

**Exact words:** "Can I read your log from this week?" · "Which bug taught you the most?" · "Show me a test that would have caught it sooner."

**One thing to avoid:** fixing their code over their shoulder. At this age, a fix you make is a fix they didn't learn—and, for their records, help you gave belongs on their log as "Help I used," just like a tool's.

## In-home — COACH-04 Read the Log, Not the Screen

- **Time:** 15 minutes (micro)
- **Materials:** Your teen's Debug Log (with their permission), a pencil
- **Safety and privacy:** Their log, their permission; don't photograph or share it.
- **The fun:** Seeing how your teen thinks
- **The skill:** Reading a teen's Debug Log and asking one question about it
- **The crumb:** Your one question written on the page

**How it goes.** Ask, don't take: "Could I read a page of your log this week?" Pick one page. Read the Expected, Actual, and Guess lines. Then write a single question in the margin, small and in pencil. Hand it back.

What usually happens is that the teen answers the question out loud before they've finished reading it, and you end up having the best five-minute conversation of the week. If you practice on your own log first, you'll know which questions are useful: the ones about guesses and tests, not about style.

## If it isn't clicking

**Tests feel pointless.** If writing tests feels like busywork, you probably haven't seen one catch a bug you'd have missed. Do TEST-04 Break It on Purpose and TEST-03 The Test That Can't Fail back to back. Watching a good test catch your planted bug, and a bad test sleep through it, usually changes your mind in one session.

**Tracebacks overwhelm you.** Read only the last line, then the line number. Ignore everything else for a week. Copy the last line into your log in your own words. After ten tracebacks, the rest of the report will start making sense on its own.

**Debugging spirals.** If you notice you've changed four things and lost track, stop at the timer and log. Put the code back to the last version where you knew what was happening (Chapter 6 will make this painless). Then walk the Hour's rescue order for yourself from the top, and make one change, run, and write the result. The spiral breaks the moment you write.

**When to get a person.** After two logged sessions stuck on the same thing, ask a human: a library coding group, a meetup, a community-college lab, or a friend who programs. Bring the spec, the test table, and the log pages. People who help beginners love a learner who arrives with a log; it makes helping fast. There's no shame in asking. Asking well is a skill too.

## Tools, including AI

The AI box in **The Computing Hour** holds the rules; here's only what's different for testing.

A tool can be very useful for reading a traceback you don't understand. Paste the error—not your private files—and ask what the error type means. Then go back to your own code and find the line yourself.

A tool can suggest test cases you missed. That's a good use: you still decide whether each suggested row matches your spec, and you write the Expected column yourself. Be careful about the opposite use. Asking a tool to “fix the failing test” sometimes returns a changed test instead of a fixed function—a test that can't fail, made by a tool. Developers surveyed about AI tools commonly report that debugging AI-written code takes extra time; Chapter 8 looks at that honestly.

For your editor, a beginner-friendly one with a step-through debugger, such as Thonny, is enough. Python Tutor can draw your program's variables as it runs, which is wonderful for a stubborn trace.<sup>3</sup> Check each tool's current version and terms before you rely on it.

## What “done enough” looks like

You're ready to move on when you have:

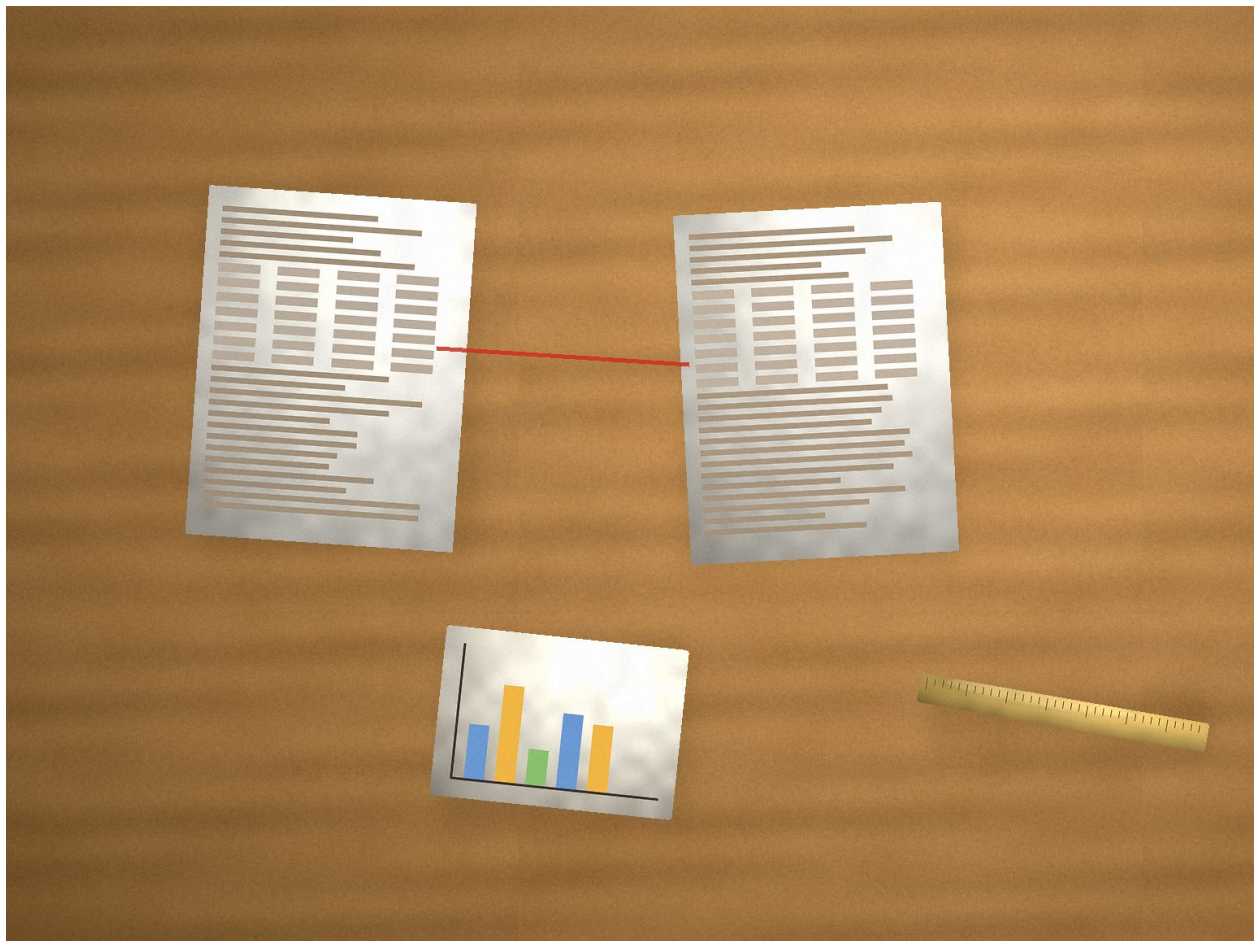
- ☐ A test table written **before** the code, for at least two programs, with normal, boundary, and invalid rows
- ☐ A test file of asserts that fail loudly and now all pass
- ☐ One rewritten test that used to be unable to fail

- ☐ Five dated Debug Log pages, each with one change and its result
- ☐ One traceback annotated bottom-up in your own words
- ☐ **AS-01 Cold Trace** once: trace one tested function by hand, no computer, and match the asserts
- ☐ **AS-02 Debug on the Spot** once: with no tool open, find and fix a planted bug using the traceback and one Debug Log page
- ☐ **AS-04 Explain Your Code** passed once: with no tool open, explain every line of one tested function to a person (or out loud to the duck), including why each test row exists

Placement is by skill, not by how many weeks you've spent here. If you can already do all of this, move on to Chapter 4. If one line is still shaky, give it two micro sessions and check again. "Done enough" means ready for the next chapter—you'll keep testing for as long as you program.

# Chapter 4

## Data: Spreadsheets, Questions, and SQL



### Why this matters

You already work with data. Maybe it's a household budget, a spreadsheet of who's driving which carpool, a club roster, a reading list, or a sheet at work that someone else built and everyone is afraid to touch. Most adults have spent more hours inside spreadsheets than inside any programming language. This chapter starts from that experience and makes it sturdier.

The central idea is simple and easy to forget: **a question comes before a chart**. Data doesn't speak. People ask it questions, and the answer depends on the question, on which rows you kept, and on how carefully you cleaned the file. Start with the chart and you'll find a story that isn't there. Start with the question—written down, with what would change your mind—and the data has a chance to tell you something true.

From there the chapter moves in a straight line. You'll make a table tidy: one row per thing, one column per fact. You'll clean a messy household-style file and keep notes on every change. You'll draw a chart that tells the truth. Then you'll let Python total a column from a CSV file, which is where data meets the programming you learned in Chapter 2. Finally, you'll draw a small database on paper and ask it questions in SQL, the language most databases speak, including the one join that connects two tables.

Why learn code for this when spreadsheets exist? You don't have to choose. A spreadsheet is wonderful for looking, sorting, and trying things. Code is better when you'll repeat the job every month, or when you need a record of every step so someone else—or you, in a year—can check how a number was made. Many careful people do both: explore in a spreadsheet, then write down the steps as code once they know what they want.

There's a household reason, too. Your kids will meet data everywhere: charts in the news, survey results in school, numbers in advertising. An adult who asks "What question is this chart answering?" in a calm voice at the dinner table gives them a habit worth more than any formula.

One honest limit: this is not data science in a chapter. It's the foundation underneath—honest questions, tidy tables, careful cleaning, simple totals, and a first conversation with a database. That foundation is enough for a great many real jobs at home and work, and it's the part people skip.

And one safety rule runs through every page: **practice data is made up or public**. Don't practice on your real bank records, your children's health information, or anyone's personal details. Make a file of invented grocery trips, or download a public dataset from your town or library. If a practice file would embarrass anyone if it leaked, it's the wrong practice file.

## Understand it yourself

### The plain picture

A tidy table has one row per thing and one column per fact. A row is one grocery trip, one library book, one child's practice session. A column is one fact about every row: the date, the amount, the title. If you can say "each row is one \_\_\_\_," your table is probably tidy. If you can't finish that sentence, it isn't.

### The precise picture

Four ideas carry the rest of the chapter.

- **Types.** Every column has a type—what kind of value it holds: whole numbers, decimal numbers, text, dates. A column that mixes "12.50" with "twelve fifty" and "12.50 (cash)"

has no single type, and every total built on it is shaky.

- **Keys.** A **database** is organized tables. A **key** is the column that links them—usually an ID number that names each row exactly once. Names make bad keys: two people can share a name, and one person can spell theirs two ways.
- **Joins.** A **query** is a question asked of a database. A join is a query that connects two tables through a key: “for each loan, find its book.”
- **Aggregation.** Totals, counts, and averages squeeze many rows into one number. They’re powerful, and they hide things. Always know how many rows went into a number.

## Messy to tidy

Here’s a messy table of the kind adults make every day. Picture a family tracking kids’ activities:

| Kid     | Mon       | Tue              | Notes         |
|---------|-----------|------------------|---------------|
| Child A | piano 30  | —                | soccer Sat 60 |
| Child B | soccer 45 | piano 30, art 20 |               |

It’s readable to the person who made it. It’s nearly impossible to total. Days are spread across columns, one cell holds two facts (“piano 30”), another holds two activities, and Saturday is hiding in Notes. Here’s the same information, tidy:

| kid     | day | activity | minutes |
|---------|-----|----------|---------|
| Child A | Mon | piano    | 30      |
| Child A | Sat | soccer   | 60      |
| Child B | Mon | soccer   | 45      |
| Child B | Tue | piano    | 30      |
| Child B | Tue | art      | 20      |

Each row is one activity session. Each column is one fact. Now “total minutes of piano this week” is a sort and a sum.

## Wrong answers you should be able to hear

- **A chart before a question.** Someone opens the data and starts making charts to “see what’s there.” Say: “What question is this chart answering?”
- **A truncated axis.** A bar chart whose vertical axis starts at 90 instead of 0 makes a 2-point difference look like a landslide. Say: “Where does this axis start?”
- **One column holding two facts.** “piano 30” in a single cell. Say: “Can I split this into two columns?”
- **A join with no prediction of row count.** You run a join and accept whatever comes back. Say: “How many rows did I expect, and why?”

- **Averaging percentages.** One club has 1 of 2 members coming (50%), another has 9 of 10 (90%). The average of the two percentages is 70%, but 10 of 12 people overall are coming, about 83%. Say: “What are the counts underneath?”

## A five-minute warm-up

Take any messy table you’ve made—or the activity table above—and redraw it on paper as a tidy table. Write at the top: “Each row is one \_\_\_\_.” Count the columns. If any cell still holds two facts, split it. That’s the whole skill in miniature.

## How to run it this week

This week runs in three steps: a question, a clean file, and a first query. Each fits a single session.

### Session 1 (micro, about 15 minutes): A Question Before a Chart

DATA-01. Specify, for data, means writing the question. On a card, write three things: the question (“Which store did we spend more at this month?”), the population (“our grocery trips, 1–31 October, made-up file”), and what would change your mind (“if one store is more than twice the other, I’ll rethink where we shop”). Predict the answer before you open anything.

### Session 2 (desk, about 40 minutes): Clean a Messy Spreadsheet

Open your practice file and make it tidy, keeping a list of every change. The card is below, after the worked session.

### Session 3 (desk, about 30 minutes): First SELECT

SQL-02 (in Practice). Ask a small practice database three questions, each with a prediction written first.

Each session still uses the Hour’s steps—Specify, Predict, Build and Test, Debug and Log, Explain and File. For data, “test” often means checking a number by hand: total three rows yourself, then confirm that the program or spreadsheet agrees.

## The words to say to yourself

The chapter’s question is “What question am I answering, and what would change my mind?” When the numbers come back, say what they are in careful words: “In this file, for these rows, the total is...” rather than “The data says...” The first sentence is true. The second is a claim about the world.

## Worked session: grocery trips in a CSV

*An illustration, not a reported family:* A learner has a weekend morning—a deep session, about 75 minutes.

**Specify (10 minutes).** A question card: “How much did we spend on groceries in this made-up file, and at which store?” Prediction: “About 50 total; more at the warehouse.”

**Predict (5 minutes).** Before writing code, they make the practice file—three invented trips, saved as `trips.csv`, a plain-text table with commas between the columns:

```
date,store,amount
2026-10-03,Market,12.50
2026-10-07,Warehouse,30.25
2026-10-10,Market,7.25
```

They total it by hand first:  $12.50 + 30.25 + 7.25 = 50.00$ . That hand total is the test.

**Build and Test (30 minutes).** The first program reads each row as a list and adds the third item. It’s labeled as the deliberate bug, because it’s the one the session is built around:

```
import csv
total = 0
with open("trips.csv") as f:
    for row in csv.reader(f): # DELIBERATE BUG: header row read as data
        total = total + float(row[2])
print(round(total, 2))
```

It stops with a traceback. The last line reads `ValueError: could not convert string to float: 'amount'`. The learner reads it bottom-up, as in Chapter 3: the word ‘amount’ is the column heading. The program tried to turn the header row into a number.

### Debug and Log (10 minutes).

**Expected:** 50.0 **Actual:** `ValueError` on the header row **Guess:** the first row is the heading, not a trip **One change:** use `csv.DictReader`, which reads the header as column names **Result:** see below

```
import csv
total = 0
with open("trips.csv") as f:
    for row in csv.DictReader(f):
        total = total + float(row["amount"])
print(round(total, 2))
```

It prints 50.0, matching the hand total. Notice the bonus: `row["amount"]` names the column, so the code reads like the question.

With time left, the learner answers the second half of the question—spending by store—and writes a report file:

```
import csv
totals = {}
```

```

with open("trips.csv") as f:
    for row in csv.DictReader(f):
        store = row["store"]
        totals[store] = totals.get(store, 0) + float(row["amount"])
with open("report.txt", "w") as out:
    for store, amount in sorted(totals.items()):
        out.write(f"{store}: {amount:.2f}\n")
print(open("report.txt").read(), end="")

```

It prints two lines: Market: 19.75 and Warehouse: 30.25. The prediction was right about the warehouse. **Explain and File (10 minutes).** “I specified a total by store. I predicted about 50, more at the warehouse. It did: 50.0, with the warehouse at 30.25. Next I’d add a month column and a test with an empty file. Help I used: the csv module page in the Python docs.”<sup>15</sup>

## In-home — DATA-02 Clean a Messy Spreadsheet

- **Time:** 40 minutes (desk)
- **Materials:** A made-up or de-identified household spreadsheet; a notebook for change notes
- **Safety and privacy:** Made-up data only, or a copy with every name and account number removed. Never clean the original.
- **The fun:** The moment a column you couldn’t total suddenly totals
- **The skill:** Fixing types, blanks, duplicates, and inconsistent labels in a household spreadsheet
- **The crumb:** Before/after file with change notes

**How it goes.** Make a copy first and name it so you’ll know: `budget-practice-clean.csv`. Work in four passes, one problem at a time. **Labels:** sort each text column and look for near-duplicates—“Groceries,” “groceries,” “Grocery.” Pick one spelling. **Types:** in each number column, find anything that isn’t a plain number—“12,50” with a comma where the decimal point belongs, “12.50 cash,” “about 20”—and fix or flag it. **Blanks:** decide for each blank whether it means zero, unknown, or “doesn’t apply,” and write the rule down. **Duplicates:** sort by date and amount and look for rows that appear twice.

What usually happens is that one decision is genuinely hard: is that blank a zero or an unknown? Write the question and your choice in the change notes. Those notes are the crumb, and they’re more valuable than the clean file, because they let anyone—including you next month—see how the numbers were made.

## Stopping when the timer ends

Stop between passes, never in the middle of one. Write the next pass at the top of your change notes: “Next: duplicates.” Save the file with today’s date. A half-cleaned file with good notes is a perfect place to start next time; a half-cleaned file with no notes is a puzzle.

## Practice that actually builds it

### Blocked: SELECT and WHERE

five SELECT queries with WHERE, one session, each predicted. Then do the same with cleaning: one session that fixes only labels, another that fixes only types.

### Mixed: which tool, and which chart?

a session where you pick the tool—spreadsheet, Python, or SQL—for three questions and say why. A one-off look is a spreadsheet. A monthly report is Python. A question across two tables is SQL. There’s no single right answer; what matters is that you can say why. Write the three questions, your tool choices, and one sentence of reasoning for each in your log. Next week, do one of them in a different tool and compare how long it took and how sure you felt about the answer.

A good mixed session to try: take your cleaned practice file and answer the same question twice—once by sorting and summing in the spreadsheet, once with a short Python program. If the two answers match, you’ve tested both. If they don’t, you have a perfect bug to log, and it’s usually a blank, a duplicate, or a label spelled two ways that one method caught and the other didn’t.

### Diagnose this: the truncated axis

Picture a bar chart of two weeks’ reading minutes: 92 and 94. The vertical axis starts at 90, so the second bar looks twice as tall as the first. Nothing in it is false, and the impression is wrong. The fix is one change: start the axis at zero, and the bars look nearly equal—which is the truth. Write that caption in your notebook: “Reading time was about the same both weeks.”

## In-home — DATA-05 Python Reads a CSV

- **Time:** 60 minutes (deep)
- **Materials:** A made-up CSV of at least ten rows (grocery trips, reading minutes, practice sessions); your editor; a calculator
- **Safety and privacy:** Made-up data only. Practice folder only.
- **The fun:** A report that writes itself next month
- **The skill:** Using the csv module to total a column and write a report
- **The crumb:** Program and report

**How it goes.** Repeat the worked session above with your own made-up file, but larger. Start with a question card. Total three rows by hand and write the subtotal; that’s your first test. Write the program with `csv.DictReader`, run it on just those three rows (copy them into a tiny test file), and check it matches. Only then run it on the full file.

Add one invalid case on purpose: a row with a blank amount. Predict what happens. (It stops with a `ValueError`, because an empty string isn’t a number.) Decide what your spec says—skip it, count it as zero, or stop with a clear message—and write that rule in the program and in your

change notes. The crumb is the program, the report file, and the tiny test file side by side. Next month, you'll change one filename and run it again.

### In-home — DATA-03 Tidy Tables on Paper

- **Time:** 20 minutes (micro)
- **Materials:** Paper, pencil, one messy table (yours, or the activity table above)
- **Safety and privacy:** Made-up data only.
- **The fun:** Untangling a table you've lived with for years
- **The skill:** One row per thing, one column per fact; redesigning a messy table
- **The crumb:** Redrawn table

**How it goes.** Write “Each row is one \_\_\_\_” at the top of the page before you draw anything. Fill the blank. Then list the facts you know about each of those things; those become your columns. Redraw the table with three or four sample rows. If a column name has “and” in it, split it.

Most people discover that their messy table was really two tables—a list of kids and a list of sessions, or a list of books and a list of loans. That discovery is the doorway to databases, and it's the first step of SQL-01 below.

### Out-of-home — DATA-01 A Question Before a Chart

- **Time:** 15 minutes (micro)
- **Materials:** A notebook; a public chart—on a library bulletin board, in a town meeting hand-out, in a newspaper
- **Safety and privacy:** Public material only; no photos of people.
- **The fun:** Reading a chart like a detective instead of a believer
- **The skill:** Writing the question, the population, and what would change your mind before opening the data
- **The crumb:** Question card

**How it goes.** Find one public chart while you're out—at the library, a town meeting, a community center. Copy its title. Then, on a card, write the question you think it answers, the population (whose data? which years?), and one thing that would change your mind about its message. Check the axes: where does the vertical one start? Are the labels clear?

What usually happens is that you can't write the population, because the chart doesn't say. That's useful knowledge. Write “population not stated” on the card. Next time, do the card *before* you look at your own data—that's the real habit.

### In-home — DATA-04 Chart That Tells the Truth

- **Time:** 30 minutes (desk)
- **Materials:** Your cleaned practice file; a spreadsheet program or paper and colored pencils
- **Safety and privacy:** Made-up data only.
- **The fun:** Making a chart you'd defend at a family meeting
- **The skill:** Choosing a chart for the question; labeling axes; avoiding a truncated axis

- **The crumb:** Chart with caption

**How it goes.** Start with your question card. Pick the chart for the question: comparing a few categories is usually a bar chart; change over time is usually a line; parts of a whole only works if the parts truly add to the whole. Make it. Then label both axes with names and units, start bar charts at zero, and write a one-sentence caption that says what the chart shows and what it doesn't.

## SQL: asking a database questions

A spreadsheet can hold one table nicely. When information naturally lives in two tables—books and loans, kids and sessions—a database keeps them apart and lets you join them with a key. SQL (often said “sequel” or spelled out) is the language for asking. SQLite is a small, free database that lives in a single file on your computer, which makes it ideal for practice.<sup>16</sup>

### In-home — SQL-01 A Paper Schema

- **Time:** 20 minutes (micro)
- **Materials:** Paper, pencil, a household collection—books, board games, tools you lend
- **Safety and privacy:** Made-up names only (“Reader A”); no real neighbors’ names.
- **The fun:** Designing the catalog for your own shelves
- **The skill:** Two tables and a key drawn on paper for a household collection
- **The crumb:** Schema drawing

**How it goes.** Draw two boxes. The first is **books:** id, title, shelf. The second is **loans:** id, book\_id, borrower, due, returned. Draw a line from loans.book\_id to books.id. That line is the key relationship: each loan points to exactly one book, and one book can have many loans over time.

Fill three sample rows in each box. Then ask the paper two questions: “Which books are out right now?” and “How many times has each book been borrowed?” Trace your finger along the line to answer. If you can answer on paper, SQL is just writing down what your finger did.

### In-home — SQL-02 First SELECT

- **Time:** 30 minutes (desk)
- **Materials:** A practice database built from your schema (four books, five loans, all made up); SQLite
- **Safety and privacy:** Practice folder only. Made-up data only.
- **The fun:** Asking a question and getting exactly the rows you predicted
- **The skill:** SELECT, WHERE, ORDER BY on a small practice database
- **The crumb:** Saved queries file

**How it goes.** Here’s the practice library used in this chapter. The books are Bread at Home and Rivers of Stone (shelf “adult”), and Counting Stars and The Quiet Garden (shelf “kids”). Five loans: one each for Bread at Home, Rivers of Stone (already returned), and The Quiet Garden, and two for Counting Stars. Write the prediction first, then the query:

```
SELECT title, shelf
```

```
FROM books
WHERE shelf = 'kids'
ORDER BY title;
```

Prediction: two rows, alphabetical. Result: Counting Stars | kids and The Quiet Garden | kids. Read it aloud as English: “Show me the title and shelf, from books, where the shelf is kids, in title order.” That’s SQL’s great kindness—it reads close to a sentence.

Write three queries this way, each with a prediction comment above it (a line starting with `--` is a comment in SQL). Save them in a file. That file is the crumb.

## In-home — SQL-03 Join Two Tables

- **Time:** 40 minutes (desk)
- **Materials:** The same practice database; your paper schema
- **Safety and privacy:** Practice folder only. Made-up data only.
- **The fun:** A surprise row count that teaches you something real
- **The skill:** An inner join answering a real question; predicting row count first
- **The crumb:** Query with prediction

**How it goes.** The question: which books are out, and when are they due?

```
SELECT books.title, loans.due
FROM books
JOIN loans ON loans.book_id = books.id
WHERE loans.returned = 0
ORDER BY loans.due;
```

Most people predict three rows: “three books are out.” It returns four. Counting Stars appears twice, due on two different dates, because two readers each borrowed a copy and both loans are open. Nothing is broken. The join produced one row per *open loan*, not one per book. This is the most important lesson about joins: the row count follows the many side. Log it: Expected 3 / Actual 4 / Guess one book has two open loans / One change: none needed; the question was “which loans,” not “which books” / Result: spec rewritten to say which.

Then count, to check. `SELECT COUNT(*) FROM loans WHERE returned = 0;` returns 4. Your hand count and the database agree.

## One step further: counting per book

Once a join makes sense, you can ask “how many times has each book been borrowed?” This needs two new words. `GROUP BY` gathers rows into one group per book, and `COUNT` counts each group. A `LEFT JOIN` keeps books with no loans at all, so nothing quietly disappears:

```
SELECT books.title, COUNT(loans.id) AS times_borrowed
FROM books
LEFT JOIN loans ON loans.book_id = books.id
GROUP BY books.id
ORDER BY books.title;
```

Prediction from your paper schema: Bread at Home 1, Counting Stars 2, Rivers of Stone 1, The Quiet Garden 1. That’s exactly what comes back. If your prediction and the result disagree, trace the surprising book across the schema with your finger before touching the query. Don’t worry if LEFT JOIN feels like a lot today; it’s here so you know the shape exists. A plain JOIN and a careful prediction will carry you a long way.

## In-home — SQL-04 Keep Queries Safe

- **Time:** 15 minutes (micro)
- **Materials:** Notebook; the short example below
- **Safety and privacy:** Practice database only. Never try this on any system you don’t own.
- **The fun:** Understanding one of the most common security mistakes in software
- **The skill:** Why queries built by gluing strings are risky; parameters instead
- **The crumb:** Two-line before/after

**How it goes.** When a program asks a user for a book title and then looks it up, the risky way is to build the query by gluing in whatever the user typed. If someone types something that looks like SQL, it can become part of the query and change what it does. The safe way is to pass the user’s text separately, as a parameter, so the database treats it only as data:

```
import sqlite3
con = sqlite3.connect("lib.db")
cur = con.cursor()
title = input("Title: ")
cur.execute("SELECT id, shelf FROM books WHERE title = ?", (title,))
print(cur.fetchall())
```

Typing Counting Stars prints [(3, 'kids')]. Typing something tricky that’s meant to look like SQL prints []—no book has that odd title, so nothing comes back. The ? is the parameter’s place, and the user’s text fills it safely. Write the two-line before/after in your notebook in words: “Before: glue the typed text into the query. After: a ? and the text passed separately.” Chapter 7 comes back to this habit of mind.<sup>17</sup>

## Talk to yourself (this chapter)

**Opening:** “What question am I answering, and what would change my mind?”

**Follow-ups:** 1. **Prediction** — What number, or how many rows, did I write down before I ran it? 2. **Evidence** — What exactly came back, and how many rows went into it? 3. **Location** — Is the surprise in the data, in the cleaning, or in the question? 4. **Change** — What’s the one change—to the query, the file, or the question—and what do I expect it to do? 5. **Revision** — Did my question card need to change instead?

**How to wait:** Set a five-minute timer. Total three rows by hand. Count the rows with your finger. Read the question card aloud.

**What a stuck silence usually means:** The table isn’t tidy yet—one column is holding two facts, or one table is really two.

**Don't say / say instead:** Don't say "The data says..." before you've checked it. Say instead "In this file, for these rows, ..."

## Coaching your kids at every age

Data is the friendliest computing topic for a family, because children have been sorting and counting since before they could read.

### Ages 5–10

**The move:** sort and tally real objects, then make a picture graph. Dump the sock drawer on the bed. "How should we sort these?" Let the child choose: by color, by owner, by whether they have a partner. Then tally—a mark for each sock—and draw a picture graph with one little sock per mark. Ask a question the graph can answer: "Which color do we have the most of?"

**Exact words:** "What question can our graph answer?" · "How did you decide which pile?" · "Is there a sock that doesn't fit any pile?"

**One thing to avoid:** choosing the categories for them. The child's categories—even "fuzzy" and "not fuzzy"—are real data design. Sorting by a rule they made up is exactly the skill.

### Ages 11–14

**The move:** an honest chart caption. When your child makes a chart—for a science project, a survey of friends, a sports season—ask for a one-sentence caption that says what the chart shows and what it doesn't. Then check the axes together.

**Exact words:** "What question is this chart answering?" · "Where does the axis start?" · "How many people are in this?"

**One thing to avoid:** telling them the chart is misleading. Ask the axis question and let them see it. A child who discovers a truncated axis remembers it; a child who's told forgets.

### Ages 15–18

**The move:** a paper schema review, with one question: "What's the key?" Teens working with databases often jump to typing queries. Ask to see the schema on paper first. Trace one question across the tables with your finger together. Then ask: "What's the key that links these?" and "How many rows will this join return?"

If your teen is using it, *Computing for Emerging Adults*, Chapter 5, "Data: Questions, Tables, and Databases," carries this from the teen's side.

**Exact words:** "Show me the schema before the query." · "What's the key?" · "Write the row count you expect, then run it."

**One thing to avoid:** practice files with real family or friends’ data. Teens often want to use real information because it’s interesting. Help them make a convincing made-up version instead—that’s a skill too.

## If it isn’t clicking

**Charts mislead you, or you’re not sure what to chart.** Go back to the question card. Write the question, the population, and what would change your mind. Then pick the chart for the question. If you can’t write the question, you don’t need a chart yet.

**SQL rows surprise you.** Predict the count every time, on paper, before running. Then count with `COUNT(*)`. A surprise is almost always a join following the “many” side, or a `WHERE` that’s broader or narrower than you meant. Draw the schema and trace one surprising row.

**Data feels personal or uncomfortable.** Switch to made-up data. It’s not a lesser kind of practice; it’s what careful professionals use for teaching and testing. Invent twelve rows in five minutes and keep going.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. If you’ve spent two logged sessions on the same cleaning or query problem, bring the question card, the change notes, and the query to a library group or a friend who works with data. They’ll ask you “what’s the key?” and “how many rows?”—and now you’ll know why.

## Tools, including AI

The AI box in **The Computing Hour** holds the rules; here’s what differs for data.

**Never paste personal or family data into an AI tool.** That includes budgets, health records, grades, and anything with a name on it. At work, the employer’s rules come first; many forbid pasting work data into consumer tools.

A tool can be good at explaining what a SQL query does, or suggesting how to fix a cleaning problem in a made-up sample. It’s often fluent and wrong about the numbers themselves: a tool may “total” a column confidently and incorrectly. Check every number by hand on a few rows, or in code you can explain.

For tools: a spreadsheet program you already have; Python’s built-in `csv` module; and SQLite, which you can use from its own command-line program or from Python’s built-in `sqlite3` module.<sup>16</sup> Check the current version and terms of any tool before relying on it.

## What “done enough” looks like

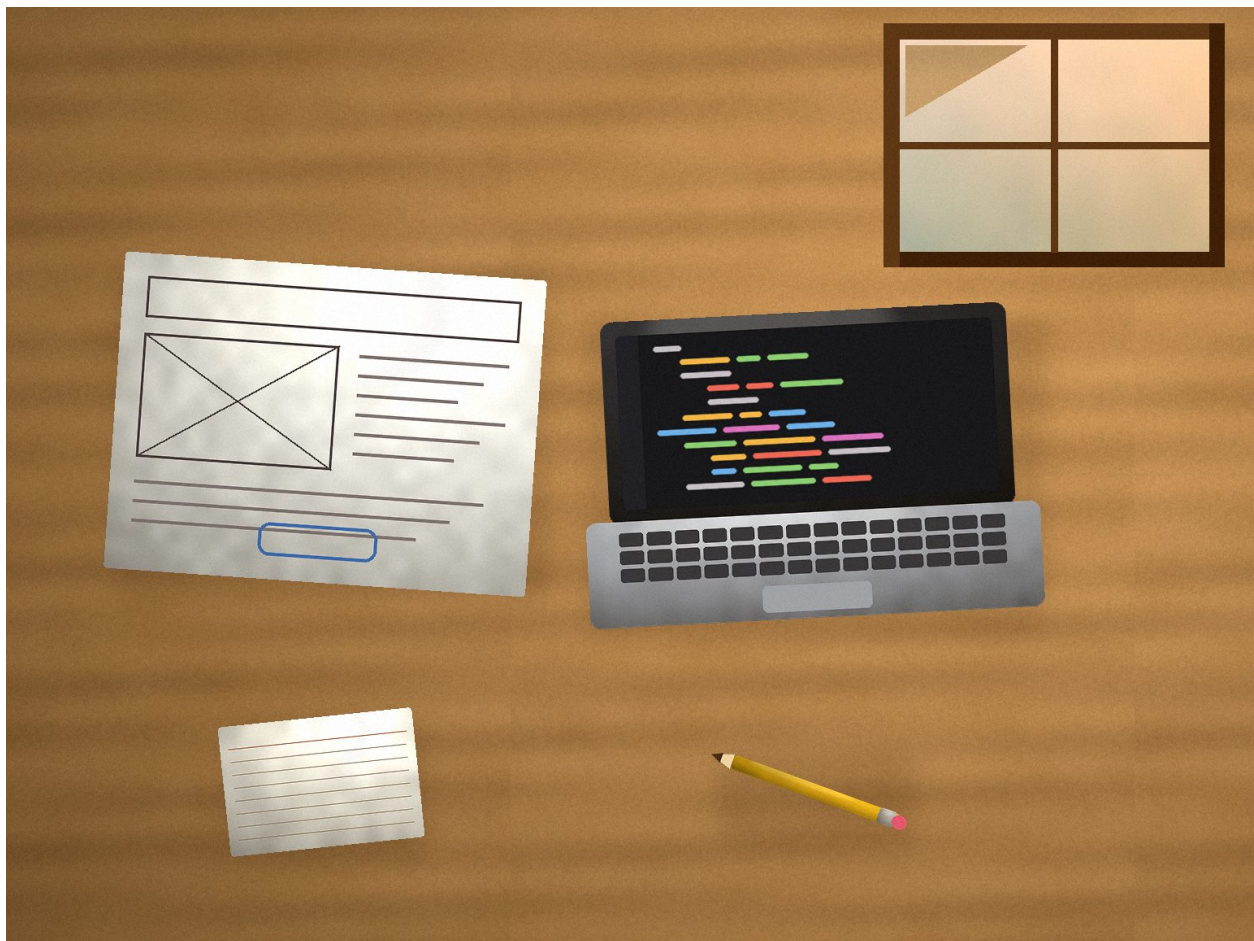
You’re ready to move on when you have:

- ☐ A dated question card (question, population, what would change your mind) written before the data
- ☐ A cleaned practice file with change notes for labels, types, blanks, and duplicates
- ☐ One honest chart with labeled axes, a zero baseline for bars, and a one-sentence caption
- ☐ A Python program that reads a CSV with `csv.DictReader`, matches your hand total, and writes a report
- ☐ A paper schema with two tables and a key
- ☐ Two SQL queries—one SELECT with WHERE, one JOIN—each with a written prediction of the rows
- ☐ **AS-01 Cold Trace** once: trace the CSV-totaling program by hand on three rows, no computer, and match its output
- ☐ **AS-02 Debug on the Spot** once: with no tool open, find and fix a planted bug in a short data program (a header counted as data is a good one)
- ☐ **AS-04 Explain Your Code** once: explain every line of your CSV program and both SQL queries to a person, with no tool open

Placement is by skill. If you already handle spreadsheets well, you may move through the first half quickly; give the SQL half its full time. “Done enough” means ready for the next chapter, not finished with data forever.

# Chapter 5

## Building for People: The Web



### Why this matters

Everything you've built so far has had one user: you. You wrote the spec, ran the program, read the output, and decided whether it was right. A web page changes that. A page is something another person uses—someone who didn't read your spec, doesn't know what you meant, and may be using a keyboard instead of a mouse, a screen reader instead of their eyes, or a small phone in

bright sunlight. The moment another person is the user, how you build changes.

That's the real lesson of this chapter, and it's bigger than the web. Building for people means asking, before you type, "What does a visitor need first?" It means testing with someone who isn't you. It means noticing when "it looks fine to me" is the only evidence you have. Those habits make you a better builder of anything, and a far better first user of your children's projects.

The technical part is pleasantly small. A web page is a plain text file. You can write one by hand, in any text editor, and open it in the browser you already have. No account, no installation, no cost. In a single desk session you can make a page with headings, a list, a link, and an image, and in a deep session you can add a button that actually does something. The building blocks are three languages that work together:

- **HTML** is the outline: headings, paragraphs, lists, links, images.
- **CSS** is the clothes: colors, spacing, fonts, layout.
- **JavaScript** is the light switches: small bits of behavior that respond when someone clicks or types.

You'll meet all three, in that order, and you'll learn to open the browser's console—the place where the page quietly reports its errors.

Why hand-write pages when site builders exist? Site builders are perfectly good for publishing a club newsletter or a family blog. Use one if you need a site this week. But a builder hides exactly the things this chapter teaches: what a heading really is, why an image needs words, how a button knows it was clicked. Hand-written pages teach what the builder hides, and once you understand it, you'll use builders better too.

One honest limit. You won't build a business website this weekend. You'll build a small, true, usable page—something a family member can open, read with a keyboard, and click without confusion. That's a real accomplishment, and it's the foundation everything bigger stands on.

And one rule you'll see again at the end: nothing with family names, faces, school, or town goes public. Building a page and publishing a page are two different decisions. For a minor's work, publishing is always the adult's decision.

## Understand it yourself

### The plain picture

HTML is the outline, CSS the clothes, JavaScript the light switches. Picture a printed recipe card. Its title is big and at the top. Its ingredients are a list. Its steps are numbered. Its photo has a caption. HTML says "this is the title, this is a list, this is a photo"—it describes the *structure*, not the looks. CSS says "make titles dark green and lists a little indented." JavaScript says "when someone presses the 'double it' button, double the amounts."

## The precise picture

- **Elements.** HTML is made of elements written with tags: `<h1>` starts a top-level heading and `</h1>` ends it. `<p>` is a paragraph, `<ul>` an unordered list with `<li>` items, `<a href="...">` a link, `` an image with its alternative text.
- **Selectors and rules.** CSS picks elements with a selector and sets properties in a rule: `h2 { color: #336633; }` means “every second-level heading gets this green.”
- **The cascade, in one sentence.** When two CSS rules disagree about the same element, the more specific one wins, and when they’re equally specific, the later one wins.
- **Event handlers.** JavaScript can listen for an event, like a click, and run a function when it happens.
- **The console.** Every modern browser has developer tools with a console, where JavaScript errors appear in red. A page can look perfect and still be reporting errors there.<sup>18</sup>

Here’s a small, complete page—the start of the family recipes site you’ll build this week:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Our Family Recipes</title>
</head>
<body>
  <h1>Our Family Recipes</h1>
  <p>Three dishes we make every month.</p>
  <h2>Pancakes</h2>
  
  <p><a href="soup.html">Read the soup recipe</a></p>
</body>
</html>
```

Save it as `index.html` and open it in your browser. You’ll see a big heading, a sentence, a smaller heading, the image (or its alt text, if the picture file isn’t there yet), and a link. The `lang="en"` tells browsers and screen readers what language the page is in. The `<title>` is what appears on the browser tab.

## Headings are an outline, not a font size

This is the single most useful idea in the chapter. A heading’s level says where it sits in the outline: `<h1>` is the page’s title, `<h2>` the main sections, `<h3>` the parts inside a section. People who use screen readers often move through a page heading by heading, the way you’d skim a table of contents.<sup>19</sup> If you choose `<h4>` because you liked its size, the outline becomes nonsense. Choose the heading by its place in the outline; change its size with CSS.

## Alt text is the picture, in words

The `alt` attribute on an image is what a screen reader says aloud, and what shows if the image fails to load. Good alt text says what the picture shows *for this page’s purpose*: “A stack of three

pancakes on a white plate.” Not “image,” not “pancakes.jpg,” not “click here.” If an image is pure decoration, an empty `alt=""` tells assistive technology to skip it.

## Files, folders, and links

A small site is just a folder. `index.html` is the front page by convention; `soup.html` sits beside it; images often live in a subfolder called `images`. A link like `<a href="soup.html">` means “the file called `soup.html` in this same folder.” A link like `` means “go into the `images` folder and find `soup.jpg`.” Most broken links and missing pictures on a first site are a folder mismatch: the file is in one place and the link points to another, or the name differs by a single capital letter. When a picture doesn’t appear, read the `src` aloud and look in the folder with your own eyes. Chapter 6 will give you a terminal command for exactly this.

## Pages on a small screen

Many visitors will open your page on a phone. One line in the `<head>` tells phone browsers to use the phone’s real width instead of pretending to be a desktop screen: `<meta name="viewport" content="width=device-width, initial-scale=1">`. Add it to every page. Then test the way a visitor would: narrow your browser window until it’s about as wide as a phone, and read the page top to bottom. Does any text run off the side? Is any button too small to tap? A page of plain headings, paragraphs, and lists usually behaves well on a phone without any extra work—another quiet reward for building structure first.<sup>18</sup>

## Wrong answers you should be able to hear

- **Missing alt text.** “It’s just a picture.” Say: “If you couldn’t see it, what would you need to know?”
- **Headings chosen for size.** “I used `h4` because `h2` was too big.” Say: “Pick the level for the outline; we’ll change the size in CSS.”
- **A link that says “click here.”** Out of context—and screen readers can list links on their own—it means nothing. Say: “If this link were all you heard, would you know where it goes?”
- **Styling everything at once.** Twenty CSS rules typed before one is checked. Say: “One rule, predict, reload.”
- **JavaScript errors ignored because the page “looks fine.”** Say: “Let’s open the console and look.”

## A five-minute warm-up

Take a printed page with a clear structure—a recipe, a library flyer, a school newsletter. With a pencil, label the parts: which line would be `h1`? Which are `h2`? Where’s a list? Which pictures would need alt text, and what would you write? You’ve just done the hardest part of HTML without a computer.

## How to run it this week

This week runs in three sessions: a wireframe, a page, and an accessibility pass.

### Session 1 (micro, about 20 minutes): Wireframe on Paper

WEB-01, plus a user story with an acceptance line (SPEC-04, in Practice). This is your Specify step. The card is below, after the worked session.

### Session 2 (desk, about 40 minutes): Hand-Written HTML Page

WEB-02 (in Practice). Build the page from the wireframe.

### Session 3 (micro, about 20 minutes): Accessibility Pass

WEB-05 (in Practice). Test the page as a visitor with a keyboard only.

## The Hour's steps, for the web

- **Specify:** the wireframe and the user story.
- **Predict:** before you reload, say what will change on screen.
- **Build and Test:** a few lines, save, reload. The test is the acceptance line and the keyboard pass.
- **Debug and Log:** when the page surprises you, open the console first, then log.
- **Explain and File:** say what each part does; save the folder in two places.

## The words to say to yourself

The chapter's question is "What does a visitor need first, and can they reach it with a keyboard?" Ask it at the start, and again when you think you're done.

## Worked session: the family recipes page

*An illustration, not a reported family:* A learner has an evening at the table—a desk session, about 45 minutes.

**Specify (8 minutes).** The wireframe has a title at the top, a one-line welcome, three recipe sections, and links between pages. The user story: "As a relative visiting the page, I want to find the pancake recipe quickly, so that I can make it on Sunday." Acceptance line: "From the top of the page, using only the Tab key, I can reach the pancake section's link within three presses."

**Predict (2 minutes).** Before the first reload: "three headings under the big one."

**Build and Test (20 minutes).** They type the page above, adding two more recipe sections. Reload: three headings, as predicted. They add the image with alt text and check it by renaming the image file for a moment; the alt text appears where the picture was. Good.

Then the keyboard pass. Mouse pushed aside. Press Tab: the focus outline jumps to the first link. Tab again: the second. But the “Soup recipe” link at the bottom never highlights. **Debug and Log (10 minutes)**. A look at the code shows why: the learner had copied a styled `<span>` with a click handler instead of an `<a href>`. It looks like a link and works with a mouse, but a keyboard can’t reach it.

**Expected:** the Tab key reaches “Soup recipe” **Actual:** it’s skipped **Guess:** it isn’t really a link; it’s a `<span>` with a click handler **One change:** replace it with `<a href="soup.html">Read the soup recipe</a>` **Result:** Tab reaches it; Enter opens it; the acceptance line passes

**Explain and File (5 minutes)**. “I specified a recipe page a relative could use with a keyboard. I predicted three sections and a reachable soup link. The link wasn’t reachable—it was a span. Next I’d write better alt text for the soup photo. Help I used: none.”

## In-home — WEB-01 Wireframe on Paper

- **Time:** 20 minutes (micro)
- **Materials:** Paper, pencil, a ruler if you like straight lines
- **Safety and privacy:** No real names, faces, school, or town on the page—even in the sketch.
- **The fun:** Designing a whole site in the time it takes the kettle to boil twice
- **The skill:** Sketching a one-page site: what the visitor needs first
- **The crumb:** Wireframe

**How it goes.** Draw a tall rectangle: that’s the screen. Before drawing anything inside it, write on the side: “The visitor is, *and they came here to* .” Then draw boxes, top to bottom, in the order the visitor needs them. Label each box with what it is—“title,” “one-line welcome,” “pancake section,” “link to soup”—not with content. Use an X in a box for an image, and write its alt text underneath.

What usually happens is that the first sketch puts the thing *you* care most about at the top, not the thing the visitor needs. That’s normal. Look at your sentence on the side, and move one box. Date the page. Tomorrow, the wireframe becomes HTML.

## Stopping when the timer ends

Stop with the page saving and opening cleanly, even if half the sections are empty. Leave a comment at the end of the file—`<!-- next: soup section, alt text -->`—which the browser ignores but you won’t. File the folder in two places.

## Diagnose this: the link that works only on your computer

Here’s an incorrect example. A learner builds a two-page recipe site, tests every link on her own laptop, and sends the folder to her sister. On the sister’s computer, the pancake picture is missing and the link to the soup page goes nowhere. On the learner’s laptop, everything still works.

Before you read on, guess what’s different. The page contains this:

```
<a href="file:///C:/Users/home/recipes/soup.html">Soup</a>

```

Those are *absolute* paths: they name one exact place on one exact computer. The sister's computer has no such folder, so the browser has nothing to open. The fix is a *relative* path, which describes where the file is compared with the page that links to it:

```
<a href="soup.html">Soup</a>

```

Now the links work wherever the folder goes, as long as the files keep their places inside it. Two habits prevent this bug: keep every file for a site inside one project folder, and test the site by opening it from a copy of that folder somewhere else, such as a USB drive. If it works from the copy, it will work for your reader. Write the bug in your Debug Log the first time it happens to you; nearly everyone meets it once.

## Practice that actually builds it

### Blocked: one CSS rule at a time

Blocked practice here is just that: one rule, predict, reload, ten times in a row, until predictions are mostly right. Use WEB-03 below as the frame.

### Mixed: wireframe, then build

Mixed practice is a fresh wireframe built into HTML and styled in one session, choosing which elements and rules you need without a list. Start from a new user story, not an old page, so you have to decide which elements fit.

### Diagnose this: three small barriers

Here's a section someone wrote for their recipe site. Find three things a visitor might struggle with.

```
<h4>Soup</h4>

<p>For the full recipe, <a href="soup.html">click here</a>.</p>
```

First, <h4> was chosen for its size; in the outline, this is a main section and should be <h2>. Second, the image has no alt text, so a screen reader may announce only the file name. Third, “click here” tells a visitor nothing out of context. A fixed version: <h2>Soup</h2>, , and <a href="soup.html">Read the full soup recipe</a>.

## In-home — SPEC-04 User Story and Acceptance Line

- **Time:** 15 minutes (micro)
- **Materials:** Index cards

- **Safety and privacy:** Made-up visitors only.
- **The fun:** Seeing your page through someone else's eyes
- **The skill:** “As a ..., I want ..., so that ...” with a checkable acceptance line
- **The crumb:** Story card

**How it goes.** Write three cards, one for each kind of visitor: a relative, someone using only a keyboard, someone on a small phone. Each card follows the shape: “As a , *I want* , so that \_\_\_\_.” Under each, write one acceptance line that you could check in under a minute: “Using only Tab and Enter, I reach the soup recipe.” “On a narrow window, no text runs off the side.”

The hard part is making the acceptance line checkable. “The page is easy to use” isn’t. “I find the pancake recipe within three Tab presses” is. If a line can’t be checked, it’s a wish, not a test.

## In-home — WEB-02 Hand-Written HTML Page

- **Time:** 40 minutes (desk)
- **Materials:** A plain text editor, your browser, your wireframe, an optional image you made (a drawing of a pancake works)
- **Safety and privacy:** Practice folder only. No family photos, names, school, or town.
- **The fun:** Opening a file you typed and seeing a real web page
- **The skill:** Headings, paragraphs, a list, a link, and an image with alt text in a plain text file
- **The crumb:** HTML file

**How it goes.** Make a folder called `recipes-site`. In it, create `index.html` and type the page from “Understand it yourself,” one element at a time, reloading after each. Add a list of ingredients with `<ul>` and `<li>`. Add a second page, `soup.html`, and make sure the link reaches it.

Predict before every reload. Most mistakes at this stage are small and visible: a missing `>` swallows the next line, or a forgotten `</h2>` makes everything after it big. When that happens, look for the last place the page looked right, and read the line after it. Your file is the crumb. If you made a drawing for the image, take a photo of the drawing—not of a person—and use that.

## In-home — WEB-03 CSS One Rule at a Time

- **Time:** 30 minutes (desk)
- **Materials:** Your recipe page; a new file called `style.css`
- **Safety and privacy:** Practice folder only.
- **The fun:** Changing the whole look of a page with one line
- **The skill:** Changing one style rule at a time and predicting the result
- **The crumb:** Stylesheet with comments

**How it goes.** Link the stylesheet by adding `<link rel="stylesheet" href="style.css">` inside the page's `<head>`. Then add one rule, predict, and reload:

```
body { font-family: Georgia, serif; }
p { color: #555555; }
.note { color: #222222; }
h2 { border-bottom: 2px solid #336633; }
```

Add them one at a time. The first changes every font on the page. The second turns every paragraph a medium gray. The third needs a matching class in the HTML—`<p class="note">`—and makes just that paragraph darker, because a class selector is more specific than a plain `p`. That’s the cascade, in practice. The fourth draws a thin green line under each section heading.

Put a short comment above each rule—`/* section headings get a green underline */`—so the stylesheet explains itself. That commented file is the crumb.

## In-home — WEB-04 A Button That Does Something

- **Time:** 60 minutes (deep)
- **Materials:** Your editor and browser; the browser’s developer tools
- **Safety and privacy:** Practice folder only.
- **The fun:** A page that answers when you click
- **The skill:** A small JavaScript function triggered by a button; reading the browser console
- **The crumb:** Page with script

**How it goes.** Make a new practice page with this inside its `<body>`:

```
<button id="go">Double it</button>
<p id="out">2</p>
<script>
  document.getElementById("go").addEventListener("click", function () {
    const out = document.getElementById("out");
    out.textContent = Number(out.textContent) * 2;
  });
</script>
```

Predict: each click doubles the number—2, 4, 8. Click twice; the paragraph shows 8. The script finds the button by its `id`, listens for a click, and when one comes, reads the number, doubles it, and writes it back.

Now break it on purpose, as you did with tests in Chapter 3. Change `getElementById("out")` to `getElementById("Out")`—a capital O—and save. This is a deliberate bug. Reload and click. Nothing happens. The page looks fine. Open the developer tools (often F12, or right-click and choose Inspect) and click the Console tab. In red you’ll see an error. In one widely used browser it reads `Uncaught TypeError: Cannot read properties of null (reading 'textContent')`; other browsers word it differently.<sup>18</sup> It means “I looked for something called Out, found nothing, and then tried to read its text.” IDs are case-sensitive. Debug Log, one change, reload, click: 4.

This is the habit the whole chapter wants: when the page does nothing, the console is where it tells you why.

## In-home — WEB-05 Accessibility Pass

- **Time:** 20 minutes (micro)
- **Materials:** Your own page; a checklist on paper
- **Safety and privacy:** Your own page only.

- **The fun:** Using your page the way some of your visitors do
- **The skill:** Keyboard-only navigation, alt text, and contrast checked on your own page
- **The crumb:** Checklist

**How it goes.** Push the mouse away. Using only Tab (forward), Shift+Tab (back), and Enter, try to do everything a visitor would do. Can you see where you are on the page at each press? Can you reach every link and button? Write each problem on your checklist. Then check every image: does it have alt text that says what it shows? Then check contrast: is light gray text on a white background readable in bright light, or for tired eyes? If you have to squint, make it darker.

This is plain good practice, not a formal audit. There’s a detailed international guideline for web accessibility, and if you build pages for others regularly, it’s worth learning.<sup>19</sup> For your own pages, these three checks—keyboard, alt text, readable contrast—catch a large share of the problems that keep people out.

## Out-of-home — COACH-05 Be the First User

- **Time:** 20 minutes (micro)
- **Materials:** Someone else’s page or project—your teen’s, a friend’s, a co-op class’s (with permission); a notebook
- **Safety and privacy:** With the builder’s permission; no photos of people or screens with names.
- **The fun:** Being genuinely useful to a builder
- **The skill:** Using a teen’s project as a real user and reporting what confused you
- **The crumb:** User notes

**How it goes.** This one works at a co-op, a library teen night, or your own child’s table. Ask the builder: “What’s it for, and who’s it for?” Then use it, out loud: “I’m looking for... I think this is... I clicked that and expected...” Don’t fix anything. Don’t suggest features. Just report what you did, what you expected, and what happened—Expected and Actual, the same shape as a Debug Log.

Write your notes as questions and observations, not grades: “I couldn’t tell where the contact link was.” “Tab skipped the menu.” Hand them over. What usually happens is that the builder says “Oh!” at least once. That’s the sound of the most useful testing there is: a real person who isn’t the builder.

## Talk to yourself (this chapter)

**Opening:** “What does a visitor need first, and can they reach it with a keyboard?”

**Follow-ups:** 1. **Prediction** — What did I expect to see on screen when I reloaded? 2. **Evidence** — What does the page show, and what does the console say? 3. **Location** — What’s the last line where the page still looked right? 4. **Change** — What’s the one change I’ll make, and what should change on screen? 5. **Revision** — Did my wireframe or user story need to change instead?

**How to wait:** Set a five-minute timer. Read the HTML aloud, element by element, checking that each one that opens also closes. Open the console. Tab through the page.

**What a stuck silence usually means:** A quiet error in the console. Open it before anything else.

**Don't say / say instead:** Don't say "It looks fine." Say instead "Let me open the console and tab through it."

## Coaching your kids at every age

The web is a wonderful place to coach, because a page is something your child can show other people—and you can be its first user.

### Ages 5–10

**The move:** draw a page on paper together. Young children don't need to type HTML. Fold a sheet of paper and say, "Let's make a page about your favorite animal. What should go at the top?" Let them decide. Draw boxes for a title, a picture, and three facts. Ask what words someone who couldn't see the picture would need. That's alt text, without the name.

**Exact words:** "What goes at the top?" · "If your friend couldn't see the picture, what would you tell them?" · "Which part would you look at first?"

**One thing to avoid:** taking over the design to make it "look right." A crooked page in their order teaches more than a neat one in yours.

### Ages 11–14

**The move:** CSS one rule at a time, with a prediction first. When your child is styling a page, ask them to say what will change before they reload. Make it a game: you each predict, then reload together. Keep a tally if they like.

**Exact words:** "What do you think will change?" · "Was that what you expected?" · "Which rule did that?"

**One thing to avoid:** letting twenty rules pile up before checking. If a page suddenly looks wrong, help them remove the last rule and reload, rather than diagnosing all twenty at once.

### Ages 15–18

**The move:** be the first user, and write down what confused you. Use COACH-05 above with your teen's project. Take it seriously as a real visitor. Your notes are a gift: "I expected the menu to work with Tab, and it didn't."

If your teen is using it, *Computing for Emerging Adults*, Chapter 6, "Building for People: The Web, the Command Line, and Git," covers this work from the teen's side.

**Exact words:** "Who's it for, and what should they do first?" · "Can I try it with just the keyboard?" · "Here's where I got stuck—what did you mean to happen?"

**One thing to avoid:** publishing their work for them without talking it through, or letting it go public with their name, face, school, or town. **The publishing rule:** building and publishing are separate decisions. Nothing with family names, faces, school, or town goes public, and for a minor's work, publishing is the adult's decision. Many teens are happy to share a site with family by sending the folder, or by keeping it on a private address, until they're older.

## If it isn't clicking

**CSS feels chaotic.** Things change that you didn't touch, or nothing changes. Go back to one rule at a time. Comment out all but one rule (put `/*` before and `*/` after), reload, and add them back one by one. The first rule that surprises you is the one to read.

**JavaScript does nothing.** Open the console and read the red text, bottom line first, as with Python tracebacks. Most first errors are a misspelled ID, a missing bracket, or a script that runs before the element it's looking for exists. The console names the line.

**The page is unusable by keyboard.** Fix headings and links before styling. Replace anything clickable that isn't a real `<a>` or `<button>`. Then tab through again. Structure first, looks second.

**The page looks different on another computer.** Fonts, screen sizes, and browsers differ. That's normal, and it's why a page built from structure—real headings, plain paragraphs, lists—holds up better than one positioned pixel by pixel. Check on one other device, write what differed, and fix only what stops a visitor from reading or reaching something.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. After two logged sessions on the same problem, bring your wireframe, your user story, and your file to a library coding group or a friend who builds pages. Show them the console. They'll be glad you brought it.

## Tools, including AI

The AI box in **The Computing Hour** holds the rules. For the web, two things differ.

A tool can quickly generate a whole page that looks polished—and often has missing alt text, headings chosen for size, or unlabeled buttons. Treat any generated page as someone else's draft: made by a tool, not your work until you can explain every line and it passes your keyboard check. It's usually more useful to ask a tool to explain one CSS rule or one console error than to ask for a page.

For tools, you need only a plain text editor and a browser. Many editors color your HTML as you type, which helps you spot a missing bracket. Your browser's developer tools include the console and an inspector that shows which CSS rule applies to each element. Check your browser's current help pages for how to open them, because menus change.

## What “done enough” looks like

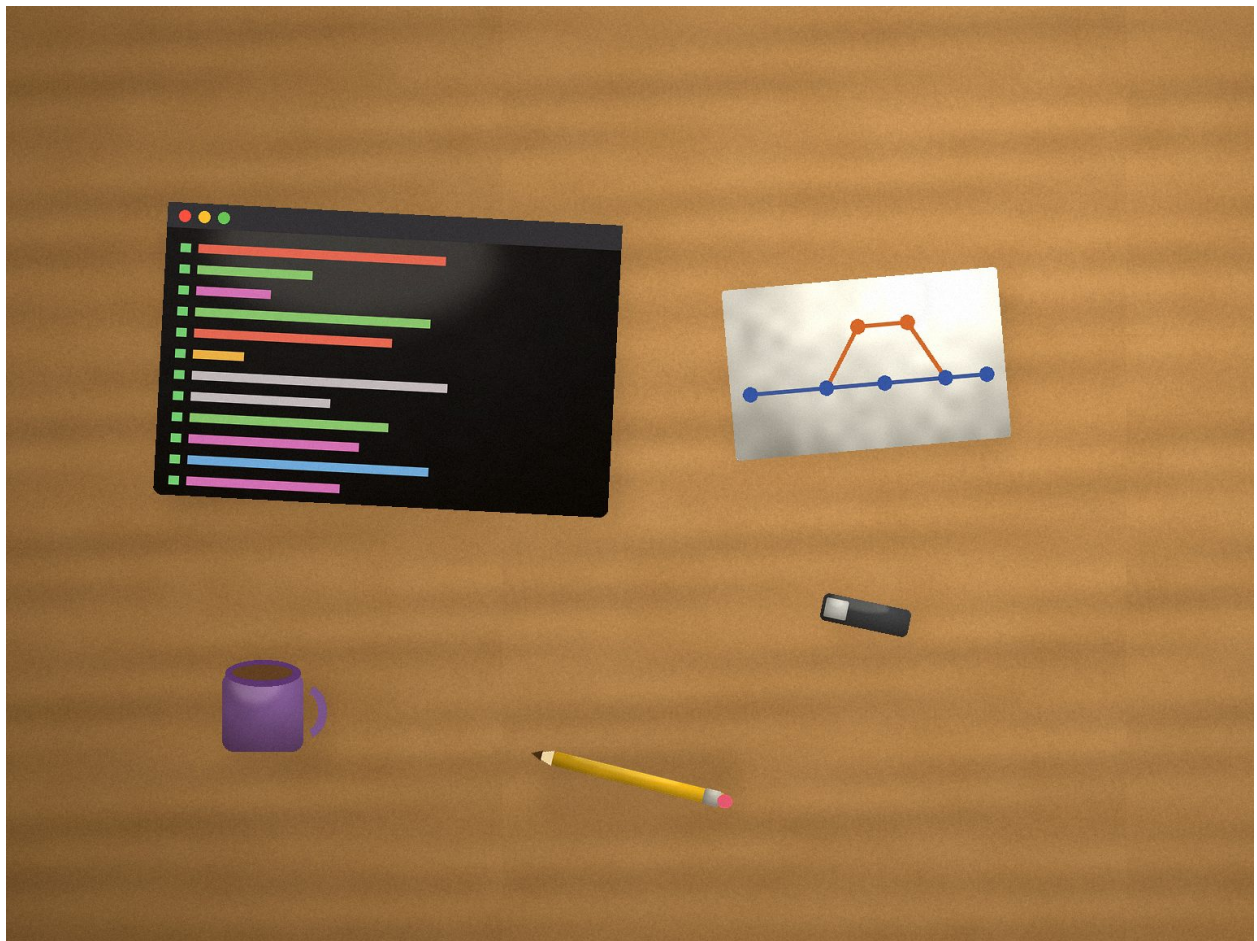
You're ready to move on when you have:

- ☐ A dated wireframe and at least one user story with a checkable acceptance line
- ☐ A hand-written page with headings in outline order, a list, a link with meaningful text, and an image with alt text
- ☐ A stylesheet built one commented rule at a time
- ☐ One working button, and one console error you found, logged, and fixed
- ☐ A keyboard-only pass on your own page with every link and button reachable
- ☐ One first-user session with notes (yours, a partner's, or your teen's project)
- ☐ **AS-01 Cold Trace** once: on paper, predict what the button page shows after three clicks, and why
- ☐ **AS-02 Debug on the Spot** once: find a planted console error (a misspelled ID) with no tool open
- ☐ **AS-04 Explain Your Code** once: with no tool open, explain every line of your page and its button to a person

Placement by skill: if you've built pages before, move quickly through HTML and spend your time on the keyboard pass and the first-user session. "Done enough" means ready for the next chapter.

# Chapter 6

## The Command Line and Git: Save Points and Grown-Up Tools



### Why this matters

Two tools in computing frighten adults more than any programming language: the black window where you type commands, and Git, the tool programmers use to keep the history of their work. Both look like something from a film about hackers. Both are kinder than they look. This chapter is about making them ordinary.

Start with why they're worth your evenings. The **terminal** (or **command line**) is a way of typing commands to the computer instead of clicking. It sounds like a step backward, and in one sense it is: it's older than the mouse. But it's exact. "Move this file into that folder" is a single line you can read, repeat, and write down in your log. When something goes wrong, the line is right there to look at. You've already seen this in a smaller way: running a Python program and reading its output is a conversation in text, and the terminal is simply that conversation with your folders.

**Git** is a tool that keeps the history of every saved change. Each saved change is a **commit**, with a short message saying what changed and why. Think of commits as labeled photographs of your work. If you break something next Tuesday, you can look at last Thursday's photograph and bring back exactly what you had. That's why the chapter's subtitle says *save points*: Git turns "I've ruined it" into "Let me go back to the last good version."

Here's the honest reason for an adult with limited time: these tools make you calmer, not more technical. A learner with a clean folder, a terminal they can use for ten commands, and a Git history of their projects is a learner who doesn't lose work and doesn't panic. It's also the learner whose portfolio, in Chapter 9, tells a true story: every commit is a dated record of what you did. If your teen is building a transcript, a commit history is some of the best evidence there is that the work is theirs.

And if you're coaching kids, this chapter changes what you can see. You won't need to look over a shoulder; you can read a history. "Show me your last three commits" is one of the most respectful questions a parent can ask a teenager who programs, because it asks about the work, not the screen time.

Two limits, said plainly. First, Git doesn't mean you'll never lose work again. It protects what you commit, in the folder you committed it in; it isn't a backup on its own (Chapter 7 covers backups). Second, the terminal can do real damage if you type a command you don't understand in the wrong folder. That's why everything in this chapter happens in a **practice folder** you made for the purpose, and why one rule runs through every page: **never paste a command from the internet that you can't explain.**

## Understand it yourself

### The plain picture

Folders are a family tree. At the top is one starting place. Inside it are folders, and inside those are more folders and files, branching down like generations. When you work in the terminal, you are always standing in exactly one folder—one spot on the tree—and every command you type happens there unless you say otherwise.

Commits are labeled photographs of your work. Each one shows every file in the project as it was at that moment, with a note on the back: "Added rounding to the recipe scaler." Lined up in order, the photographs make a history line you can walk back along.

## The precise picture

- **Path.** A path is a file’s address on the tree: `/home/you/learning/practice/recipes/scaler.py`. Each `/` steps down one level. On Windows, paths start with a drive letter and often use `\` instead.
- **Working directory.** The folder you’re standing in right now. `pwd` (print working directory) tells you where that is.
- **Relative and absolute paths.** An absolute path starts at the top of the tree. A relative path starts from where you stand: `recipes/scaler.py` means “the recipes folder here, then `scaler.py` inside it.” Two dots, `..`, mean “one folder up.”
- **Repository.** A folder whose history Git is keeping. `git init` turns an ordinary folder into one.
- **Staging.** Before a commit, you choose which changes go in the photograph with `git add`. That choosing step is called staging.
- **Commit and log.** `git commit -m "message"` takes the photograph. `git log` shows the history line.
- **Restore.** `git restore` brings a file back to how it was in the last commit, throwing away the changes you haven’t committed.

## Ten commands, in one place

These work in the terminal on Mac and Linux, and in Git Bash on Windows; Windows PowerShell understands most of them too, though a few behave differently.<sup>20</sup>

| Command                            | What it does                                                                                           |
|------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>pwd</code>                   | Show the folder you’re in                                                                              |
| <code>ls</code>                    | List what’s in this folder                                                                             |
| <code>cd recipes</code>            | Step into the recipes folder                                                                           |
| <code>cd ..</code>                 | Step up one folder                                                                                     |
| <code>mkdir recipes</code>         | Make a new folder called recipes                                                                       |
| <code>cp a.py b.py</code>          | Copy a file                                                                                            |
| <code>mv b.py old.py</code>        | Move or rename a file                                                                                  |
| <code>rm old.py</code>             | Delete a file— <b>immediately, with no trash can</b>                                                   |
| <code>cat scaler.py</code>         | Show a file’s contents                                                                                 |
| <code>python3 scaler.py 2 6</code> | Run a Python program with two inputs (on some Windows setups, <code>python</code> or <code>py</code> ) |

Look at `rm` again. Deleting from the terminal usually skips the trash. There’s no “are you sure?” and no undo. That’s not a reason to fear the terminal; it’s a reason to practice in a folder that holds nothing you’d miss.

## Wrong answers you should be able to hear

- **“stuff” as a commit message.** It’ll mean nothing to anyone, including you next month. Say: “What would a stranger need to know about this change?”
- **Committing a password.** Once a secret is in a commit, it’s in the history, even if you delete it later. Say: “Let’s keep that file out of the repository before we add anything.”
- **Running a command in the wrong folder.** The command was right; the place was wrong. Say: “Let’s type `pwd` first and read where we are.”
- **A repository with one giant commit.** A month of work in a single photograph can’t be walked back. Say: “Commit after each test passes—small save points.”
- **Pasting a command you found online.** It might do what it promises. It might not. Say: “Can I explain every part of this before I run it?”

## A five-minute warm-up

On paper, draw your learning folder as a tree. Put your home folder at the top, then `learning` under it, then whatever’s inside. Draw a star on the folder where you’d want to practice. Then write, next to the star, the path you’d type to get there from the top. You’ve done the command line’s hardest idea before opening the terminal.

## How to run it this week

### Session 1 (micro, about 15 minutes): The command line on paper

Draw the tree, then plan your first commands on paper before typing them. The card is below.

### In-home — TERM-01 The Command Line on Paper

- **Time:** 15 minutes (micro)
- **Materials:** Paper, pencil, your warm-up drawing
- **Safety and privacy:** Paper only.
- **The fun:** Walking through folders with a pencil like a map
- **The skill:** Folders as a tree; `pwd`, `ls`, `cd`, `mkdir` drawn before typed
- **The crumb:** Tree drawing

**How it goes.** Start from your warm-up tree. Put your pencil on the home folder; that’s where a new terminal usually starts. Write the commands you’d type, one per line, and move the pencil with each: `cd learning` (pencil moves down), `mkdir practice` (draw a new branch), `cd practice` (pencil moves to it), `mkdir recipes` (another branch). After each line, write what `pwd` would say.

What usually happens is that someone writes `cd recipes` while the pencil is still in `learning`, before `practice`. The paper shows the mistake at once: there’s no `recipes` branch under the pencil. That’s the “wrong folder” bug, caught for free. Date the drawing; you’ll type it next session.

## Session 2 (desk, about 30 minutes): Ten commands, your own folder

TERM-02 Ten Commands, Your Own Folder (in Practice). Type the paper plan from Session 1, then work through the ten-command table in the practice folder only.

## Session 3 (deep, about 60 minutes): A first repository

GIT-02 First Repository (in Practice). Turn the recipe scaler from Chapter 2 into a repository with three honest commits. The worked session below is one version of it.

**Self-talk for this chapter:** “Which folder am I in, and what will this command change?” Ask it before every command that changes something—`mkdir`, `mv`, `rm`, `git commit`, `git restore`.

## Worked session: the recipe scaler gets a history

*An illustration, not a reported family:* A learner has a Saturday morning—a deep session, about 60 minutes—and the recipe scaler from Chapter 2, rewritten to take its two numbers from the command line.

**Specify (5 minutes).** On a card: “Put the scaler under Git. Done when the log shows three commits a stranger could read, and I’ve broken and restored the file once on purpose.”

**Predict (3 minutes).** “After three commits, `git log --oneline` will show three lines, newest first.”

**Build and Test (35 minutes).** In the terminal, the learner checks where they are first:

```
$ pwd
/home/you/learning/practice/recipes
$ ls
scaler.py
$ python3 scaler.py 2 6
3.0
```

The program works: 2 cups for 4 servings becomes 3.0 for 6. (The \$ stands for the prompt; you don’t type it.) Before Git’s first use on a computer, it needs to know whose name goes on the commits—a one-time setup with `git config --global user.name "Your Name"` and the same for `user.email`. Then:

```
$ git init
$ git add scaler.py
$ git commit -m "Scale a recipe from 4 servings to any number"
$ git log --oneline
87137ab Scale a recipe from 4 servings to any number
```

Git also prints a line or two after `init` and `commit` (“Initialized empty Git repository in...”, a summary of the files changed); we’ve left those out. The seven characters at the start of the log line are the commit’s short ID; yours will differ. The learner adds a README file saying what the scaler does, commits it, then changes the print line to round to two decimal places, runs `python3 scaler.py 2 3` (it prints 1.5), and commits that too.

**Debug and Log (10 minutes).** Now the planned break. The learner deletes the print line by accident-on-purpose and runs the program. Nothing prints at all—a quiet bug, the worst kind. `git status --short` shows `M scaler.py`: the file is modified since the last commit.

**Expected:** `python3 scaler.py 2 6` prints 3.0 **Actual:** nothing prints **Guess:** I deleted the print line; the file is different from the last commit **One change:** `git restore scaler.py` **Result:** `git status` says “nothing to commit, working tree clean”; the program prints 3.0 again

**Explain and File (7 minutes).** `git log --oneline` shows three lines, newest first:

```
bb60ce5 Round scaled amounts to two decimal places
a18023f Add a README that says what the scaler does
87137ab Scale a recipe from 4 servings to any number
```

“I specified a repository with three readable commits and one restore. I predicted three lines in the log, and that’s what I got. When I broke the file, `git restore` brought back the last commit. Next I’d try looking at an old version with `git show`. Help I used: the Git documentation for restore.” The learning folder, `.git` and all, is copied to the second place as usual.

## Stopping when the timer ends

Stop on a commit. If the timer goes off mid-change and the tests don’t pass yet, don’t commit broken work just to have a save point; write a note at the top of the file—`# NEXT: rounding passes for 2 and 3, fails for 0`—and stop. Run `git status` before closing the terminal so you know what’s waiting. Next session, `git status` and `git log --oneline` tell you exactly where you left off.

## Practice that actually builds it

### Blocked: ten commands

Spend two sessions doing nothing but the ten commands in your practice folder. Make three folders, copy a file into each, rename one, list each folder, step up and down, and delete a copy you made for the purpose. Type `pwd` before every command that changes anything. Keep a command log—each command on one line, with a note of what you expected—and you’ll find you stop needing the table.

### Mixed: commit after each passing test

Once the commands are smooth, mix Git into your ordinary sessions. Every time a test from Chapter 3 goes from failing to passing, commit, with a message that says what now works. A session might have two commits; a deep session might have four. Then, once a week, read your own log aloud. You’re practicing the choice of *when* to commit, which matters more than any command.

## Diagnose this: a history nobody can read

Here’s the log of a repository someone handed in. Find what would help a stranger.

```
f3a91c2 stuff
9b02e7d fix
41d6a88 more changes
0c7e115 final
2a8b9f0 final final
```

Every message is real, and none of them says what changed. “fix” doesn’t say what was broken. “final final” is the sound of someone who didn’t trust “final.” Rewritten, the same history might read: “Count words longer than n letters,” “Return count after the loop, not inside it,” “Add tests for the empty list,” “Read the word list from a file,” “Write a report file with the counts.” Now anyone—including the author in six months—can find the commit where the loop bug was fixed, and see the test that was added to keep it fixed.

## In-home — TERM-02 Ten Commands, Your Own Folder

- **Time:** 30 minutes (desk)
- **Materials:** Your terminal; the TERM-01 drawing; the ten-command table
- **Safety and privacy:** Practice folder only. Nothing in it you’d miss.
- **The fun:** Feeling the computer do exactly what you typed
- **The skill:** Moving, copying, and listing files in a practice folder only
- **The crumb:** Command log

**How it goes.** Open the terminal. Type `pwd` and compare it with the top of your drawing. Then type your paper plan, line by line: `cd learning`, `mkdir practice`, `cd practice`, `mkdir recipes`. After each, type `pwd` or `ls` and check against the drawing. Then try the rest of the table on a throwaway file: `cp scaler.py scaler-copy.py`, `mv scaler-copy.py old.py`, `ls` (you’ll see `old.py`), `rm old.py`, `ls` again (it’s gone).

Write the command log as you go: command, what you expected, what happened. When something surprises you—`cd` says “No such file or directory,” for instance—that’s almost always the wrong folder or a misspelled name. Type `pwd` and `ls` and read. The log is the crumb, and it doubles as your personal cheat sheet.

## In-home — TERM-03 Run Python From the Terminal

- **Time:** 20 minutes (micro)
- **Materials:** The recipe `scaler` that takes its numbers from the command line
- **Safety and privacy:** Practice folder only.
- **The fun:** Running your own program like a real tool
- **The skill:** Running a script with an argument; reading its output
- **The crumb:** Command and output

**How it goes.** The version of the `scaler` in this chapter reads its two numbers from the command line using `sys.argv`, the list of words you typed after the program’s name:

```
import sys

cups = float(sys.argv[1])
servings = float(sys.argv[2])
print(cups * servings / 4)
```

Predict, then run `python3 scaler.py 2 6`. It prints `3.0`. Now try two planned crashes, one at a time, and read each traceback bottom-up as in Chapter 3. `python3 scaler.py two 6` ends with `ValueError: could not convert string to float: 'two'`—the program can’t turn a word into a number. `python3 scaler.py 2` ends with `IndexError: list index out of range`, because there’s no second number for `sys.argv[2]` to find. Each last line tells you exactly what went wrong. Write each command and its output on one page; that’s the crumb.

## Out-of-home — OUT-03 Find Your People

- **Time:** 45 minutes (desk)
- **Materials:** A notebook; your own Git log printed or on a laptop
- **Safety and privacy:** Public meeting places; share only practice repositories, never anything with family data.
- **The fun:** Seeing how other people keep their histories
- **The skill:** Visiting a library coding group, meetup, or community-college open house
- **The crumb:** Notes and next date

**How it goes.** You met this card in Chapter 1; this time, go with Git in mind. Many libraries and community colleges host coding groups or open labs, and many towns have informal meetups. Go to one with a single question ready: “Could you show me how you use Git on a project?” People who use it every day are usually glad to show a real history—how often they commit, how they write messages, what they do when they break something.

Write down two habits you saw and one command you didn’t know. Before you leave, write the date of the next meeting. What usually happens is that you discover the experts commit more often and write shorter, plainer messages than you expected. That’s the crumb: notes, and a next date.

## Diagnose this: the password Git remembers

Here’s an incorrect example. A learner commits a script that has his email password typed into it, notices an hour later, deletes the line, and commits again with the message “Removed password.” He feels safe. He isn’t, quite.

Every commit is a saved snapshot, and the earlier snapshot still holds the password. Anyone who has a copy of the repository can run `git log`, find the earlier commit, and use `git show` to see exactly what it contained.<sup>21</sup> If the repository was ever pushed to a remote, or copied to another computer, the old snapshot went with it. Deleting a line from today’s file doesn’t delete it from history; that’s the whole point of history.

The calm fix has two parts, in this order. First, treat the password as leaked: change it, and turn on two-step login if it isn’t already on (Chapter 7). That step protects you no matter who has a

copy. Second, prevent the next one: keep secrets in a separate file that's listed in `.gitignore`, and read `git status` before every `git add`, looking for any file you didn't mean to include. Rewriting history to scrub the old commit is possible, but it's an advanced job and easy to get wrong; changing the password is the part that actually keeps you safe.

## In-home — GIT-01 Commits as Save Points

- **Time:** 15 minutes (micro)
- **Materials:** Paper, pencil
- **Safety and privacy:** Paper only.
- **The fun:** Drawing your project's story
- **The skill:** Drawing a history line: what changed, why, and how to go back
- **The crumb:** History drawing

**How it goes.** Draw a horizontal line with dots on it, left to right. Each dot is a commit. Under each, write a message for a change in a project you know—the shipping rule from Chapter 3 works well: “Free shipping at more than 50,” “Test the 50 boundary,” “Use at least 50,” “Error for negative totals.” Then mark the dot you'd go back to if the last change broke something.

This is the picture behind every Git command. `commit` adds a dot. `log` reads the dots. `restore` brings back the most recent dot's version of a file. If your drawing makes sense, the commands will.

## In-home — GIT-02 First Repository

- **Time:** 60 minutes (deep)
- **Materials:** Your terminal; Git installed; a small working project (the recipe scaler is ideal)
- **Safety and privacy:** Practice folder only. No passwords, keys, or family data in the repository.
- **The fun:** A real history of your own work, with your name on it
- **The skill:** `init`, `add`, `commit`, `log` on a real learning project
- **The crumb:** Repository with three commits

**How it goes.** Follow the worked session above with your own project. Check your folder with `pwd` first. Then `git init`, `git status` (Git lists your files as untracked—it doesn't know them yet), `git add` for each file you want, `git commit -m` with a clear message, and `git log --oneline` to see it.

Before any `git add`, look for anything private. If your project folder has a file of personal notes or a password, keep it out with a file called `.gitignore` that lists the names Git should ignore—one per line. Add and commit `.gitignore` itself. Then make two more real changes, committing after each, and you have three commits: the crumb. Check the current Git documentation for installation on your computer, because the steps vary.<sup>21</sup>

## In-home — GIT-03 Commit Messages a Stranger Can Read

- **Time:** 10 minutes (micro)

- **Materials:** The “Diagnose this” log above, or your own
- **Safety and privacy:** Paper only.
- **The fun:** Turning “stuff” into a story
- **The skill:** Rewriting vague messages (“stuff,” “fix”) into clear ones
- **The crumb:** Before/after list

**How it goes.** Write five vague messages on the left of a page. On the right, rewrite each so a stranger would know what changed. A good message usually starts with a verb and names the thing: “Add,” “Fix,” “Remove,” “Round,” “Test.” Keep it short—one line, about the length of a text message.

A helpful test: read only your new messages and ask whether you could find the commit that fixed a particular bug. If yes, they’re good. If you’re doing this with a teen, rewrite one of each other’s messages and compare.

## In-home — GIT-04 Undo Without Panic

- **Time:** 30 minutes (desk)
- **Materials:** Your practice repository from GIT-02
- **Safety and privacy:** Practice repository only.
- **The fun:** Breaking your file on purpose and getting it all back
- **The skill:** Restoring a file and viewing an old version in a practice repository
- **The crumb:** Note of the command that saved you

**How it goes.** Make sure `git status` says the working tree is clean. Then break the file on purpose: delete a line, or type nonsense into it. Run the program and watch it fail. Now `git status --short`—the M shows Git noticed. Predict what `git restore scaler.py` will do, then run it. The file is back as it was in the last commit. Run the program to prove it.

Second exercise: look at an old version without changing anything. `git log --oneline` shows the IDs. `git show HEAD~2:scaler.py` prints the file as it was two commits ago—in the worked session, its last line is the unrounded `print(cups * servings / 4)`. HEAD means “the latest commit,” and ~2 means “two before it.” Nothing has changed in your folder; you’ve only looked. Write the command that saved you on a card and keep it by the computer. One caution: `git restore` throws away uncommitted changes for good, so use it only when you mean it.

## Talk to yourself (this chapter)

**Opening:** “Which folder am I in, and what will this command change?”

**Follow-ups:** 1. **Prediction** — What did I expect this command to do, and to which files? 2. **Evidence** — What do `pwd`, `ls`, and `git status` actually say? 3. **Location** — Am I in the folder I think I’m in, and is this the file I think it is? 4. **Change** — What’s the one command I’ll run next, and what do I expect `git status` to say after it? 5. **Revision** — Did my plan—the tree drawing or the history line—need to change instead?

**How to wait:** Set a five-minute timer. Type nothing that changes anything. Run only `pwd`, `ls`, `git status`, and `git log --oneline`, and read every line they print. Redraw the tree or the history

line on paper.

**What a stuck silence usually means:** You're one folder off. Type `pwd` and read it slowly.

**Don't say / say instead:** Don't say "I'll just delete it and start over." Say instead "Let me look at the log and restore the last good version."

## Coaching your kids at every age

Save points are a natural idea for children—every video game and board game has some version of "go back to where it was." The adult's job is to make the habit visible.

### Ages 5–10

**The move:** take a picture of the block tower before you change it. When your child builds something—blocks, a drawing, a short paper program—suggest taking a photo (of the build, not the child) before trying a big change. If the change goes badly, look at the photo together and rebuild from it. That's a commit and a restore, without a computer.

**Exact words:** "Let's take a save-point picture before we try that." · "Do you want to go back to the picture, or keep this new one?" · "What changed between the picture and now?"

**One thing to avoid:** making the decision for them. Whether to go back or keep the new version is theirs. The point is that they *can* go back, so trying something bold is safe.

### Ages 11–14

**The move:** two save points per session. Whether your child uses Git or simply saves numbered copies of a file, ask for two save points each session: one when something first works, one at the end. Help them name each in a sentence.

**Exact words:** "What works right now that didn't before? That's your save-point message." · "Which folder are you in?" · "If this breaks, which save point would you go back to?"

**One thing to avoid:** typing terminal commands on their keyboard. If they're stuck in the terminal, read `pwd` aloud together and draw the tree on paper. Their fingers, their commands.

### Ages 15–18

**The move:** read their commit history together, and ask about one message. Ask your teen to run `git log --oneline` on a project and walk you through it. Pick one message and ask what changed and why. You don't need to understand the code to ask a good question about the history.

If your teen is using it, *Computing for Emerging Adults*, Chapter 6, "Building for People: The Web, the Command Line, and Git," covers these tools from the teen's side.

**Exact words:** "Can you show me your last few commits?" · "What was going on here, in this one?" · "Which commit would you show someone to prove this is your work?"

**One thing to avoid:** putting anything private in a public repository. GitHub, a popular site for sharing repositories, requires users to be at least 13; for minors, keep repositories private unless you’ve decided together, as the adult, that something is safe to share—no names, faces, school, or town, and never passwords or family data.<sup>22</sup>

## If it isn’t clicking

**The terminal feels frightening.** Go back to the practice folder and the paper tree. Run only the three read-only commands—`pwd`, `ls`, `cat`—for a whole session. Nothing can be harmed by looking. Fear fades fast once you’ve spent twenty minutes typing commands that only read.

**Git feels confusing.** Draw the history line (GIT-01) before every Git session for a week. Before each command, point to the drawing and say what will change on it. Most confusion comes from losing the picture, not from the commands themselves.

**You’ve lost work, or think you have.** Stop typing. Run `git status` and `git log --oneline` and read them. If the file was committed, GIT-04 will get it back. If it wasn’t, check your second copy from your usual backup. Then commit more often from now on—that’s the lesson, not a failure.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. After two logged sessions stuck on the same terminal or Git problem, bring your command log and the output of `git status` to a library coding group or a friend who uses Git. Those two things let a helper see your situation in a minute.

## Tools, including AI

The AI box in **The Computing Hour** holds the rules; here’s what differs for the terminal and Git.

**Never run a command from an AI tool that you can’t explain**—the same rule as for commands from the internet. Tools often suggest commands confidently, including ones that delete or overwrite. Ask the tool to explain each part, then check that part against the official documentation, then decide.

A tool can be useful for turning a description into a commit message, but write your own first; the message is part of your record of what *you* did. If a tool helped with a change, say so on the log: “Help I used: ...”

For tools: your computer’s built-in terminal; Git from its official site; and, if you prefer windows to typing, a Git app or your editor’s Git panel. Either path is fine—the four commands (`init`, `add`, `commit`, `log`) and the idea of restore are the same underneath. Check the current version and terms of any tool or site before you rely on it.

## What “done enough” looks like

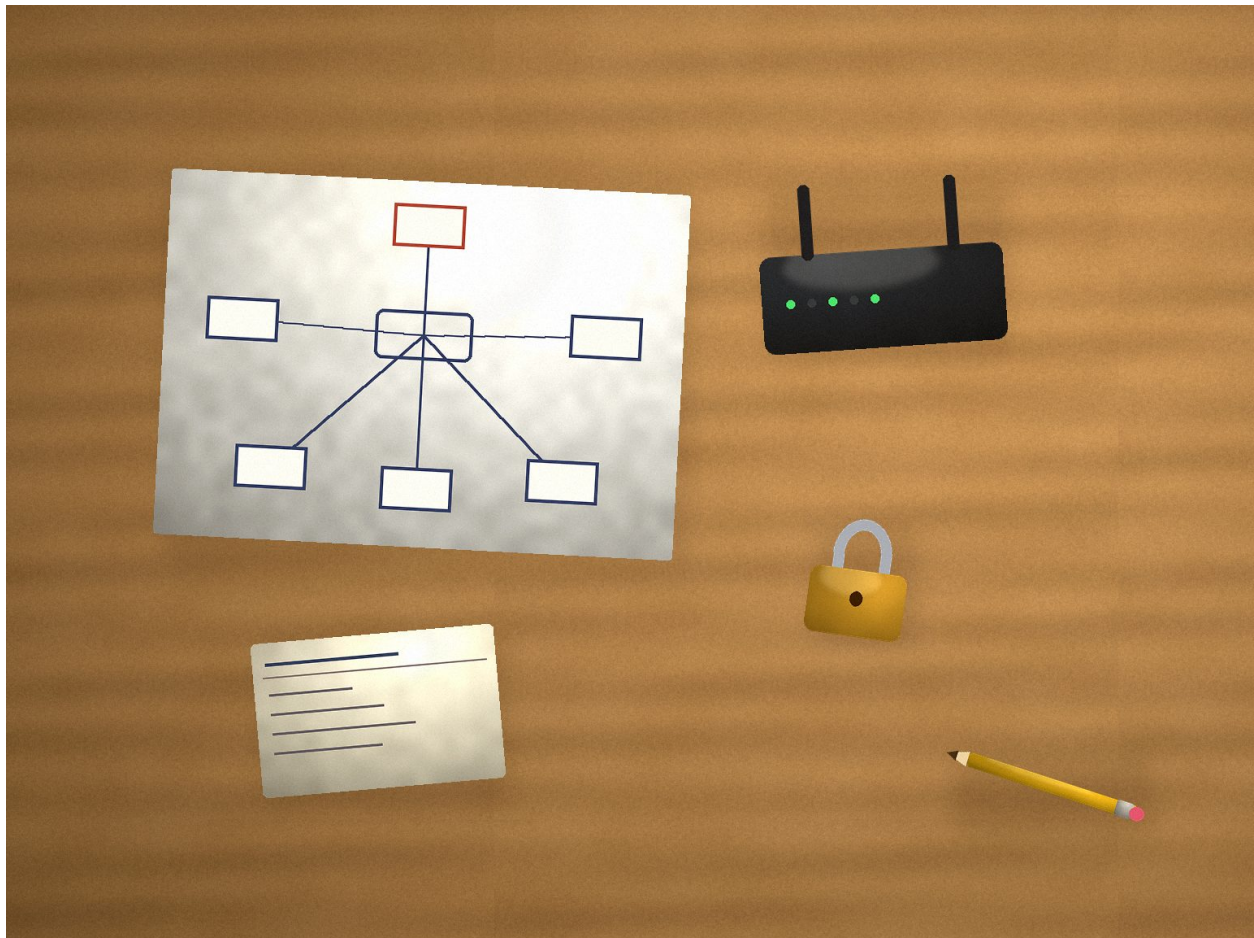
You’re ready to move on when you have:

- ☐ A dated tree drawing of your learning folder, with your practice folder starred
- ☐ A command log from TERM-02 covering the ten commands, run in the practice folder only
- ☐ A Python program run from the terminal with arguments, and two tracebacks read bottom-up
- ☐ A repository with at least three commits whose messages a stranger can read
- ☐ A `.gitignore` that keeps anything private out
- ☐ One restore with `git restore`, and one old version viewed with `git show`
- ☐ **AS-01 Cold Trace** once: on paper, predict what `pwd` and `ls` will print after a short list of `cd` and `mkdir` commands
- ☐ **AS-02 Debug on the Spot** once: someone renames or moves a file in your practice folder; find it and fix the path with no tool open
- ☐ **AS-04 Explain Your Code** once: walk a person through your Git history, commit by commit, explaining what changed and why

Placement by skill: if you already use a terminal at work, go quickly through the commands and spend your time on Git and on messages. “Done enough” means ready for the next chapter. You’ll use both tools for as long as you build things.

# Chapter 7

## Systems and Security at Home: Backups, Passwords, and the Family Rule



### Why this matters

In most homes, somebody ends up being the systems person. They're the one who knows where the router lives, who gets the call when the printer vanishes from the network, and who remembers,

vaguely, that the family photos are “in the cloud somewhere.” Often that person is a parent, and often they took the job by accident.

This chapter makes the job deliberate, and smaller than it looks. You don’t need to become a security professional to keep a household safe. You need a short list of habits done well: long passphrases kept in a password manager, two-step login on the accounts that matter, updates switched on, a calm way of handling suspicious messages, and backups you have actually tested. A federal guide for ordinary households boils the core down to four easy steps, and most of this chapter is those four steps done at your kitchen table.<sup>23</sup>

Calm beats fear here. Security advice is often sold through dread, and dread makes people either freeze or buy things they don’t need. The better posture is the one you’ve been practicing all book: **Specify** what you’re protecting, **Predict** what could go wrong, **Build and Test** a fix, **Debug and Log** what you learned, and **Explain and File** the result so the next person in the house can follow it. A household threat model is just a spec for safety.

There’s a second reason this chapter matters, and it’s the one your kids will notice. As they learn more computing, they’ll learn how systems can be abused as well as used. Every rung of this series draws the same line, and you’ll draw it at home too: **we practice only on systems we own, or have written permission to test**. You’ll write that line down, sign it, and keep it with your records. It costs nothing, and it protects curious, capable people from crossing a line they didn’t see.

Finally, nearly everything you do in this chapter protects you directly, today. A tested backup is the difference between an annoying afternoon and losing ten years of photos. Two-step login on your email is the difference between a strange password-reset notice and a stolen identity. Few chapters in any book pay off this quickly.

## Understand it yourself

### The plain picture

Think of your household’s digital life the way you think of your house.

- **Passwords are keys.** A long key is harder to copy than a short one with a clever notch. A password manager is the key cabinet: you remember one strong combination, and it holds the rest.
- **Two-step login is a deadbolt.** Even if someone copies your key, they also need the second thing: a code from your phone, an app, or a small physical key.
- **Updates are repairs.** When a maker finds a weak hinge, they send a fix. An update you haven’t installed is a repair kit still sitting in the box.
- **Backups are a fire drill.** It isn’t enough to own an extinguisher. You need to know it works, which means trying a restore before you need one.
- **Phishing is a con at the door.** Someone dresses as the gas company and asks to come in, right now, before you have time to think. The fix is to slow down and call the real gas company.

And the threat model is the conversation you’d have about any house: what’s valuable here, who

might want it, and what's the cheapest strong lock?

## The precise picture

Here's the same picture with the words you'll meet on settings pages.

**Inside the box.** Every device, from a laptop to a phone to a smart speaker, has the same four jobs inside it. The **processor** follows instructions. **Memory** holds what's being worked on right now and forgets it when the power goes off. **Storage** keeps files when the power is off. The **network** connection lets it talk to other devices. The operating system manages all four and decides which program may touch which file. That's why separate user accounts on a shared computer are more than politeness: they let the operating system keep one person's files out of another's way.

**Authentication factors.** A login can check three kinds of evidence: something you know (a password), something you have (a phone or a security key), and something you are (a fingerprint). A **two-step login** (a password plus a second thing, such as a code or a key) uses two different kinds. Settings pages may call it multifactor authentication, two-factor authentication, or 2-step verification. They mean the same idea.

**The length rule.** The current federal standard for websites that handle logins says a password used on its own should be at least 15 characters long. When the password is paired with a second factor, the minimum drops to 8. The same standard says sites should allow passwords of at least 64 characters and accept spaces.<sup>24</sup> That's why the modern advice is a **passphrase**: four or more unrelated words, with spaces if the site allows them. Length does the work that symbol tricks used to pretend to do.

**Updates as patches.** Software has bugs, just like the programs you've been writing, and some bugs let strangers in. An update is the maker's fix. Automatic updates on every household device are the cheapest security you'll ever buy, because they cost nothing but a restart.

**Backup versus sync.** These sound alike and behave very differently. **Sync** keeps two places identical: delete a photo on your phone, and the synced copy disappears too. A **backup** is a separate copy that doesn't follow your mistakes. Many families have sync and think they have backups. The test is simple: if you deleted a folder by accident this morning, could you get yesterday's version back? A common rule of thumb is to keep at least two copies in two different places, one of them away from the house.

**A threat model** (what you're protecting, from whom, and the cheapest strong fix) has three columns: assets (the things that would hurt to lose), threats (who or what could cause the loss), and fixes. For most households the assets are the email account (because it resets everything else), the bank, the family's photos, and the kids' schoolwork. The threats are mostly ordinary: a reused password leaked from some other site, a convincing fake message, a dead drive, a lost phone. The fixes are mostly cheap.

**The family security rule.** Testing, scanning, or guessing passwords on a system you don't own can be against the law, even when the person doing it meant well. The household line is plain: practice only on what you own, or with written permission from the owner.

## Wrong answers you should be able to hear

You'll hear these from yourself first, then from your kids. Each has a calm reply.

- **“My password has a symbol, so it's strong.”** Old advice. Ask: “How long is it?” A short password with a symbol is still short.
- **“We back up.”** Ask: “When did we last restore something?” A backup you've never restored is a hope, not a backup.
- **“It's only a practice scan.”** Ask: “Whose network is it?” If it isn't ours, the answer is no, no matter how harmless the tool.
- **“I keep the codes on a sticky note by the screen.”** A second factor written next to the first one is one factor. Keep backup codes in the password manager or in a locked drawer, never on the device.
- **“We're too small to be a target.”** Most attacks aren't aimed at anyone in particular. They're aimed at everyone, and they land on whoever left a door open.

## A five-minute warm-up

Take a sheet of paper and write the five accounts that would hurt most to lose. Don't open a computer. Most adults write email first, then banking, then something like photos, a phone account, and a shopping account with a saved card. Now circle the one that resets the others. That's almost always email, and it's where this week starts.

## How to run it this week

This week has three sessions, in this order: two-step login, a backup with a test restore, and the family rule. Each one follows the steps of **The Computing Hour**.

### Session 1 (desk, about 40 minutes): Two-step login

Use **SEC-02 Turn On Two-Step Login**. **Specify:** “Two-step login on email and the bank, with backup codes stored safely, and no codes written on paper near the device.” **Predict:** how long you think each account will take, and which one will fight you. **Build and Test:** turn it on, then sign out and sign back in to prove it works. **Debug and Log:** if a step fails, write the Debug Log page. **Explain and File:** a checklist of which accounts are done, with no codes on it.

### Session 2 (desk, about 40 minutes): A backup and a test restore

Use **SYS-02 Files, Folders, and Backups**. Write a one-page backup plan: what gets backed up, where, how often, and who checks. Then do the part most people skip: restore one small folder to a new location and open a file from it.

### Session 3 (micro, about 15 minutes): The family rule

Use **SEC-05 The Family Security Rule**. Write the rule in your household's words, sign it, date it, and file it. If you have a teen, write it with them.

The self-talk for every session is the chapter's question: *What are we protecting, from whom, and what's the cheapest fix?* When you're unsure what to do next, say it out loud and pick the cheapest fix.

### In-home — SEC-02 Turn On Two-Step Login

- **Time:** 30 minutes (desk)
- **Materials:** your phone, the password manager if you have one, a printed checklist of your five most important accounts
- **Safety and privacy:** Your own accounts only. Write no codes, passwords, or recovery answers on the checklist; it records only “done” and the date.
- **The fun:** the small satisfaction of signing back in and watching the second step stand guard
- **The skill:** multifactor authentication on the accounts that matter most
- **The crumb:** Account checklist (no codes written)

**How it goes.** Start with email, because email resets everything else. Find the security settings, choose an authenticator app or a security key if you can (text-message codes are better than nothing), and save the backup codes inside your password manager or print them and put them somewhere locked. Then sign out completely and sign back in. That last step is the test. Most people find one account where the setting hides two menus deep; write down where it was, so the next account goes faster. Tick the box, write the date, move to the bank.

### Worked session: two-step login and one test restore

*An illustration, not a reported family:* Picture a parent at the kitchen table on a Thursday evening with a 40-minute window and a mug of tea going cold.

She opens the notebook and writes the spec: “Two-step login on my email and our bank. One test restore of the ‘Summer’ photo folder from the backup drive. Done means I can sign in with the second step on both, and I can open one restored photo.” She predicts email will take ten minutes and the bank fifteen, and that the restore will “just work.”

Email takes eight minutes. The bank takes twenty, because the setting lives under “Profile,” not “Security.” She writes that down for next time.

Then the restore. She plugs in the backup drive, finds the backup app, and asks it to restore “Summer” to a new folder on the desktop. The folder appears. It's empty.

She doesn't panic, and she doesn't start clicking around. She opens the Debug Log page:

**Date:** Thursday **Expected:** “Summer” folder restored with about 300 photos. **Actual:** Folder restored, zero files inside. **Guess:** The backup only includes the Documents folder, not Pictures. **One change:** Open the backup settings and read the list of included folders. **Result:** Confirmed. Pictures wasn't selected. Added it, ran a fresh

backup, restored again: 312 photos, opened two. Backup plan updated: “Pictures included as of today; test restore monthly.”

Her exit line in the log: “I specified two-step login on two accounts and one restore. I predicted the restore would work; it didn’t, because Pictures wasn’t in the backup. Next I’d check the phone’s backup the same way. Help I used: the backup app’s own help page.”

That empty folder was the most valuable thing she found all week. A backup that had been quietly skipping the photos for months would have stayed invisible until the day the drive died.

## Stopping when the timer ends

Security work tempts you to keep going, because every settings page leads to another. Stop when the timer ends. Write the next account on the checklist, so tomorrow starts there. An account half-done is fine if the log says where you stopped. An account changed in a rush at 11 p.m. with no note is how people lock themselves out.

## Practice that actually builds it

### Blocked: passphrases

Start with one skill until it’s easy. **SEC-01 Passphrase Workshop** takes 15 minutes. Make three practice passphrases of four or more unrelated words, count the characters, and say them aloud. Notice how easy a 25-character phrase is to remember compared with an 8-character jumble. Then destroy the practice sheet. These are for practice, not for use.

If you don’t yet use a password manager, this is the week to choose one. Pick one with a good reputation, protect it with your strongest passphrase and two-step login, and move your accounts in a few at a time, starting with the five from your warm-up. The manager generates long random passwords for every other site, so the only passphrase you must remember is the one that opens it.

### In-home — SEC-01 Passphrase Workshop

- **Time:** 15 minutes (micro)
- **Materials:** paper, a pencil, a timer
- **Safety and privacy:** Practice phrases only; never write a real password on this sheet; shred it when you’re done.
- **The fun:** inventing a ridiculous, memorable phrase like “kettle orbit lantern pickle”
- **The skill:** long passphrases; why length beats symbols; a password manager
- **The crumb:** Practice phrase (destroyed after)

**How it goes.** Write three practice phrases of four random words each. Count the characters, including spaces, and check each against the 15-character minimum. Then write one “clever” short password the old way, with a capital, a number, and a symbol, and count it too. The comparison makes the point better than any lecture. Read your favorite phrase aloud twice, cover it, and write it from memory. Then shred the page.

## Mixed: update day plus backup

Once two-step login feels routine, mix skills. **SEC-04 Update Day** pairs well with a backup check: before you update every device, confirm the backup ran recently, so an update that goes wrong costs nothing. Walk the house with a list: each laptop, each phone, each tablet, the router, the smart TV, any game console. For each, check that automatic updates are on, install anything waiting, and write the date. Routers and smart devices are the ones most often forgotten.

### In-home — SEC-04 Update Day

- **Time:** 30 minutes (desk)
- **Materials:** the device list from your network map, a pencil, chargers
- **Safety and privacy:** Household devices only; check that a recent backup exists before you start.
- **The fun:** crossing off a whole house's worth of devices in one sitting
- **The skill:** updating operating systems, browsers, and apps on every household device
- **The crumb:** Dated update list

**How it goes.** Pick an evening when nobody needs a device for an hour. Start with the computers, because their updates take longest; set them going and move on. Then phones and tablets, then the browser on each computer, then the router, the TV, and anything else on the map. For each one, write three things: the date, “automatic updates on: yes/no,” and anything that needed a restart. The first update day usually turns up one device that hasn't updated in a long time, often the router or an old tablet in a drawer. Decide whether it gets updated, retired, or kept off the network. After the first time, update day becomes a ten-minute check each season, because automatic updates do most of the work.

### Choosing a second step

Not every second factor is equal, and it helps to know the order without fussing over it. A **security key** (a small physical device you plug in or tap) is the strongest common choice, because a fake website can't trick it. An **authenticator app** on your phone, which shows a fresh six-digit code every thirty seconds, is a strong, easy default for most families. A **text-message code** is weaker, because phone numbers can sometimes be moved to another phone by a determined scammer, but it's far better than nothing. Use the best option each account offers, and don't let the perfect choice delay a good one.

Whatever you choose, plan for losing your phone. That's what backup codes are for. Store them in the password manager or print them and lock them away, and make sure a second adult in the house knows where. A household where only one person can get into the email account is one dropped phone away from a very long week.

### The backup plan, written out

A backup plan fits on half a page. Here's the shape, as a labeled illustration rather than a recommendation of any particular product:

**What:** Documents, Pictures, the kids’ portfolios folder, the password manager’s emergency kit. **Where:** an external drive at home (weekly) and an online backup service (continuous). **Who checks:** Dad, first Sunday of the month. **Test restore:** one folder, monthly, to a new location; open two files. **Last tested:** (date) — Pictures restored, 312 files.

Notice the two places. A drive at home protects you from a dead laptop; it doesn’t protect you from a fire, a flood, or a burglary that takes the laptop and the drive together. The online copy covers that. Notice, too, that the plan names a person and a day. A backup plan with nobody’s name on it slowly stops being a plan.

### Diagnose this: the sticky-note code

Here’s an incorrect example to look at closely. A parent sets up two-step login on the family’s shared email, then writes the six backup codes on a sticky note and tapes it under the keyboard “so we never get locked out.”

What went wrong? Two-step login is supposed to need two different things. Anyone who sits at that keyboard now has both: the password (saved in the browser) and the codes (taped under it). The fix keeps the safety net and removes the hole: put the backup codes in the password manager, or print them and lock them away from the computer. Say the rule out loud once: *the second factor never lives next to the first*.

### In-home — SEC-06 Threat Model for the Household

- **Time:** 30 minutes (desk)
- **Materials:** one sheet of paper with three columns: What we protect · From whom or what · Cheapest strong fix
- **Safety and privacy:** No passwords, account numbers, or addresses on the page; names of accounts only (“email,” “bank”).
- **The fun:** realizing that most of the scary column has a boring, cheap answer
- **The skill:** what we protect, from whom, and the three cheapest fixes
- **The crumb:** One-page model

**How it goes.** Fill the first column from your warm-up list, then add things that aren’t accounts: the family photos, the kids’ portfolios, the laptop itself. In the second column, write ordinary threats, not movie villains: a leaked password reused from another site, a fake delivery text, a spilled drink, a lost phone, a teen who installs something odd. In the third column, write the cheapest strong fix for each. You’ll find the same few fixes repeating: two-step login, a password manager, updates, a tested backup, a separate user account. Circle the three that cover the most rows and do those first. File the page; review it once a year or when something big changes.

### Out-of-home — SEC-03 Phishing Sort

- **Time:** 15 minutes (micro)

- **Materials:** a dozen printed sample messages (made up, or examples from an official consumer-safety page), three envelopes labeled Real · Suspicious · Report
- **Safety and privacy:** Printed samples only; no live links, no real inboxes on screen.
- **The fun:** playing detective with friends, and arguing over the tricky ones
- **The skill:** sorting messages into real, suspicious, and report
- **The crumb:** Sorted pile with reasons

**How it goes.** Take the stack to a library adult-education session, a friend’s table, or a homeschool co-op meeting with the organizer’s okay. Each person sorts the messages and writes one reason for each choice. The usual clues surface fast: urgency (“act within one hour”), a sender address that’s almost right, a link whose words don’t match where it goes, a request for a code or a gift card. Then compare piles. The disagreements are the lesson. Close with the household habit: pause, don’t click from the message, open the real site or call the real number yourself, and report the fake.

## Out-of-home — SYS-03 Home Network Map

This one starts at home and ends anywhere you can sit with a notebook: a library table, a coffee shop, a park bench.

- **Time:** 20 minutes (micro)
- **Materials:** paper, a pencil, a list of devices you made the night before
- **Safety and privacy:** No addresses, network names, or passwords written down; label devices by type only (“kitchen laptop,” “teen’s phone”).
- **The fun:** discovering how many things in your house talk to the internet
- **The skill:** drawing router, devices, and guests, with no addresses or passwords written down
- **The crumb:** Network sketch

**How it goes.** Draw the router in the middle. Draw a line to every device that connects to it: computers, phones, tablets, the TV, the thermostat, the doorbell, the speaker. Draw guests in a separate corner. Most families are surprised by the count. Then ask two questions of the drawing: which of these get updates automatically, and does the guest corner have its own network? You don’t need to fix anything today. You need to see it.

## Talk to yourself (this chapter)

**Opening:** “What are we protecting, from whom, and what’s the cheapest fix?”

**Follow-ups:** 1. **Prediction** — What did I expect this setting, restore, or update to do before I touched it? 2. **Evidence** — What exactly did the screen show: a confirmation, an error, an empty folder? 3. **Location** — Where does reality first part ways with my plan: this account, this device, this step? 4. **Change** — What’s the one change I’ll make, and what do I expect it to do? 5. **Revision** — Did my threat model or backup plan need to change instead?

**How to wait:** set a five-minute timer before changing anything on a live account; read the settings page twice; write the next step down before you click it.

**What a stuck silence usually means:** the list is too long—pick three.

**Don't say / say instead:** Don't say "We're too small to be a target." Say instead "Let's turn on two-step login for the accounts that matter most."

## FAM-02 Shared-Computer Check

Many families share at least one computer, and shared computers collect things: photos in the downloads folder, a teen's portfolio on the desktop, a saved login nobody remembers making. **FAM-02 Shared-Computer Check** is a 15-minute walk through each shared machine. List who uses it, give each person their own user account if you can, and move family photos and your teen's portfolio to a family drive that's backed up. If you use Grok Bot, remember that all of your bots share one computer. Keep photos of your children and family off it, and keep your student's portfolio on a family drive.

## Stop, Save, Tell

One more household skill belongs here, and it's best learned on a quiet day. If anyone in the house is threatened, pressured, or blackmailed online—including with a real or AI-made intimate image—the response is the same three words: **Stop** replying and don't pay; **Save** the evidence (screenshots, usernames, links) without forwarding the images; **Tell** a trusted adult, and report to the platform and the authorities. Since May 2026, federal law has required covered platforms to remove reported nonconsensual intimate images, including computer-made ones, within 48 hours of a valid request, and the Federal Trade Commission takes complaints about platforms that don't.<sup>25</sup> For images of anyone under 18, the National Center for Missing & Exploited Children runs a service called Take It Down that helps get them removed.<sup>26</sup> If you have a teen, read the sextortion box in *Computing for Emerging Adults*, Chapter 8, together; it explains how this scam usually unfolds and why telling quickly is the protection. Then rehearse the three words with **FAM-04 Stop, Save, Tell** and put the script card where everyone can find it.

## A plain word on monitoring

How much should you watch a teen's devices? Families land in different places, and this isn't legal advice. A sound general approach is to match supervision to age, tell your kids plainly what you monitor and why, and move toward privacy as they show judgment. Secret monitoring tends to cost more trust than it buys. Open rules, written down and revisited, tend to hold.

## Coaching your kids at every age

You don't have to be a security expert to coach this. You have to model it. Your kids will copy what you do with your own phone far more than what you say.

## Ages 5–10

**The move:** **SYS-01 What's Inside the Box**, together, on paper. Pick a household device and draw four boxes: thinking (processor), remembering-for-now (memory), keeping (storage), and talking

(network). Don't open anything; draw what it must have.

**Exact words:** "A password is a secret, like a toothbrush. You don't share it, even with your best friend. Grown-ups at home can know it, and that's all."

**One thing to avoid:** fear stories. A young child doesn't need to hear about criminals. They need a few clear rules and an adult who answers questions calmly.

## Ages 11–14

**The move:** SEC-03 **Phishing Sort** as a game. Print a handful of sample messages and race to sort them. Give a point for every clue spotted, even on the real ones.

**Exact words:** "What would make you stop before you clicked this? What would you check first?"

**One thing to avoid:** making them the household's tech support without the rules. A capable twelve-year-old can turn on updates for the family; they shouldn't be the only person who knows the router's password.

## Ages 15–18

**The move:** SEC-05 **The Family Security Rule**, written together and signed by both of you. Teens at this rung are often learning how systems break; the rule is what makes that learning safe.

**Exact words:** "You can take apart anything we own. You can break your own practice apps on purpose. What we don't do is test anyone else's system, even to help, even if a friend says it's fine. Let's write that in our own words."

**One thing to avoid:** treating curiosity as a threat. The teen who asks "how would someone get into this?" is asking the right question. Answer it on systems you own, and let them teach you what they find.

## If it isn't clicking

**You feel overwhelmed.** Security has no end, and every article adds a new worry. Go back to your threat model and do only the three cheapest fixes from it. Ignore everything else until those are done. Most of the protection lives in those three.

**The family resists.** A spouse doesn't want two-step login; a teen rolls their eyes at the password manager. Start with your own accounts. Do them well, show how little friction it adds, and offer to set it up for one other person when they're ready. Change spreads by example.

**Your teen pushes back on rules.** Co-write the rule (SEC-05) instead of handing it down. Teens hold rules they helped write far better than rules they were given, and the conversation itself is the lesson.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. If you've spent two

logged sessions stuck on the same thing, such as an account you can't recover, a backup app that won't cooperate, or a router whose settings you can't reach, ask a human. A library tech session, a knowledgeable friend, or the device maker's official support line is the right next step. If a message threatens you or your child, skip straight to Stop, Save, Tell.

## Tools, including AI

The AI box in **The Computing Hour** holds here. Two notes are specific to security.

First, never paste a password, a backup code, a recovery answer, or a screenshot of a security settings page into an AI tool. Ask general questions ("How does an authenticator app work?") and keep your actual accounts out of it.

Second, AI tools can give confident, out-of-date security advice. If an answer tells you to change your password every month or to add a symbol for strength, it's repeating older guidance. Check anything important against the official pages in Chapter 10, and check the current version of any settings page, because menus move.

For tools, you need very little: a password manager, an authenticator app or security key, a backup drive or backup service, and automatic updates turned on. Choose by reputation and fit, not by fear.

## What "done enough" looks like

You're done enough with this chapter when you have:

- ☐ two-step login on your email, your bank, and the family's cloud storage, with backup codes stored away from the device (SEC-02)
- ☐ a written backup plan and one successful test restore, dated (SYS-02)
- ☐ a one-page household threat model with three fixes circled and done (SEC-06)
- ☐ a signed, dated family security rule filed with your records (SEC-05)
- ☐ an update list showing every household device checked (SEC-04)
- ☐ a Stop, Save, Tell script card where the family can find it (FAM-04)
- ☐ **AS-04 Explain Your Code**, adapted: explain your backup plan aloud to a partner, including how you'd restore a folder, without looking at notes

Placement by skill: if you already had a tested backup and two-step login everywhere, skip to the threat model and the family rule, and spend the rest of the week on Chapter 8.

# Chapter 8

## Working With AI as an Adult Learner: Where It's Wrong and Whose Work It Is



### Why this matters

You're going to use AI tools. Most working programmers already do, or plan to, and so will your kids.<sup>27</sup> The question this chapter answers isn't whether to use them. It's how to use them so that

you're still the one learning.

That matters twice over for a parent. Whatever habits you keep with an AI tool are the habits your kids will copy. If you ask a chatbot for a finished function, paste it in, and move on, your teen will notice. If you try first, ask the tool to explain, test what it gives you, and write "Help I used: ..." on your log, they'll notice that too. You can't hold a rule for your kids that you don't keep yourself, and you won't need to lecture about one you do keep.

Here's the plain truth this chapter is built on: **what a tool makes is made by a tool**. A suggested function becomes part of your work only after you can explain every line and it passes your tests. That isn't a moral stance against AI. It's the same standard you'd apply to code copied from a book or a forum, and it's what keeps you able to fix the program when it breaks at 10 p.m. with nobody to ask.

People sometimes ask whether AI makes learning to program pointless. Someone still has to say what's wanted, check what came back, and fix it when it's almost right. That's exactly what this book teaches, and it's what the focus line has been saying all along: **specify it small; predict before you run; debug before you rewrite; explain it before you call it yours**.

This chapter also covers the household side. Most consumer AI tools have age rules, privacy settings, and terms that change. You'll learn to look them up, write them down with a date, and turn them into a short family AI agreement that you write and your older kids co-sign.

## Understand it yourself

### The plain picture

A text-generating AI is a very well-read guesser. It has seen an enormous amount of writing and code, and when you give it the start of something, it guesses what most likely comes next, one small piece at a time. It's remarkably good at sounding right. Sounding right and being right are different things, and the gap between them is where your judgment lives.

So treat what it gives you the way you'd treat a confident stranger's directions. Often useful. Sometimes out of date. Occasionally invented. Always worth checking against the map.

### The precise picture

**How text generation works.** A large language model learns patterns from its **training data** (the examples an AI learned from). When you type a prompt, it predicts likely next words, or pieces of words, again and again until it stops. It isn't looking facts up in a reference book unless the tool has been connected to search, and even then it can misread what it finds. That's why it can produce a citation that looks perfect and doesn't exist, or a library function with a plausible name that no library has.

**Fluent is not true.** The writing is smooth because smoothness is what the model learned. Truth comes from checking: running the code, testing the edge cases, opening the source.

**Code suggestions are drafts.** A suggested function is a starting point written by something that doesn't know your spec unless you wrote it down. It can invent options, use older ways of doing things, skip the edge case you care about, or quietly do something insecure, such as gluing a user's input straight into a database query (the same risk you met in Chapter 4).

**What the research says about speed.** Two careful studies point in different directions, and it's worth knowing both. In one lab study, developers asked to build a small web server with an AI coding assistant finished 55.8% faster than those without; the researchers included people from the company that makes the tool. In a later randomized study, experienced open-source developers working on their own long-familiar projects took 19% *longer* with AI tools allowed, even though they expected to be faster and still believed afterward that they had been.<sup>28</sup> Different people, different tasks, different tools. These are two honest studies that point different ways. The lesson for you isn't a number. It's this: the feeling of speed isn't a measurement, so measure your own (AI-07).

**What it means for learners.** Researchers who watched university beginners use AI help while programming found a widening gap. Students who already knew what they meant to write used the tool to get there faster and ignored its wrong suggestions. Students who were struggling were often led along step by step, thought they had done better than they had, and came away feeling they understood more than they did.<sup>29</sup> That pattern is why this book puts the attempt first and the explanation last. The habits of **The Computing Hour** are what let a tool help instead of mislead.

**What working programmers say.** In a large 2025 survey of developers, more said they distrust the accuracy of AI tools than trust it, and the most common frustration, named by about two-thirds, was answers that are “almost right, but not quite.” Many also said debugging AI-written code takes longer.<sup>27</sup> “Almost right” is the hardest kind of wrong, because it passes a glance. It doesn't pass a test table.

## Wrong answers you should be able to hear

- **“It ran, so it's right.”** Running isn't passing. Ask: “Which tests did it pass? Did we test the edge cases?”
- **A citation nobody opened.** Ask: “Where's the source? Let's open it and find the sentence.”
- **A function nobody can explain.** Ask: “What does line four do? What happens if the list is empty?” If nobody knows, it isn't ready to keep.
- **“It's my code, I asked for it.”** Asking is a real skill, but it isn't authorship. Ask: “What did the tool write, and what did you write?”
- **“The AI said so.”** Ask: “How would we check?”

## A five-minute warm-up: a next-word table

Before you open any tool, see the guessing for yourself. Write this sentence on paper: *the cat sat on the mat the cat ran*. Make a two-column table. In the left column, write each word. In the right, write every word that came right after it. For “the,” you'll write *cat, mat, cat*. Now play the machine: start with “the,” pick a next word from its list (cat is twice as likely as mat), then look up that word's list, and keep going. You'll produce sentences that sound like the original and mean

nothing in particular. That's the core idea, at toy scale. Real models are vastly larger and cleverer, but they're still predicting what comes next.

If you'd like to see it in Python, here's the same table in six lines:

```
text = "the cat sat on the mat the cat ran"
words = text.split()
table = {}
for first, second in zip(words, words[1:]):
    table.setdefault(first, []).append(second)
print(table["the"])  # ['cat', 'mat', 'cat']
```

Predict the output before you run it. Then try `print(table["cat"])`. (It prints `['sat', 'ran']`.)

## How to run it this week

Three sessions this week: attempt then ask, explain every line, and an age-terms map. They follow the steps of **The Computing Hour** exactly, with one firm rule: the tool comes in only after your own prediction and attempt are on the page.

### Session 1 (micro, about 20 minutes): Attempt, then ask

Use **AI-01 Attempt, Then Ask**. Pick a small problem from your current project. **Specify** it, **Predict** what your code will do, and **Build and Test** your own attempt first. Only then ask a tool to explain an error or suggest a test you missed. **Debug and Log** what it said and whether it was right. **Explain and File** with “Help I used: ...” on the log line.

### Session 2 (desk, about 40 minutes): Explain every line

Use **AI-03 Explain Every Line**, as in the worked session below.

### Session 3 (micro, about 15 minutes): An age-terms map

Use **AI-06 Age Terms Map**. For each AI tool anyone in your family uses or wants to use, look up the current minimum age and teen rules on the company's own help or terms page, and write them in a table with today's date.

The self-talk is the chapter's question: *What did I try first, and can I explain every line of what came back?*

## In-home — AI-01 Attempt, Then Ask

- **Time:** 20 minutes (micro)
- **Materials:** your notebook, your project file, one AI tool you're allowed to use
- **Safety and privacy:** No family names, private messages, or work code in the prompt; practice code only.

- **The fun:** finding out whether you’d already solved it before you asked
- **The skill:** solving first; asking an AI to explain only after; logging what it said
- **The crumb:** Log entry with Help I used

**How it goes.** Set a timer and work the problem yourself until it rings: spec line, prediction, attempt, Debug Log page if it breaks. Then, and only then, ask the tool a narrow question: “Explain this error message,” or “What edge case might this function miss?” Don’t ask it to rewrite your code. Compare its answer with your log. About half the time, you’ll find you were one step away; the other half, you’ll learn something specific. Either way, write the log line: “Help I used: asked a chatbot to explain a KeyError; it was right that the key was misspelled.”

## Worked session: explain every line

*An illustration, not a reported family:* Picture an adult at a desk with 40 minutes and a small spec: “count\_words(text) returns how many words are in a piece of text. Done means it passes my test table.”

He writes his own version first:

```
def count_words(text):
    """Return how many words are in text."""
    return len(text.split())

print(count_words("the cat sat"))      # 3
print(count_words(" two spaces "))     # 2
print(count_words(""))                 # 0
```

He predicts 3, 2, and 0. It prints 3, 2, and 0. Good.

Then he asks a tool, “Write a better word-counting function in Python.” It returns this:

```
import string

def count_words(text):
    cleaned = text.lower().translate(
        str.maketrans("", "", string.punctuation))
    return len(set(cleaned.split()))
```

It looks more sophisticated. He doesn’t paste it in. He explains it line by line, aloud. Line one brings in a list of punctuation marks. The next line makes everything lowercase and deletes punctuation. The last line splits into words, then wraps them in `set(...)`. He stops there. What does `set` do? He checks the official docs: a set keeps only one of each item.

So he writes two tests for the tool’s version:

```
assert count_words("the cat sat") == 3
assert count_words("Hi, hi!") == 2
```

The first passes. The second fails: the tool’s version returns 1, because “Hi” and “hi” become the same word and the set keeps only one. His Debug Log page:

**Expected:** 2 **Actual:** 1 **Guess:** set removes repeated words; the function counts *different* words, not words. **One change:** remove `set(...)` and rerun. **Result:** returns 2. But now: should “Hi,” and “hi!” count as the same word or two words? That’s a spec question.

That last line is the real lesson. The tool couldn’t know what he meant, because he hadn’t written it down. He adds one line to his spec (“count every word, repeats included; punctuation doesn’t count as a word”) and keeps his own simpler function, which already does that. His exit line: “I specified a word counter; I predicted the tool’s version would match mine, and it didn’t, because it counted distinct words. Next I’d add a test with punctuation only. Help I used: a chatbot’s suggested function (not kept); Python docs on `set`.”

## Stopping when the timer ends

AI tools make it easy to keep asking “one more question.” When the timer rings, stop asking. Write the last thing the tool said and whether you’ve checked it. Unchecked answers wait on the page, marked “unchecked,” until next session.

## Practice that actually builds it

### Blocked: claim checks

Start with **AI-02 Fluent Is Not True** until it’s automatic. Ask a tool three factual questions about something in this book: what a Python function does, what a term means, where a rule comes from. For each answer, find the primary source (the official docs, the standard, the study) and write in a two-column table what the tool said and what the source says. Do this three or four times across a few weeks. You’ll quickly develop a feel for where tools are reliable and where they drift: niche details, recent changes, and exact numbers are the usual weak spots.

### In-home — AI-02 Fluent Is Not True

- **Time:** 15 minutes (micro)
- **Materials:** one AI tool, a browser, a two-column claim/check table on paper
- **Safety and privacy:** General questions only; nothing personal in the prompt.
- **The fun:** catching a confident machine in a small, specific mistake
- **The skill:** checking three AI-generated claims against primary sources
- **The crumb:** Claim/check table

**How it goes.** Ask three questions whose answers you can verify: “What does Python’s `str.strip` do?” “What’s the minimum length for a standalone password in the current federal guidance?” “Who wrote the PRIMM approach to teaching programming?” Write each answer in the left column. Then find the real source and write what it says on the right. Mark each row: matches, partly, or wrong. Most sessions turn up at least one “partly.” Keep the table; it’s a good thing to show a teen.

## Mixed: explain every line, then test

Once claim checks feel easy, mix them into your build sessions with **AI-03 Explain Every Line**: after your own attempt, take any AI suggestion, explain each line aloud or in writing, write at least two tests (one ordinary, one edge case), and run them before keeping a single line. This is the same move as the worked session, done on whatever you’re building now.

## In-home — AI-03 Explain Every Line

- **Time:** 30 minutes (desk)
- **Materials:** your own attempt at a small function, one AI suggestion for the same job, a pencil, the official docs open in a browser
- **Safety and privacy:** Practice code only; no work code and no personal data in the prompt.
- **The fun:** the moment a test you wrote catches something the tool missed
- **The skill:** explaining an AI-suggested function line by line, then testing it, before using any of it
- **The crumb:** Annotated code with tests

**How it goes.** Print or copy the suggestion into a scratch file, never straight into your project. Beside each line, write in plain words what it does. Any line you can’t explain gets a question mark and a trip to the official docs. Then write a small test table: one ordinary input, one empty input, one odd input a careless user might type. Predict each result, run the tests, and log any surprise on a Debug Log page. Keep only what you can explain and what passes, and say so on the whose-work card. The annotated page is worth filing; months from now it shows you exactly how much you understood.

## In-home — AI-06 Age Terms Map

- **Time:** 15 minutes (micro)
- **Materials:** a table with columns Tool · Minimum age · Teen rule · Where I read it · Date checked
- **Safety and privacy:** Read the company’s own help or terms pages; don’t create accounts to find out.
- **The fun:** finally knowing the actual rule instead of the playground rumor
- **The skill:** looking up current age terms for the tools your family uses, with check dates
- **The crumb:** Table with check dates

**How it goes.** List every AI tool anyone in the house uses or has asked about, including ones built into apps you already have. For each, find the official page that states the age rule; search the company’s help center for “minimum age.” Copy the rule in your own words and write the date. You’ll usually find a spread: some at 13, some needing a parent’s permission until 18, at least one at 18. File the table with your family AI agreement and recheck it when you revisit the agreement each year. Terms change more often than most families expect.

## What tools are genuinely good for

None of this means AI tools are useless to a learner. Used after an attempt, they're often good at a handful of jobs:

- **Explaining an error message** in plainer words, which you then confirm against the trace-back.
- **Suggesting edge cases** you hadn't thought to test. You decide whether they matter for your spec.
- **Explaining an unfamiliar line** of someone else's code, which you then check in the docs.
- **Quizzing you.** "Ask me five questions about Python dictionaries, one at a time, and don't give the answers until I've tried" turns a tool into retrieval practice.
- **Playing the confused user** of your program, so you can see where your instructions are unclear.

Notice what these share. In each, you stay the author and the judge. The tool supplies a question, a suggestion, or a second explanation, and your tests and your sources decide. That's the posture **The Computing Hour** has been building since the first page.

## Asking a narrow question

How you ask shapes what you get back, and the most useful questions are small. A wide question ("Write me a budget program") invites a finished program you didn't specify and can't explain. A narrow question keeps you the author. Compare a few pairs:

- Instead of "Fix my code," try "Here's the error message and the three lines it names. What does this error type usually mean?"
- Instead of "Write a function that averages a list," try "What edge cases should a function that averages a list handle?" Then write the function yourself.
- Instead of "Is my code good?," try "Is there any input that would make this function return the wrong answer? Show me the input, not a rewrite."
- Instead of "Explain Python dictionaries," try "Ask me three questions about dictionaries, one at a time, and tell me only whether I'm right."

Three habits make any question better. Paste your spec line first, so the tool knows what "right" means. Paste only practice code, never private data. And ask for one thing at a time; a tool asked five things at once tends to answer four of them loosely. When the answer arrives, it goes through the same gate as everything else in this chapter: explain it, test it, and log it.

## Diagnose this: the function nobody can explain

Here's an incorrect example. A learner's project includes this function, with a comment saying "from AI, works":

```
def average(nums):
    return sum(nums) / len(nums)

print(average([2, 4, 6]))    # 4.0
```

It does work, for that input. Before you read on, find the problem. Ask the edge-case question: what happens with an empty list? `average([])` divides by zero and crashes with a `ZeroDivisionError`. The function isn't wrong for its happy path; it's unfinished, and "from AI, works" told the learner not to look. The fix is a spec decision first ("what should the average of nothing be?"), then a test, then code. Tools tend to write the happy path. Your test table writes the rest.

## In-home — AI-05 Next-Word Table

- **Time:** 20 minutes (micro)
- **Materials:** paper, a pencil, a paragraph from a book or a nursery rhyme
- **Safety and privacy:** Public or made-up text only.
- **The fun:** generating gloriously silly sentences by following the table
- **The skill:** a word-pair table that predicts the next word, to see how text generators guess
- **The crumb:** Table

**How it goes.** Do the warm-up with a longer text: a nursery rhyme or a paragraph of a public-domain story. Build the two-column table of each word and the words that followed it. Then generate: pick a start word, roll a die or close your eyes to choose among the followers, and write ten words. Read the result aloud. It will sound a little like the original and say nothing true. Then ask the question that matters: "If this tiny table can sound right without knowing anything, how would I tell when a much bigger one does the same?" The answer is the rest of this chapter: check, test, and open the source.

## Out-of-home — OUT-04 Where Is AI Deciding?

- **Time:** 20 minutes (micro)
- **Materials:** a notebook, a normal errand
- **Safety and privacy:** Observe only; no photos of staff or customers; ask politely, accept "I don't know."
- **The fun:** seeing the hidden machinery in an ordinary afternoon
- **The skill:** spotting automated decisions in banks, stores, and offices; asking one staff question
- **The crumb:** List with one answer

**How it goes.** On your next errands, list every place where a computer might be making or shaping a decision: a store's suggested items, a bank's fraud alert, a self-checkout that calls for help, a phone menu that guesses why you're calling. Then ask one staff member one friendly question: "When the system flags something, does a person look at it?" Write down the answer, whatever it is. The point isn't to catch anyone out. It's to notice that automated decisions are everywhere, and that the useful question is always the same one you ask of your own code: who checks it?

## AI-07 Measure It Yourself

The two studies above disagree, and neither studied you. **AI-07 Measure It Yourself** is a deep session (about an hour) that settles the question for your own work. Pick two comparable small tasks from your project, of about the same size and kind. Do one with no AI help and one with

an assistant allowed. Time both. Then, over the next week, note any bugs you find later in each. Write a two-row table: task, minutes, bugs found later, how well you can explain it. One pair of tasks proves little. Three or four pairs across a month start to tell you something honest about your own speed and your own understanding.

## Talk to yourself (this chapter)

**Opening:** “What did I try first, and can I explain every line of what came back?”

**Follow-ups:** 1. **Prediction** — What did I write down, and what did I attempt, before I asked the tool? 2. **Evidence** — What exactly did the tool say, and what did the code do when I ran my tests? 3. **Location** — Which line, claim, or edge case is the first one I can’t explain or verify? 4. **Change** — What’s the one change I’ll make, and what do I expect my test to show? 5. **Revision** — Did my spec need a sentence I hadn’t written, so the tool couldn’t have known?

**How to wait:** keep the tool closed until the timer for your own attempt rings; read the suggestion twice before you run it; explain one line at a time, out loud.

**What a stuck silence usually means:** you can’t explain it—and that’s the answer.

**Don’t say / say instead:** Don’t say “The AI wrote it, so it’s probably right.” Say instead “I’ll test it and explain it before I keep any of it.”

## The whose-work card

Every project you file gets a short **AI-04 Whose Work Is It? Card**: what you did, what a tool did, and how you checked it. A labeled illustration: “Recipe scaler. All code mine. A chatbot explained one error message (logged Tuesday). Tests mine.” Or: “Word counter. My function kept; a tool’s suggested version tested and rejected.” The card takes two minutes. It keeps you honest with yourself, and it gives you the exact words to ask of your kids’ projects.

At work, the same honesty applies with one more step: check your employer’s rules before pasting any work code or data into a consumer AI tool. Many workplaces have them, and the rules exist for good reasons.

## Coaching your kids at every age

Your kids will meet AI tools whether or not you introduce them. Your job is to hold the device, hold the rule, and model the habits.

### Ages 5–10

**The move:** AI is a tool grown-ups use *with* the child, not a friend and not a babysitter. You hold the device. If you show a young child a chatbot at all, you type, you read the answer aloud, and you check one thing together.

**Exact words:** “This computer program is guessing what words come next. Sometimes it guesses wrong. Let’s check whether it’s right.”

**One thing to avoid:** letting a young child chat alone with any AI tool, or describing it as a person who knows them. At this age, the next-word table on paper teaches more than the real thing.

## Ages 11–14

**The move:** check the terms together with **AI-06 Age Terms Map**, and put a whose-work card on every project. Most consumer AI services set 13 as their minimum age, many require a parent’s permission until 18, and at least one major one sets 18.<sup>30</sup> Look up the ones your child wants to use, on the company’s own pages, and write the date you checked.

**Exact words:** “Before we sign up for anything, let’s read what the company says about age. And on every project, we write what you did and what the tool did.”

**One thing to avoid:** banning tools in secret or allowing them in secret. Either way, the child learns the rule is about not getting caught.

## Ages 15–18

**The move:** the ladder’s editor rule and the explain-it-back check. AI help built into the code editor stays off through age 16. At 17, it’s allowed on one named project, after the teen passes a Build From a Blank check, and it’s off during every check. Before you sign off on a project for their records, use **COACH-08 The Explain-It-Back Check**: three lines of their code, no screen, explained aloud. *Computing for Emerging Adults*, Chapter 8, sets out the teen’s side of this in full.

**Exact words:** “Walk me through these three lines without the screen. What would happen if this list were empty?”

**One thing to avoid:** asking whether they used AI as an accusation. Ask what the card says, then ask them to explain. Explanation settles the question better than suspicion.

## The family AI agreement (every age)

Tie it together with **FAM-01 The Family AI Agreement**, a desk session of about 30 minutes. You write it; teens co-sign from about 15; you revisit it once a year, or when a new tool arrives. Keep it to one page. A labeled illustration of the clauses many families use:

1. We try first. A tool comes in after our own attempt is on the page.
2. What a tool makes is made by a tool. We write “Help I used: ...” on every log and fill in a whose-work card on every project.
3. We can explain every line we keep, and it passes our tests.
4. No family photos, private messages, names, addresses, or school details go into AI tools.
5. We check the current age and privacy terms before anyone signs up, and grown-ups hold the account where the terms require a parent.
6. No companion or role-play chat apps for anyone under 18.

7. AI help is off for all checks and tests.
8. If a tool says something upsetting or asks for something personal, we stop and tell a grown-up.

Pair it with **FAM-03 Privacy Settings Tour**: for each family account on an AI tool, open the settings together and decide on memory, chat history, and whether conversations may be used for training. Write what you chose and the date. Settings move, so check again when you revisit the agreement.

## If it isn't clicking

**You're leaning on the tool too much.** You notice you open it before you've written a prediction. Go back to attempt first, timer on. Put the tool in another window, or another room, until the timer rings. Logging "Help I used" every time makes the habit visible fast.

**The tool's confidence keeps fooling you.** You keep keeping answers that turn out wrong. Run AI-02 on every answer for a week, even the ones that feel obvious. The point isn't distrust; it's calibration.

**The family can't agree on AI.** One parent is enthusiastic, the other wary, the teen exasperated by both. Do FAM-01 together. Writing the clauses as a group turns a running argument into a shared document, and the yearly review gives everyone a date to reopen it.

**When to get a person.** First run the rescue order from **The Computing Hour**: reread your spec, wait, hint, a worked example in a scratch file, then back to your own file. If you've spent two logged sessions unable to tell whether a tool's suggestion is right, take it to a human: a library coding group, a meetup, or a community-college lab. Explaining the problem to a person is also the best test of whether you understand it.

## Tools, including AI

This whole chapter is the AI chapter, and the AI box in **The Computing Hour** still holds. What's specific here: no tool is endorsed as safe, and the age rules, privacy settings, and features of every AI product change, sometimes several times a year. Check the current terms on the company's own pages, write the date, and recheck when you revisit the family agreement. Be wary of tools that claim to detect AI-written work; the explain-it-back check is more reliable than any detector.

## What "done enough" looks like

You're done enough with this chapter when you have:

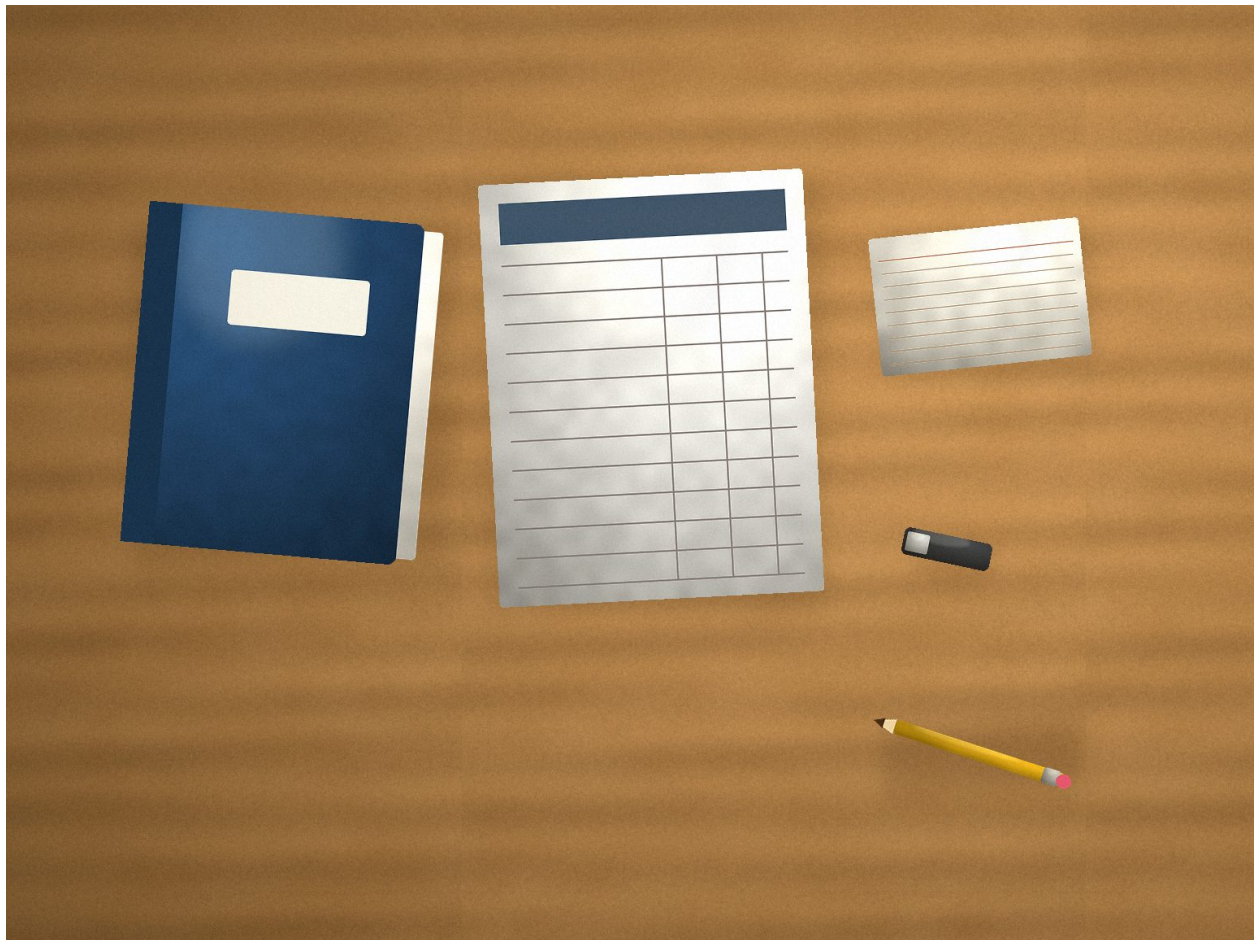
- ☐ an AI-03 annotated function: every line explained in writing, with at least two tests, one of them an edge case
- ☐ an AI-02 claim/check table with three rows
- ☐ an AI-06 age-terms table for your family's tools, with check dates

- ☐ a whose-work card on every project you've filed (AI-04)
- ☐ a signed family AI agreement (FAM-01) and a privacy-settings checklist (FAM-03)
- ☐ **AS-03 Build From a Blank:** a small spec built from an empty file with no tutorial and no AI, dated
- ☐ at least one AI-07 timing pair recorded, with your own note on what it showed

Placement by skill: if you already work AI-free by habit and keep honest logs, focus on the family agreement and the age-terms map, then move on to the records chapter.

# Chapter 9

## Records: Your Learning Log and Portfolio—and Your Teen’s Transcript



### Why keep records at all

Progress in programming is hard to feel. On any given evening, you’re stuck on something, and being stuck feels the same in month six as it did in week one. The difference is what you’re stuck

on. In week one it was a missing colon. In month six it's whether two tables should share a key. You won't notice that change from the inside. You'll notice it on paper.

That's the first reason to keep records: they show you progress you can't feel. A dated log, a short project list, and a few unaided checks across the months are the honest evidence that the work is working. On the evenings when it doesn't seem to be, open the log from three months ago and read one page. It helps more than any pep talk.

The second reason is your kids. Whatever you ask of their records, you can now show them from your own. A teen who sees a parent keep a plain, honest log—including “Help I used: a chatbot explained a `KeyError`”—learns what honest work looks like without a lecture.

The third reason is practical. If you ever take a community-college course, talk to someone who programs for a living, or apply for a role that involves code, a portfolio of small, real, explained projects says more than any line on a résumé.

One thing this book won't give you is a certificate, and that's on purpose. A certificate printed by the book you studied from proves only that you owned the book. Real credentials come from outside bodies, and this chapter explains which those are. What this book helps you build is better evidence: work you can show and explain.

## Your learning log

The learning log is the heart of your records, and it's small. **LOG-01 The Learning Log Page** takes about five minutes at the end of every Computing Hour. Each entry has six parts:

- **Date and length**, with the context: micro, desk, or deep.
- **What I specified**: the one-line spec for the session.
- **What I predicted**.
- **What happened**: what you built, and what the program actually did.
- **What I'd change next**.
- **Help I used**: people, docs, tools. “None” is a fine answer, and so is “a chatbot explained one error.”

You'll recognize these as **The Computing Hour**'s exit line written down. Here's a labeled illustration of a single entry:

Tue 6:40–7:05 p.m. (micro). Spec: count words in a file. Predicted 12; got 13 (trailing newline). One change: strip lines. Help I used: Python docs, `str.strip`.

That's a whole entry. It takes a minute, and it records what you did, what surprised you, and where the help came from. A notebook works well. So does a single text file named `log.txt` in your practice folder, committed to Git with your code, which gives every entry a date and a history.

Keep Debug Log pages with the learning log, not in a separate place. When you look back, the bugs are the most interesting part.

## When you miss a week

You will miss weeks. Illness, work deadlines, a family visit, a season when the kids need everything you've got. The log handles this without drama: write one line that says “no sessions, (dates), family” and carry on. A gap on the page is honest; it isn't a failure. Habit research suggests that one missed opportunity doesn't undo a habit that's forming,<sup>2</sup> and the log is how you notice the habit coming back.

When you return after a gap, don't start with the hardest project. Read your last three log pages, do one **AS-01 Cold Trace** to see where you are, and pick a micro session you can finish. The first entry after a break is the most important one to write.

## Paper or file?

Either works, and many people use both. Paper is fast, works anywhere, and is pleasant to flip through at the monthly look back. A text file in your practice folder sits next to your code, can be searched, and gets a history for free when you commit it with Git. If you choose paper, photograph the month's pages once a month and put the photos in your records folder, so the backup plan covers them. If you choose a file, keep it plain text, not a format that needs a particular app to open in five years.

## The monthly look back

Once a month, take 20 minutes for **PLAN-02 The Monthly Look Back**. Read the month's log pages and answer three questions on one page:

1. **What moved?** Which skills or projects actually got further?
2. **What stalled?** Where did the same problem show up across several sessions?
3. **What will I drop?** Every month, something should come off the list.

Add up the hours, roughly, and note how often “Help I used” named a tool. If that line has crept up, next month's plan is simple: more attempts first. If you find a project has stalled for weeks, **PLAN-03 Scope It Down** usually rescues it: cut it to a version that can ship in two weeks of sessions.

## Dated checks

**The Computing Hour's** checks belong in the log too, each with a date: **AS-01 Cold Trace** once a month (trace a short program and predict its output, unaided), **AS-02 Debug on the Spot** now and then, and **AS-03 Build From a Blank** when a chapter wraps up. These are the checks you do with no help and no AI. They're short, and over a season they make the clearest picture you'll have of what you can now do on your own.

## A year of records, at a glance

It helps to see what a year of honest records looks like, so you know you're not aiming at something enormous. A labeled illustration, for an adult doing two or three sessions a week:

- about 100 learning log entries, most of them three or four lines long
- 30 or 40 Debug Log pages, the best reading in the whole folder
- 12 monthly look backs, each one page
- 12 Cold Traces, dated, which together show your tracing getting faster and surer
- three or four Build From a Blank checks
- six to ten projects on the project list, each with a whose-work card
- four portfolio walkthroughs, each with a partner’s line of notes
- a signed family security rule and a family AI agreement, if you have kids at home

All of it fits in one folder. None of it took more than a few minutes at a time. And at the end of the year, it answers the question you couldn’t answer from the inside: *Am I actually getting better?*

## Your portfolio

Your log records sessions. Your portfolio records finished things. Keep it simple.

**LOG-02 Project List With Links** is a single page or file with one line per project: date started, the spec, where the code lives, status (working, paused, retired), and the whose-work card. A labeled illustration of one line:

Recipe scaler · started (month, year) · spec, tests, repository link (private) · Whose Work: all code mine; one error message explained by a chatbot, logged.

Every entry gets a **whose-work card** (AI-04): what you did, what a tool did, how you checked it. Keep it on the list itself, so nobody, including you in a year, has to wonder.

Keep repositories private by default. You can make one public later, deliberately, after you’ve checked it for names, addresses, passwords, and real family data. Practice data should always have been made up or public anyway.

## What belongs in it

Not every exercise. A good adult portfolio after a full pass through this book might hold six to ten small projects, each with a spec, tests, a readable Git history, and a short README that says what it does and how to run it. Good candidates come straight from the chapters: a script that automates one real chore (**CAREER-03 Automate One Chore**), a small data report built from a question you cared about, a web page that passes a keyboard-only check, a program with a test table you wrote before the code. Each is modest. Together they show the whole loop: specify, predict, build and test, debug, explain.

When you compare what you planned with what you shipped, write the difference down with **SPEC-05 Spec Diff**. A short “here’s what changed and why” note is one of the most adult things a portfolio can contain.

## A README worth reading

Every project folder gets a short README file, the first thing anyone, including you, reads when they open it. Four short parts are enough. A labeled illustration:

**Recipe scaler.** Scales a recipe’s ingredient amounts for a different number of servings.

**How to run it:** `python scaler.py recipe.txt 6` **What it does and doesn’t do:**

handles whole numbers and decimals; doesn’t convert units. **Whose work:** all code mine; tests mine; one error message explained by a chatbot (logged).

Writing the “doesn’t do” line is a quiet discipline. It keeps you honest about scope, and it heads off the most common question a first user asks.

## The seasonal walkthrough

Once a season, do **AS-05 Portfolio Walkthrough**. Present one project to a partner—a spouse, a friend, a teen, someone from a library coding group—and let them ask three questions: *Why did you design it this way? What broke? What would you change?* Ask them to write one line of notes. Answering out loud, without your screen, is the best test there is of whether the work is yours. It’s also good practice for any conversation about your work later.

## Honest credentials

It’s worth being plain about what counts as a credential and what doesn’t.

**What counts:** credit from an outside institution, such as a course at a community college or university that appears on its transcript; an exam given by an outside organization that sets and grades it; a degree. These are given by bodies that have their own standards and their own stake in meaning what they say.

**What also has real value, without being a credential:** your portfolio, your Git history, your log, and your ability to explain your work. For many purposes, from a hobby to a conversation with a working programmer, this is the evidence that matters most.

**What doesn’t count:** anything this book, or any book, could print for itself. There’s no “Computing Refresh I,” no completion certificate, and no seal at the end of this chapter. If you want to call your study something on a résumé, describe it plainly: “self-study in Python, testing, SQL, and Git, with a portfolio of small projects.” That’s honest, and it invites the right question: “Can you show me?”

A note on online courses. Many good online courses offer a completion badge or certificate of their own. Some are useful as structure, and a few are attached to real college credit through a partner institution; check exactly who grants what before you count it. Remember, too, that sign-ups for open online courses are far more common than completions.<sup>31</sup> That’s no reason to avoid them, but it’s a good reason to pair one with a log, a weekly time, and a person to report to. A course you finish with a portfolio project is worth more than three you started.

## Career reality, plainly

Some readers picked up this book wondering whether programming could become work. Here's an honest picture, with no hype in either direction.

The current federal projections show employment of software developers, quality-assurance analysts, and testers growing about 10 percent over the ten years they cover, roughly 185,000 more jobs, much faster than the average across all jobs. They also show about 106,000 openings a year on average over that decade, many of them from people retiring or changing fields rather than from new jobs.<sup>32</sup> That's a projection, not a promise. It describes a whole national labor market across a decade; it says nothing about any one person, town, or year, and projections are revised as the economy and the tools change. The typical entry-level education for the occupation is a bachelor's degree.

Coding bootcamps are a real second chance for some adults, and researchers who interviewed bootcamp students found both the opportunity and the cost. The coursework itself often ran a few months, but the full career change often took a year or more, and students described real personal and financial strain. The hurdles they named were ordinary ones: credentials, experience, portfolios, networking, and job interviews that ask candidates to solve problems at a whiteboard.<sup>33</sup> If you're considering that path, go in with your eyes open and your savings plan written down.

This book makes no forecast about AI and jobs. What it can say is that AI tools are changing daily programming work right now, and that the skills this book teaches—saying exactly what's wanted, checking what came back, debugging calmly, explaining your choices—are the skills that matter when a tool writes a first draft.

Three try-its help you test the idea against reality before you spend much on it:

- **CAREER-01 Ask a Working Programmer** (desk, about 30 minutes). Have a short conversation with someone who writes code at work. Ask one question about how they test, and one about how they keep learning. Write first names only in your notes.
- **CAREER-02 Read a Job Listing Honestly** (micro, about 20 minutes). Take one real listing and sort every requirement into three columns: *have*, *can learn*, and *not my path*. The sort is usually more encouraging, and more clarifying, than reading the listing whole.
- **CAREER-04 Paths Without Hype** (micro, about 20 minutes). On one page, compare a degree, community-college courses, self-study, and a bootcamp on what each asks of you in time, what evidence of outcomes you can actually find, and how each fits your life.

And whatever you decide about work, computing is a life skill first. A script that renames a thousand photos, a spreadsheet you finally trust, a household that knows how to restore a backup: those pay off whether or not you ever change careers.

Here's what that looks like in practice. **CAREER-03 Automate One Chore** is a deep session of about 75 minutes: pick something you do by hand every week, at home or at work (within your employer's rules), and write a small script that does it. Renaming and sorting a folder of scanned receipts. Combining three monthly spreadsheets into one. Checking a list of links to see which ones are broken. Time the chore by hand once, time it with the script, and write both numbers in the log. The minutes saved matter less than what you learned getting there, but the minutes are real,

and they come back every week.

## If you're homeschooling: your teen's records

If you're teaching computing to a high-schooler at home, keep their records the same way you keep yours. You already know the shape: dated sessions, a Debug Log, a project list, whose-work cards, and checks done without help. That's most of a transcript's evidence.

**LOG-03 Teen Hours and Credit Sheet** adds the pieces a transcript needs: hours, a course title, and a list of evidence.

### Hours and credit

A widely used homeschool guideline counts about 150 hours of work as one full credit for a core course, roughly five hours a week for 30 weeks. Logging more hours doesn't earn more credit; it only shows the threshold was passed. The same guideline suggests 120 or more hours for a full credit in an elective, and 60 or more for half a credit.<sup>34</sup> It's a guideline, not a law. This series plans on about 150 hours for a full computing credit, so the course holds up however it's classed. Check your own state's rules, because homeschool requirements differ from state to state.

The hours are easier to keep than they sound. If your teen writes a log line at the end of every Computing Hour, the hours are already there; once a month, add them up on the credit sheet. Count the work that's really part of the course: sessions, unplugged try-its, reading the docs, a library coding group. Don't count time spent watching other people code for fun, any more than you'd count watching a cooking show as a cooking credit.

### Plain course titles

Use plain, recognizable course titles, the kind a college admissions reader understands at a glance: **Computer Science, Computer Programming, Web Development, Cybersecurity, or Data Science**. Never a title with a book's name in it. A labeled illustration of one line on a credit sheet:

Computer Programming · 0.5 credit · 76 hours logged · evidence: Debug Log, repository, two Build From a Blank checks.

### A course description in plain words

Many families add a short course description for each computing course, a paragraph a college or employer can read in a minute. Write it last, from the records, not first from hopes. Four parts are enough: what the course covered, how the work was done, how it was checked, and the hours. A labeled illustration:

**Computer Programming (0.5 credit).** An introduction to programming in Python: writing short specifications, functions, files, lists and dictionaries, testing with written test tables, and debugging from error messages. Work was done in two or three sessions a week with a dated learning log and Debug Log, and kept in Git. Progress was checked

with unaided code-tracing and build-from-a-blank checks done without AI help, and with a spoken walkthrough of one project. AI tools were used only to explain error messages, and each use was logged. 76 hours.

Notice what it leaves out: grades the course never gave, claims about talent, and anything you can't show. Every sentence points to something in the folder. If a reader asks, "How do you know?", you can open the log, the repository, or the dated checks and show them. Keep each description to one paragraph, and update it at the end of the course rather than as you go. A teen who is old enough can draft it, and you can check it against the records. That's useful practice in describing their own work honestly, and it's the same skill the whose-work card has been building all along.

## Evidence

For each course, keep the kinds of evidence you keep for yourself: the Debug Log, a repository with a readable history, test tables, dated Build From a Blank and Cold Trace checks, and two or three projects with whose-work cards. Before you sign off, run **COACH-08 The Explain-It-Back Check**: three lines of their code, explained aloud with no screen. And use **COACH-04 Read the Log, Not the Screen**: read their Debug Log and ask one question about it. Your one question, written on the page, is itself evidence that someone checked.

## AP, dual enrollment, and the rest

Many homeschooled teens also take an AP exam or a community-college course while still in high school. Those choices, along with how outside grades and records work, are covered in full in *Computing for Emerging Adults*, Chapter 9. Read it there rather than here; it's written for exactly that decision.

## Keeping both the same way

You'll save yourself real effort if your records and your teen's look alike. Use one folder structure for both:

- log/ for learning log pages and Debug Log pages, by month
- projects/ with one folder per project, each holding a spec, tests, code, a README, and a whose-work card
- checks/ for dated Cold Trace, Debug on the Spot, Build From a Blank, and walkthrough notes
- records/ for the project list, the credit sheet, the signed family security rule, and the family AI agreement

Keep it on a family drive, not on a shared computer's desktop, and include it in the backup plan from Chapter 7. Then do the one step that makes it a real backup: restore one folder from it, once, and open a file.

A small habit helps both of you: file as you go. When a project reaches "working," take two minutes that evening to add its line to the project list and drop its whose-work card into the folder. Records

kept weekly take minutes. Records rebuilt from memory at the end of a year take whole weekends, and they're never quite as true.

If your teen is ahead of you, and many are by 16 or 17, the records are where you stay useful. You may not follow every line of their code, but you can read a Debug Log, ask why a test exists, check that the whose-work card is filled in, and sign the credit sheet knowing what's behind it. That's real oversight, and it's exactly what a transcript needs.

## You try this week

- **Tonight:** start **LOG-01**. Write one entry for your last session, even if it was days ago.
- **This week:** make **LOG-02** with one line, for the project you're working on now, including its whose-work card.
- **If you're homeschooling:** start **LOG-03** for your teen with this week's hours and one piece of evidence.
- **At the end of the month:** put **PLAN-02** on the calendar now, with a time.

# Chapter 10

## Resources



## How to choose

There's no best resource for learning to program, only the one that fits your week. Before you pick anything from this chapter, answer four questions on a scrap of paper.

**How much time do you really have?** Be honest about micro, desk, and deep sessions. A re-

source that assumes two free hours every evening will sit unused if your week is mostly 20-minute windows.

**How do you like to learn?** Some people want a book and a quiet desk. Some want short videos. Some learn best when someone else is in the room. None of these is the right way; each is a way.

**Do you want people around you?** Learning alone is flexible. Learning with others is steadier, because someone notices when you stop showing up. Many adults do best with both: solo sessions most of the week and a group once a month.

**How much load can you carry?** If you're teaching your kids too, your own learning has to fit around theirs. Pick fewer resources and use them well. One editor, one reference, one group, and this book are enough for a long time.

A word about evidence. A good deal is known about how adults learn and how beginners learn to program, and this book leans on it. But no study looks directly at parents learning to program in order to teach their own kids. The approach here borrows from what's known about adult learners, beginners, and learning with a partner, and the try-its are built so you can see for yourself, in your own log, whether it's working.

Everything below is listed by fit, not rank. Nothing is sponsored, and no prices appear. Where a resource is called free, it was checked on its own website on the day these notes were written; check again before you rely on it, because things change.<sup>35</sup>

## Free first

Start with what costs nothing, because much of the best material does.

**Your public library.** Libraries are quietly one of the best places to learn computing as an adult. Many offer public computers, quiet rooms, printing for your Debug Log pages, technology help sessions, and sometimes coding groups or classes. Many also give cardholders access to online learning services. Ask at the desk what your library has; offerings vary widely from place to place.

**The official Python tutorial and documentation.** The Python language's own website has a tutorial written by the people who make Python, and full reference documentation. It's freely available, and it's the source to check when anyone, person or tool, tells you what a function does.<sup>36</sup> Read the examples first; the explanations make more sense after you've seen the code.

**Python Tutor.** A free website that lets you paste in a short program and step through it line by line, watching every variable change in a picture as you go.<sup>3</sup> It's the closest thing to a trace table that draws itself, and it's especially good for the moments in Chapters 1 through 3 when a loop doesn't do what you predicted. Use it after you've traced by hand, to check yourself.

**CS Unplugged.** A free collection of activities that teach computer science ideas with cards, string, crayons, and games, no computer needed.<sup>37</sup> It was made for children, and it's wonderful for adults too, especially on evenings when a screen is the last thing you want. Many activities pair naturally with the sorting, searching, and data try-its in this book.

**Official security guidance.** The federal Cybersecurity and Infrastructure Security Agency publishes plain-language household advice under the name Secure Our World, built around four easy habits.<sup>23</sup> The National Institute of Standards and Technology publishes the detailed standard behind modern password advice.<sup>24</sup> Both are public. When an article or a tool gives you security advice that contradicts them, trust the official pages.

## Editors and practice tools

You need very little software to work through this book.

**Thonny.** A free, open-source Python editor built for beginners. Its website offers downloads that include Python itself, so there's one thing to install, and its step-by-step debugger lets you watch your program run one line at a time.<sup>12</sup> It's a gentle first editor, and plenty of people never need another for small projects.

**A browser, when you can't install anything.** On a library computer, a work laptop you aren't allowed to change, or a borrowed tablet, Python Tutor runs short Python programs right in the browser and draws each step.<sup>3</sup> It suits programs of a few dozen lines, not projects with files and folders. Save your code somewhere of your own before you close the tab, and keep any AI help it offers switched off until your own attempt is on the page.

**Any plain code editor.** Once you're comfortable, any editor that can run Python and open a terminal will do. Choose one that's well maintained and widely used, so answers to your questions are easy to find. Turn off any built-in AI suggestions while you're still learning a skill, and turn them on, if at all, the way Chapter 8 describes.

**SQLite.** A small database engine that runs on your own computer as a single file, with no server to set up. Its code is in the public domain and free for anyone to use.<sup>16</sup> It's the practice database for Chapter 4. Python can talk to it directly through a module that comes with Python, so you can practice SQL without installing anything else.

**Git.** The tool behind Chapter 6, free and open source.<sup>38</sup> Its website has the official documentation and a full book on using it. Pair it with a **private remote repository** on a hosting service, so your history lives in a second place. Check the service's terms; the most widely used one requires users to be at least 13.<sup>22</sup>

**A password manager and an authenticator app.** From Chapter 7. Choose by reputation and fit; protect the manager with your strongest passphrase and two-step login.

## Learning with people

Pairing up is one of the better-supported ideas in programming education (Chapter 1 has the details), and adults often say that having nobody to learn with is one of their biggest frustrations.<sup>7</sup> You don't need to join anything formal. You need someone to show your work to.

**Library coding groups and tech nights.** Many libraries host regular coding clubs, digital-skills workshops, or help sessions for adults. Some are aimed at beginners, some at hobbyists; both are

worth trying.

**Local meetups.** In many towns and cities there are informal groups for people who program, from beginners' study nights to groups focused on one language. They're usually welcoming to newcomers who say plainly, "I'm learning." Bring one small project and one question.

**Community-college continuing education.** Many community colleges offer non-credit continuing-education courses in programming, spreadsheets, and computer basics, often in the evening, alongside regular credit courses. If you want a real credential later, a credit course is the place to look (Chapter 9).

**Adult-education and lifelong-learning programs.** Many communities run adult-education programs through the school district or a local college, and some universities host lifelong-learning programs for older adults. Course offerings vary; check what's near you.

**A study partner.** One friend, a spouse, a sibling, or a co-op parent who's learning too. Meet for a desk session once a week; take turns being the one who explains. **OUT-03 Find Your People** in Chapter 1 is the try-it for this.

## Course types by fit

When you want more structure than a book, there are four broad kinds of course. None of them is right for everyone. Here's what each asks of you.

**Self-paced online courses.** Maximum flexibility; you start and stop whenever you like. The cost is that nobody notices when you stop, and many people do; completion rates for open online courses have been low for years.<sup>31</sup> If you choose one, pair it with your learning log, a fixed weekly time, and a person you report to.

**Cohort-based online courses.** A group starts together, meets on a schedule, and usually has an instructor or mentors. Less flexible, much steadier. Good when you know you need a deadline.

**Community-college courses.** In person or online, for credit or not, usually on a semester schedule. Real instructors, real classmates, and, for credit courses, a real transcript. Often the most underrated option for adults.

**Bootcamps.** Intensive, full-time or near it, over a few months, usually aimed at a career change. A genuine second chance for some people, with real personal cost and a timeline to a new job that's often longer than the course; Chapter 9 describes what the research found. Go in with your eyes open.

Whatever you choose, keep doing **The Computing Hour** inside it. Write the spec, predict before you run, keep the Debug Log, and fill in the whose-work card. A course gives you material; the Hour turns it into skill.

## A starter kit by context

If you'd rather not choose at all yet, here's a sensible default kit for each kind of session. Add things only when you feel the lack.

**Micro (15–25 minutes).** A notebook, a pencil, your learning log, and this book. Most micro sessions need no screen: a trace table, a spec, a test table, a next-word table, a phishing sort. When you do open the computer, open Python Tutor to check a trace, then close it.

**Desk (30–50 minutes).** Your editor, the official Python docs in one browser tab, your practice folder, and the Debug Log beside the keyboard. Keep any AI tool closed until your own attempt is on the page.

**Deep (60–90 minutes).** Everything from a desk session, plus a terminal, Git, and a project with a spec. Deep sessions are for building something whole, so protect them: phone in another room, a glass of water, and a stopping time written at the top of the page.

**On the road or at the library.** A library computer or a borrowed laptop, a browser-based editor or Python Tutor, and a way to save your work to your own drive. **OUT-02 Library Computer Session** in *The Computing Hour* walks through it, including the step people forget: logging out.

## Reading the docs without drowning

The official documentation is the most reliable resource in this chapter and, at first, the least friendly. A few habits help.

Read the examples before the explanation. Most documentation pages have a short code sample; type it, predict what it prints, and run it. The paragraph above it makes much more sense afterward.

Search for the exact name. If a function is called `str.strip`, search the docs for that, not for “remove spaces Python.” You'll land on the official page instead of somebody's opinion.

Read one entry, not the whole page. Reference pages are built to be dipped into. Find the one function you need, read its description and its parameters, and leave.

Write one line in your log about what you learned, in your own words. That line is what you'll remember next month.

## Special situations

Not every reader's week looks the same. A few situations come up often enough to name.

**Shift workers and caregivers.** If your time arrives in scraps, micro sessions on paper count fully. The log is the week: three ten-minute traces and a spec written in a waiting room is real progress, and it belongs on the page.

**Low vision and other access needs.** Most editors let you enlarge the font and change the colors; many work with screen readers and full keyboard navigation. The accessibility habits from Chapter

5 apply to your own setup as much as to the pages you build. Your library may also have assistive technology and staff who can help you set it up.

**One shared family computer.** Give each person their own user account if you can, keep the teen's portfolio and family photos on a family drive, and do the shared-computer check in Chapter 7.

**Learning alongside a teen who's ahead.** You're the first user and the spec reader. Ask them to teach you one thing a week. It's a strong check of their understanding, and it keeps you learning.

**No kids at home.** The coaching sections are optional. Many readers use them with a grandchild, a niece or nephew, or a friend's child at a library session.

**Learning for work.** Follow your employer's rules on which tools and data you may use, especially AI tools and anything involving real customer or company information. Practice on made-up data at home.

## For your kids, by rung

If you're coaching your kids, these are good starting points by age. Keep accounts for younger children offline or parent-managed.

**Ages 5–7: ScratchJr.** A picture-block programming app for young children, from the same family of projects as Scratch, in which kids snap blocks together to make characters move, talk, and play.<sup>13</sup> You sit beside them, ask “what will it do?”, and never take over the tablet.

**Ages 8 and up: Scratch.** A free block-based programming language and online community from MIT, where children make games, stories, and animations.<sup>13</sup> Use an offline editor or a parent-managed account for younger children, and check the community settings before you let a child share projects.

**All ages: CodeAI (formerly Code.org).** A nonprofit offering a free K–12 computer science curriculum, including short courses for young children and longer courses for older students.<sup>13</sup> CodeAI's elementary courses pair well with the unplugged try-its in this book's coaching sections.

**All ages: CS Unplugged.** The same activities listed above work beautifully with children; many were designed for them.

**The other books in this series.** Each one is built around the same Computing Hour, so your kids' sessions will feel familiar to you:

- ***Computing for Little Thinkers*** (ages 5–10). Use it when your child is starting out, when you want most sessions to be unplugged, and when you're the literal robot following their steps.
- ***Computing for Young Minds*** (ages 11–14). Use it when your child is moving from blocks to typed Python and is ready to keep a Debug Log of their own.
- ***Computing for Emerging Adults*** (ages 15–18). Use it when your teen is ready for a full high-school course, with Git, SQL, security on their own systems, and records for a transcript.

## Five questions to ask any resource

Before you commit time to a new course, book, app, or website, ask it five questions. If it can't answer them well, keep looking.

1. **Does it make me write code, or only watch?** Watching someone else program feels like learning and mostly isn't. Look for resources where you type, predict, and run within the first few minutes.
2. **Does it show worked examples before asking me to build alone?** Beginners learn faster from clear, guided examples than from being dropped into an open problem.<sup>10</sup> A good resource shows you a complete small program, asks you to predict and change it, and only then asks you to make your own.
3. **Does it test me, and space the practice out?** Quizzes, exercises, and returning to old topics are signs a resource takes learning seriously. A long run of videos with nothing to do between them is a warning sign.
4. **Is it honest about time?** Be wary of "learn to code in a weekend" or promises about jobs. Good resources say plainly how long things take and what you'll be able to do at the end.
5. **What does it do with my data and my kids' data?** For anything with an account, read the age terms and the privacy settings before you sign up, the same way Chapter 8 describes.

## If you're starting from zero

If you opened this book with no programming at all behind you, here's a calm order for your first three months, using only the resources in this chapter:

- **Month one.** This book's front matter and Chapter 1, mostly on paper. CS Unplugged for one or two evenings when you want variety. Install an editor at the end of the month.
- **Month two.** Chapters 2 and 3 with your editor, Python Tutor for checking traces, and the official tutorial's first sections as a second explanation. Start the learning log if you haven't.
- **Month three.** Chapter 4 with SQLite, one visit to a library group or meetup, and your first portfolio entry.

That pace is a suggestion, not a schedule. If it takes you twice as long, the log will still show the progress, and the progress is what counts.

## Keeping this list current

Resources change faster than books. Websites move, services close, terms change, and new tools appear every season. Keep your own short resource page in your records folder: what you use, the date you last checked its terms, and one line on why it fits you. Look it over at the monthly look back. When something on this chapter's list disappears or changes, your own page is the one that matters.

## Safety help

Keep these together, perhaps on the back of your Stop, Save, Tell card from Chapter 7.

**Everyday security.** CISA’s Secure Our World pages for household habits; NIST’s guidance for the detail behind passwords and two-step login.<sup>23,24</sup>

**Images shared without consent.** Since May 2026, the Federal Trade Commission has enforced a federal law requiring covered platforms to remove reported nonconsensual intimate images, real or computer-made, within 48 hours of a valid request. Report to the platform first; if it doesn’t act, the FTC takes complaints.<sup>25</sup>

**Images of anyone under 18.** The National Center for Missing & Exploited Children runs Take It Down, a service that helps remove such images, and the CyberTipline for reports of child exploitation.<sup>26</sup>

**Threats, extortion, or immediate danger.** Contact local law enforcement. In an emergency, call 911.

## You try this week

- Answer the four “how to choose” questions on one page and file it with your records.
- Visit your library’s website or desk and find one thing it offers that you didn’t know about.
- Open Python Tutor, paste in a three-line loop from Chapter 1, predict the output, and step through it.
- Put one date on your calendar to meet people who are learning too: a library group, a meetup, or a friend at your kitchen table.

# A Note on Sources

Studies, standards, government pages, documentation and platform terms named in the chapters are listed in Notes, in one series at the back, so the teaching pages can keep a plain, practical voice. When the same source comes up in several chapters, it has one note, and every mark that cites it carries that note's number.

Sources were re-read on 13 October 2026. A few pages refused automated reading that day, and the notes say how their wording was checked: the Federal Trade Commission's press release on the TAKE IT DOWN Act was read through a search engine's copy, and two age-terms help pages are cited from the research-stage reading. The high-school math study in the AI box comes from a different age and subject, and the book says so where it uses it. The two studies of AI and programming speed are preprints, and one was written by people affiliated with the tool's maker; the notes label both.

Some things rest on the best available evidence rather than a perfect match. Most research on learning to program studies students in courses, not adults at a kitchen table, and no study looks at parents learning in order to teach their own children; the book borrows from what is known about adult learners, beginners, and learning with a partner, and says so once. The research on transfer from programming to other subjects is mixed, so the book treats it as possible, not promised. Most of the pair-programming studies were college courses. The federal job projections are revised with each edition, which is why the book calls them a projection, not a promise, and leaves pay out entirely. HSLDA is an advocacy organization; its credit-hour figures are guidelines, not law.

The Python, SQL, web and Git samples were run to check their output, and an error message may be worded slightly differently on your screen. The five-step Hour, the session sizes, the try-its, the age rules for AI and accounts, and the other practice rules are this book's own suggestions, not research findings, and the people in the examples are illustrations, not real people. Program pages, platform terms, age rules and dates change; the date in each note says when it was read. Nothing here is legal advice. Access date for web sources: 13 October 2026.

# Notes

1. Anthony Robins, Janet Rountree, and Nathan Rountree, “Learning and Teaching Programming: A Review and Discussion,” *Computer Science Education* 13, no. 2 (2003): 137–72, <https://doi.org/10.1076/csed.13.2.137.14200>, accessed 13 October 2026. A review of novice-programming research; it distinguishes effective from ineffective novices mainly by their strategies, including tracing and testing.
2. Phillippa Lally et al., “How Are Habits Formed: Modelling Habit Formation in the Real World,” *European Journal of Social Psychology* 40, no. 6 (2010): 998–1009, <https://doi.org/10.1002/ejsp.674>, accessed 13 October 2026. Ninety-six participants; median time to automaticity about 66 days (range 18–254), with wide variation; missing one opportunity did not materially affect the process. Published online first in 2009.
3. Philip Guo, Python Tutor, <https://pythontutor.com/>, accessed 13 October 2026. The site describes itself as a “free online compiler and visual debugger” for Python, Java, C, C++, and JavaScript. It also offers optional AI help; the AI box in **The Computing Hour** applies to it like any other tool.
4. American Academy of Pediatrics, “Family Media Plan,” HealthyChildren.org, <https://www.healthychildren.org/English/fmp/Pages/MediaPlan.aspx>, accessed 13 October 2026.
5. Hamsa Bastani et al., “Generative AI without Guardrails Can Harm Learning: Evidence from High School Mathematics,” *Proceedings of the National Academy of Sciences* 122, no. 26 (2025), <https://doi.org/10.1073/pnas.2422633122>, accessed 13 October 2026. High-school mathematics, not programming, and not adults.
6. John Dunlosky et al., “Improving Students’ Learning with Effective Learning Techniques: Promising Directions from Cognitive and Educational Psychology,” *Psychological Science in the Public Interest* 14, no. 1 (2013), <https://doi.org/10.1177/1529100612453266>; Nicholas J. Cepeda et al., “Distributed Practice in Verbal Recall Tasks: A Review and Quantitative Synthesis,” *Psychological Bulletin* 132, no. 3 (2006), <https://doi.org/10.1037/0033-2909.132.3.354>; both accessed 13 October 2026.
7. Philip J. Guo, “Older Adults Learning Computer Programming: Motivations, Frustrations, and Design Opportunities,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (New York: ACM, 2017), <https://doi.org/10.1145/3025453.3025945>, accessed 13 October 2026. Survey of 504 respondents aged 60–85 in 52 countries, recruited through the Python Tutor website; 22 percent cited missed opportunities in youth as a motive; 11 percent named lack

of free time and 10 percent lack of contact with tutors or peers as frustrations. Self-selected, highly educated, STEM-heavy sample. The study did not look at parents learning in order to teach their own children.

8. Karthikeyan Umapathy and Albert D. Ritzhaupt, “A Meta-Analysis of Pair-Programming in Computer Programming Courses: Implications for Educational Practice,” *ACM Transactions on Computing Education* 17, no. 4 (2017), <https://doi.org/10.1145/2996201>, accessed 13 October 2026. Course settings.

9. Ronny Scherer, Fazilat Siddiq, and Bárbara Sánchez Viveros, “The Cognitive Benefits of Learning Computer Programming: A Meta-Analysis of Transfer Effects,” *Journal of Educational Psychology* 111, no. 5 (2019), <https://doi.org/10.1037/edu0000314>, accessed 13 October 2026.

10. Paul A. Kirschner, John Sweller, and Richard E. Clark, “Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching,” *Educational Psychologist* 41, no. 2 (2006), [https://doi.org/10.1207/s15326985ep4102\\_1](https://doi.org/10.1207/s15326985ep4102_1), accessed 13 October 2026.

11. Sue Sentance, Jane Waite, and Maria Kallia, “Teaching Computer Programming with PRIMM: A Sociocultural Perspective,” *Computer Science Education* 29, nos. 2–3 (2019), <https://doi.org/10.1080/08993408.2019.1608781>, accessed 13 October 2026. Developed and studied in school settings; adapted here for adult self-study.

12. Thonny, <https://thonny.org/>, accessed 13 October 2026. The site says Thonny “comes with Python ... built in, so just one simple installer is needed,” and describes its step-by-step debugger. Thonny is free and open source under the MIT License (Thonny source repository, LICENSE.txt, <https://github.com/thonny/thonny>, accessed 13 October 2026).

13. Scratch, MIT Media Lab and the Scratch Foundation, <https://scratch.mit.edu/>, accessed 13 October 2026; the site describes Scratch as “a free programming language and online community.” ScratchJr, <https://www.scratchjr.org/>, accessed 13 October 2026; its FAQ (last updated June 2026) lists versions for iPads, Android tablets, Fire tablets, and, with some limitations, Chromebooks and some Mac and Windows computers. The ScratchJr pages read that day did not state the price, so the book does not call it free. CodeAI (formerly Code.org), <https://code.org/>, accessed 13 October 2026; the homepage describes “free curriculum and tools.” The organization was renamed in June 2026.

14. Python Software Foundation, “Errors and Exceptions,” *The Python Tutorial*, <https://docs.python.org/3/tutorial/errors.html>, accessed 13 October 2026: “The last line of the error message indicates what happened.” Code samples in this chapter were run with Python 3 on 13 October 2026; tracebacks are shown without the caret lines that recent versions add.

15. Python Software Foundation, “csv — CSV File Reading and Writing,” *The Python Standard Library*, <https://docs.python.org/3/library/csv.html>, accessed 13 October 2026. All code and queries in this chapter were run on 13 October 2026 with Python 3 and SQLite on made-up data; tracebacks are shown without the caret lines recent versions add.

16. SQLite, <https://sqlite.org/>, accessed 13 October 2026, which describes SQLite as “a small, fast, self-contained, high-reliability, full-featured, SQL database engine”; SQLite, “SQLite Is Public Do-

main,” <https://sqlite.org/copyright.html>, accessed 13 October 2026; Python Software Foundation, “sqlite3 — DB-API 2.0 Interface for SQLite Databases,” <https://docs.python.org/3/library/sqlite3.html>, accessed 13 October 2026.

17. Python Software Foundation, “sqlite3 — DB-API 2.0 Interface for SQLite Databases,” <https://docs.python.org/3/library/sqlite3.html>, accessed 13 October 2026: “Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks.” See also the page’s how-to guide “How to use placeholders to bind values in SQL queries.”

18. MDN Web Docs, <https://developer.mozilla.org/>, and “<meta name=“viewport”> HTML attribute value,” <https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/meta/name/viewport>, both accessed 13 October 2026. MDN recommends `width=device-width`; the `initial-scale=1` used in this chapter is a common, harmless addition. Browser developer tools and console messages vary by browser and version; the wording quoted in this chapter was observed in a current Chromium-based browser on 13 October 2026, where all the chapter’s HTML, CSS, and JavaScript were run.

19. WebAIM, “Screen Reader User Survey #10 Results,” <https://webaim.org/projects/screenreadersurvey10/>, accessed 13 October 2026: of 1,539 respondents surveyed in December 2023 and January 2024, 71.6 percent said that on a lengthy page they would first navigate through the headings. World Wide Web Consortium, *Web Content Accessibility Guidelines (WCAG) 2.2*, W3C Recommendation, 12 December 2024, <https://www.w3.org/TR/WCAG22/>, accessed 13 October 2026. The checks in this chapter are plain good practice, not a WCAG conformance audit.

20. Git for Windows, <https://gitforwindows.org/>, accessed 13 October 2026; the site describes Git BASH, “a BASH emulation used to run Git from the command line.” Microsoft, “Using aliases,” PowerShell documentation, <https://learn.microsoft.com/en-us/powershell/scripting/learn/shell/using-aliases>, accessed 13 October 2026, lists Windows PowerShell aliases including `ls`, `cd`, `pwd`, `cp`, `mv`, `rm`, and `cat`; they run PowerShell’s own commands, so options can differ. All commands and outputs in this chapter were run on 13 October 2026 in a practice folder on a Linux computer with Git and Python 3; commit IDs will differ on your computer.

21. Git, “Documentation,” <https://git-scm.com/doc>, with the reference pages “git-restore — Restore working tree files,” <https://git-scm.com/docs/git-restore>, and “git-show — Show various types of objects,” <https://git-scm.com/docs/git-show>, all accessed 13 October 2026.

22. GitHub, “GitHub Terms of Service,” <https://docs.github.com/en/site-policy/github-terms/github-terms-of-service>, accessed 13 October 2026: “A User must be at least 13 years of age.”

23. Cybersecurity and Infrastructure Security Agency, “Secure Our World,” <https://www.cisa.gov/secure-our-world>, accessed 13 October 2026. The campaign’s “four easy ways to stay safe online” are: recognize and report phishing, use strong passwords, turn on multifactor authentication, and update software.

24. National Institute of Standards and Technology, *Digital Identity Guidelines: Authentication and Authenticator Management*, NIST SP 800-63B-4, <https://pages.nist.gov/800-63-4/sp800-63b.html>, accessed 13 October 2026. Passwords used as a single factor must be at least 15 characters; passwords used with another factor at least 8; verifiers should permit at least 64 characters and

should accept spaces and printing characters. The standard governs services, not households; the book uses it as the best current guide to what “strong” means.

25. Federal Trade Commission, “FTC Begins Enforcing the TAKE IT DOWN Act,” press release, 19 May 2026, <https://www.ftc.gov/news-events/news/press-releases/2026/05/ftc-begins-enforcing-take-it-down-act>, accessed 13 October 2026 through a search engine’s copy (the page refused automated reading that day); and FTC Consumer Advice, “FTC Enforces Compliance With the Take It Down Act,” 19 May 2026, <https://consumer.ftc.gov/media/ftc-enforces-compliance-take-it-down-act>, accessed 13 October 2026. Covered platforms must remove reported nonconsensual intimate images, real or fake, and known identical copies within 48 hours of a valid request; complaints about platforms go to [TakeItDown.ftc.gov](https://TakeItDown.ftc.gov). The FTC began enforcing the law on 19 May 2026.

26. National Center for Missing & Exploited Children, “Take It Down,” <https://takeitdown.ncmec.org/>, accessed 13 October 2026. The site describes Take It Down as “a free service” that can help remove or stop the online sharing of nude, partially nude, or sexually explicit images or videos taken of someone under 18, and says users can remain anonymous and do not have to send the images to anyone. Reports of child sexual exploitation can also be made to NCMEC’s CyberTipline, <https://report.cybertip.org/>, accessed 13 October 2026.

27. Stack Overflow, “2025 Developer Survey: AI,” <https://survey.stackoverflow.co/2025/>, accessed 13 October 2026. Industry survey, self-selected respondents: 84% reported using or planning to use AI tools in their development process; 46% said they distrust the accuracy of AI tools versus 33% who trust it (3% “highly trust”); 66% named “AI solutions that are almost right, but not quite” as their top frustration; 45% said debugging AI-generated code is more time-consuming.

28. Sida Peng et al., “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot,” arXiv:2302.06590 (2023), <https://arxiv.org/abs/2302.06590>, accessed 13 October 2026. Preprint; one lab task (implementing an HTTP server in JavaScript); the treatment group “completed the task 55.8% faster than the control group”; authors affiliated with the tool’s maker. Joel Becker et al. (METR), “Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity,” arXiv:2507.09089 (2025), <https://arxiv.org/abs/2507.09089>, accessed 13 October 2026. Preprint; randomized controlled trial with experienced developers on repositories they knew well; with AI tools allowed, completion time rose 19%, while developers had forecast a 24% speedup and afterward estimated a 20% speedup. Neither study tested adult beginners.

29. James Prather et al., “The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers,” in *Proceedings of the 2024 ACM Conference on International Computing Education Research (ICER ’24)*, vol. 1 (New York: ACM, 2024), 469–86, <https://doi.org/10.1145/3632620.3671116>; full text read in the authors’ preprint, arXiv:2405.17739, <https://arxiv.org/abs/2405.17739>, accessed 13 October 2026. Twenty-one lab sessions with university novice programmers (observation, interviews, and eye tracking). Students who accelerated used AI to write code they already intended to write and ignored unhelpful suggestions; students who struggled were often led along by the tool, thought they performed better than they did, and “finished with an illusion of competence.” For background, see Paul Denny et al., “Computing Education in the Era of Generative AI,” *Communications of the ACM* 67, no. 2 (2024), <https://doi.org/10.1145/3624720>.

30. Age terms as checked on 13 October 2026; check the current terms before use. OpenAI, “Terms of Use,” effective 1 January 2026, <https://openai.com/policies/row-terms-of-use/>, accessed 13 October 2026: “You must be at least 13 years old or the minimum age required in your country to consent to use the Services. If you are under 18 you must have your parent or legal guardian’s permission.” Anthropic, “Minimum age requirement access restriction,” Help Center, 16 March 2026, <https://support.claude.com/en/articles/13117299-minimum-age-requirement-access-restriction> (18 or older for consumer accounts; read at the research stage). Google, “Guide your child’s Gemini Apps experience,” <https://support.google.com/gemini/answer/16109150> (parents can give a child under 13 access through Family Link; read at the research stage). Microsoft, “Microsoft Copilot Supplemental Terms of Use,” effective 18 August 2026, <https://www.microsoft.com/en-us/microsoft-copilot/for-individuals/terms-of-use>, accessed 13 October 2026: users must be “at least 13, but sometimes older, depending on your country’s laws,” and some features may be turned off for users under 18. GitHub, “GitHub Terms of Service,” <https://docs.github.com/en/site-policy/github-terms/github-terms-of-service> (users must be at least 13).

31. Justin Reich and José A. Ruipérez-Valiente, “The MOOC Pivot,” *Science* 363, no. 6423 (2019): 130–31, <https://doi.org/10.1126/science.aav7958>, accessed 13 October 2026. Completion and persistence rates in open online courses were low.

32. U.S. Bureau of Labor Statistics, *Occupational Outlook Handbook*, “Software Developers, Quality Assurance Analysts, and Testers,” <https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm> (page last modified 27 August 2026), accessed 13 October 2026. Projections for 2025–35: employment of software developers, quality assurance analysts, and testers projected to grow 10 percent (about 185,400 jobs), “much faster than the average for all occupations”; about 106,100 openings projected each year on average; typical entry-level education, a bachelor’s degree. An earlier edition (2024–34) gave higher figures, which is one reason the book calls these projections, not promises. BLS also publishes pay data; this book leaves it out on purpose.

33. Kyle Thayer and Amy J. Ko, “Barriers Faced by Coding Bootcamp Students,” in *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER ’17)* (New York: ACM, 2017), 245–53, <https://doi.org/10.1145/3105726.3106176>, accessed 13 October 2026. Interview study of 26 participants; describes bootcamps as a second chance with great personal costs and risks, and names hiring barriers including credentials, experience, portfolios, networking, and whiteboard interviews.

34. HSLDA, “3 Ways to Determine My Teen’s High School Course Credit,” <https://hsllda.org/post/3-ways-to-determine-my-teens-high-school-course-credit>, accessed 13 October 2026: “at least 150 hours for 1.0 credit (roughly five hours a week for 30 weeks).” A guideline from an advocacy organization, not a legal requirement; state rules vary. For a core course (English, science, history, math, or foreign language) HSLDA says to log at least 150 hours; for an elective course, 120 or more hours for 1.0 credit and 60 or more for 0.5.

35. Each resource called free in this chapter was checked on its own website on 13 October 2026. Prices, terms, and offerings change; check the current page before relying on any of them.

36. Python Software Foundation, “The Python Tutorial,” <https://docs.python.org/3/tutorial/>, accessed 13 October 2026. The documentation site describes Python and its standard library as

freely available and freely distributed.

37. CS Unplugged, <https://www.csunplugged.org/>, accessed 13 October 2026. The site describes its activities as free teaching material.

38. Git, <https://git-scm.com/>, accessed 13 October 2026. The site describes Git as free and open source.