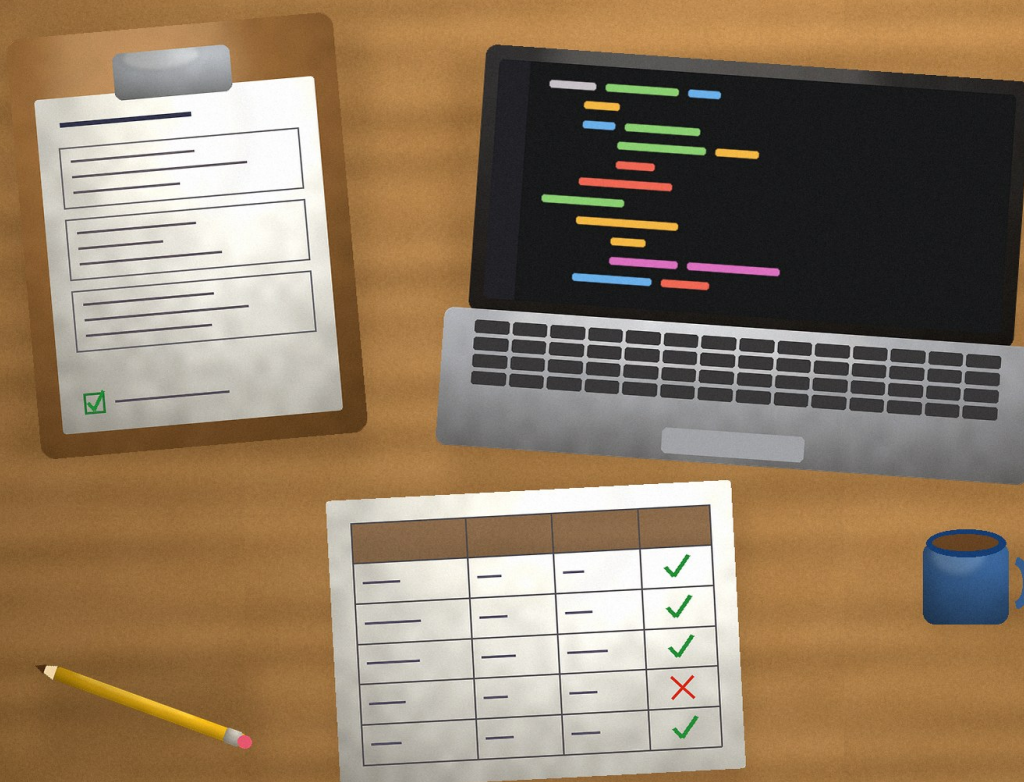


COMPUTING FOR EMERGING ADULTS

Michael Gannotti



Computing for Emerging Adults

Michael Gannotti

A homeschool guide to computing for ages 15–18: specs, Python and Java, testing, data, the web and Git, security, AI, and the transcript. Book 3 of the SMF WisdomForge Computing series. Illustrated edition, 12 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 — What Computing Looks Like at 15–18: Specs, Algorithms, and Running Your Own Work
- Chapter 2 — Python as a Tool: Programs That Read, Compute, and Write
- Chapter 3 — Testing and Debugging: Proof Before Pride
- Chapter 4 — Objects and Classes: The Bridge to Java
- Chapter 5 — Data: Questions, Tables, and Databases
- Chapter 6 — Building for People: The Web, the Command Line, and Git
- Chapter 7 — Systems, Networks, and Security: What You May and May Not Touch
- Chapter 8 — Working With AI: How It Learns, Where It’s Wrong, and Whose Work It Is
- Chapter 9 — Credit and the Transcript: Records, Portfolio, AP, and Dual Enrollment
- Chapter 10 — Resources
- A Note on Sources
- Notes

Welcome

Here is a picture. It is an illustration, not a real family.

A sixteen-year-old's spec page sits on the kitchen table beside a laptop. It is one sheet, handwritten, with a coffee ring in the corner. At the top it says what the program is for: a tool that totals the pages in a reading log. Below that come the inputs, the outputs, three examples, and a last line that says, "Done when all three examples pass and Mom can use it without asking me anything."

On the screen, a test has just failed. The third example, the one with a blank line at the end of the file, crashed the program.

In a notebook beside the laptop, the student has already written the Debug Log line: "Expected 817. Actual: ValueError on the last line. Guess: the blank line has no comma. Try: skip empty lines." Nobody has touched the code yet. Nobody is upset. The parent, who does not program, is reading the log and asking one question: "How will you know the fix worked?"

That table is what this book is for.

Who this book is for

This book is for the parent of a fifteen- to eighteen-year-old who wants a real computing year at home. You may write code for a living. You may never have opened a terminal. Either way, your job in these pages is the same, and it is one you can do well: you ask good questions, you use the program like a stranger would, you read the log, you keep the safety rules, and you sign the records.

It is also for the student, who will run most of it. At this age, computing becomes your own work. You decide what to build, you write down what it must do, you run the session, you keep the history, and you defend the result. Every chapter has a section written to you directly. Read it on your own. It assumes you are capable, because you are.

Many adults feel unsure about code. That is fine, and it changes faster than you'd think. You do not need to be ahead of your student. You need to be curious beside them.

The line the whole book runs on

Every chapter, every session, and every check in this book comes back to one sentence:

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

Specify it before you build it means a short written spec (a description of what the program must do, with examples and a line saying what “done” means) comes before any code.

Predict before you run means your student writes down what the program will do before pressing run, and then compares.

Debug before you rewrite means red text is a clue, not a verdict. Your student reads it, writes what they expected and what actually happened, changes one thing, and logs it. Starting over is a decision made after evidence, not a reaction to panic.

Explain it before you call it yours means ownership is tested out loud. If your student can’t explain a line, it isn’t theirs yet, however it was made. That matters more than ever now that a chatbot can produce a screenful of code in seconds. What a tool makes is made by a tool.

What your student will learn

Over a year, your student will learn to write a spec and build to it in Python, then in Java. They will prove their programs work with tests, debug calmly from evidence, and keep every change in Git (a tool that records the history of a project). They will ask honest questions of data with a real database, build a web page a real person can use with a keyboard alone, and work in the terminal like a professional. They will learn how computers and networks work and how to protect their own systems ethically, only on systems they own. They will learn how AI systems learn, where they go wrong, and how to use them without handing over ownership of their work.

And they will turn all of it into a record that a stranger can read: hours, artifacts, checks, a portfolio, and a transcript line.

You are not alone

Homeschooling is ordinary now. About 3.4 percent of American students in kindergarten through grade 12 were homeschooled in the 2022–23 school year, well over a million and a half students.¹ Families teaching high-school computing at home are part of that number. Co-ops, libraries, community colleges, and working programmers down the street are all part of how they do it, and this book shows you how to use each one.

What this book will not do

It will not hand you a 180-day script. It gives you a session that works, a week to start with, and a bank of 100 activities to build from. You choose.

It will not promise an AP score, a college admission, or a job. It explains how AP and dual enrollment work for homeschoolers, and it treats both as options.

It will not promise that coding sharpens every other subject. Learning to program teaches programming, and that is plenty. Other benefits are possibilities, not promises, and the book says so once and then gets on with teaching.

It will not sell you a curriculum, rank products, list prices, or issue certificates. It will not invent a course title with its name on it. Your student's transcript will say something a registrar recognizes, such as "Computer Science."

It will not reprint an earlier book. Everything here is written for fifteen- to eighteen-year-olds.

Where this book sits

This is the third title in the SMF WisdomForge Computing series. *Computing for Little Thinkers* covers ages 5 to 10, and *Computing for Young Minds* covers 11 to 14. After this book comes *Computing Refresh for Adults*, for the parent who'd like their own turn.

A student who is new to the series starts at Chapter 1 and is fine. A student coming from *Young Minds* should still start at Chapter 1, because specs are new, and can then move quickly through Chapter 2.

Where to begin

Turn to "How to Use This Book This Week." It gives you five days, starting Monday. Then read **The Computing Hour** together, out loud if you like. It is the session this whole book runs on, and your student will be leading it by Friday.

Find a pencil, a stack of index cards, and one clean sheet of paper. That sheet is going to become a spec.

— Michael Gannotti

How to Use This Book This Week

You can start on Monday. You don't need to read the whole book first, and you don't need to buy anything. You need a computer that can run Python (a shared family computer is fine), a notebook for the Debug Log, a stack of index cards, and about four hours across the week.

This week

Day	What you do	Cards	Time
Monday	Read The Computing Hour together. Sign the family security rule and the AI rule. Set up the folder plan and two places. Write and initial the Monday plan (15–16).	SEC-01, SYS-03, PLAN-01	45–60 min
Tuesday	First Hour: a tiny program from a one-page spec. Predict the output before every run.	SPEC-01, PY-02, BUG-01	50–70 min
Wednesday	Unplugged: two ways to search a stack of cards, with step counts. A short Cold Trace for a baseline.	ALG-01, AS-01	30–40 min
Thursday	Second Hour. You are first user.	SPEC-02, TEST-05, BUG-05	50–70 min
Friday	Ten-minute audit. Read the Debug Log together. Choose next week's goal.	PLAN-02	15 min

Monday. Read **The Computing Hour** out loud, taking turns. Then sign two short pages. The first

is the family security rule (SEC-01): your student practices security only on systems they own and toy apps they built, never anyone else's, even "to help." Chapter 7 teaches it in full, but it goes on the wall in week one. The second is the AI rule from the AI box in **The Computing Hour**, copied in your student's handwriting. Then make the folder plan: one folder for the course, one subfolder per project, and a second copy somewhere else, such as a USB stick or a family drive. That is "two places" (SYS-03). Finish with the Monday plan (PLAN-01): this week's goal, the first spec, and the one risk to watch. At 15–16 you initial it.

Tuesday. The first Hour. Your student writes a one-page spec (SPEC-01) for something tiny: a tip calculator, a reading-log totaler, a unit converter. Then they predict what the first run will print (PY-02), run it, and compare. When something surprises them, they write the Expected/Actual line (BUG-01) before touching the code. Use the step names out loud: Specify, Predict, Build and Test, Debug and Log, Review, Ship and Explain.

Wednesday. No screen. Sixteen numbered cards, two ways to find one of them, and a tally of the flips (ALG-01). Then a ten-minute Cold Trace (AS-01): a short program on paper, traced without help, with the predicted output written at the bottom. Date it and file it. It is not a grade. It is the "before" picture you'll compare against all year.

Thursday. The second Hour, on the same program or a new one. This time your student adds examples to the spec before writing code (SPEC-02), and you are the first user (TEST-05): you use the program like a stranger would and report what happened, without fixing anything. Your student logs what you found in the Debug Log (BUG-05).

Friday. Fifteen minutes. Read the week's Debug Log together. Count the hours. Ask, "What would you do differently next week?" and write down the answer. Choose next week's goal (PLAN-02).

The two bands

Inside each chapter, the book speaks of two bands, **15–16** and **17–18**. Place by skill, not birthday.

At 15–16, you're in the room for the Hour. You initial Monday's plan, ask one question about the spec, and act as first user every session.

At 17–18, your student hands you or emails you the plan on Monday, runs the Hour alone, and you read the Debug Log the same day. You're first user once a week.

Some cards are marked **ready 16**. Those are 17–18 moves that a sixteen-year-old with the skill can take. A seventeen-year-old who is new to programming can run the whole year at the 15–16 pace, and that is fine.

How each chapter is built

Every teaching chapter (Chapters 1 through 8) uses the same eight headings, in the same order:

1. **Why this matters** — about a page on what the chapter is for.

2. **For the parent: understand it yourself** — the idea in plain words, then precisely; one standard quoted; the wrong answers you should learn to hear; a five-minute warm-up for you.
3. **How to teach it this week** — the session shape, exact sentences to try, and the first activity.
4. **Practice that actually builds learning** — named try-its at home and out of the house, the talk box, and one wrong example to diagnose.
5. **For the student** — a longer section written to your student.
6. **If it isn't clicking** — what you see, what it usually means, and what to try next.
7. **Tools, including AI** — what's different in this chapter, pointing back to the AI box.
8. **What “done enough” looks like** — a checklist, including the checks.

Reading a try-it box

Each activity has a heading such as *In-home — SPEC-01 The One-Page Spec* or *Out-of-home — OUT-01 Algorithms in Public*. The code matches the activity bank. Inside the box you'll find the **time**, the **materials**, a **safety and privacy** line, **the fun**, **the skill**, and **the crumb**, which is what to save for the record. Then comes “How it goes,” with band notes where they matter.

About 44 of the book's 100 activities need no screen at all. That is on purpose. Specs, traces, tests, diagrams, and threat models are paper work at every age, including for professionals.

Reading a talk box

Each teaching chapter has one talk box. It gives you an opening question with no answer to recite, five follow-ups, how to wait, what a stuck silence usually means, and one “don't say / say instead.” The shape is explained once in **The Computing Hour**.

Where the checks live

The book's seven checks (AS-01 through AS-07) appear in each chapter's “done enough” list and gather in Chapter 9 with the portfolio. All of them are done without AI. None of them is a test battery or a percentile. They are dated pieces of evidence that your student can do the work alone.

What to save from week one

- The signed family security rule and AI rule
- The initialed Monday plan
- The first spec, with any marks you made on it
- The Wednesday Cold Trace, dated
- The first Debug Log page, with minutes for each day
- The program file, saved in two places

That is the beginning of a transcript. Chapter 9 shows how it grows.

Low-device homes

One shared computer works. Siblings can stagger Hours or pair up (Chapter 1 shows how). Chapters 1 through 5 run fully offline. A whole Computing Hour can also run at a public library (OUT-02, in **The Computing Hour**), which is how many families handle publishing in Chapter 6.

When life happens

A week with a move, a new baby, or a hard diagnosis can be a week of index cards and Cold Traces. Pick up where you left off. The Debug Log will remember where that was.

If You Only Remember Five Things

If the week goes sideways and you read nothing else, keep these five. The rest of the book is these five, done carefully.

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

1. Specify it before you build it.

Before any code, your student writes a short spec: what the program is for, who uses it, what goes in, what comes out, the rules, two or three examples (one of them an odd or boundary input), and one line that says what “done” means. One page is plenty. A spec is a tool, not paperwork, and it changes as your student learns. If there’s no done line, nobody can say whether the work is finished, including your student.

2. Predict before you run; debug before you rewrite.

No run without a written prediction. When the program surprises them, your student writes Expected/Actual, makes a guess, changes one thing, runs it again, and writes a line in the Debug Log. Red text is a clue with a line number. Starting over is allowed, but only after the log says why.

3. Explain it before you call it yours.

Every Hour ends with the exit line: “I specified... I predicted... It did... Next I’d change...” and then “Help I used: ...”, said every time, even when the answer is “none.” A tutorial, a sibling, the documentation, you, an AI tool: all of it counts, and none of it is shameful. If your student can’t explain a line, it isn’t theirs yet. AI output is made by a tool, never filed as your student’s work.

4. Tests, two places, and a Debug Log are the record.

From Chapter 3 on, a project without tests isn’t done. Every file lives in two places. The Debug Log carries the minutes for each day, and the minutes become hours. About 150 hours makes one full credit, and Chapter 9 turns the record into a transcript line a stranger can read.

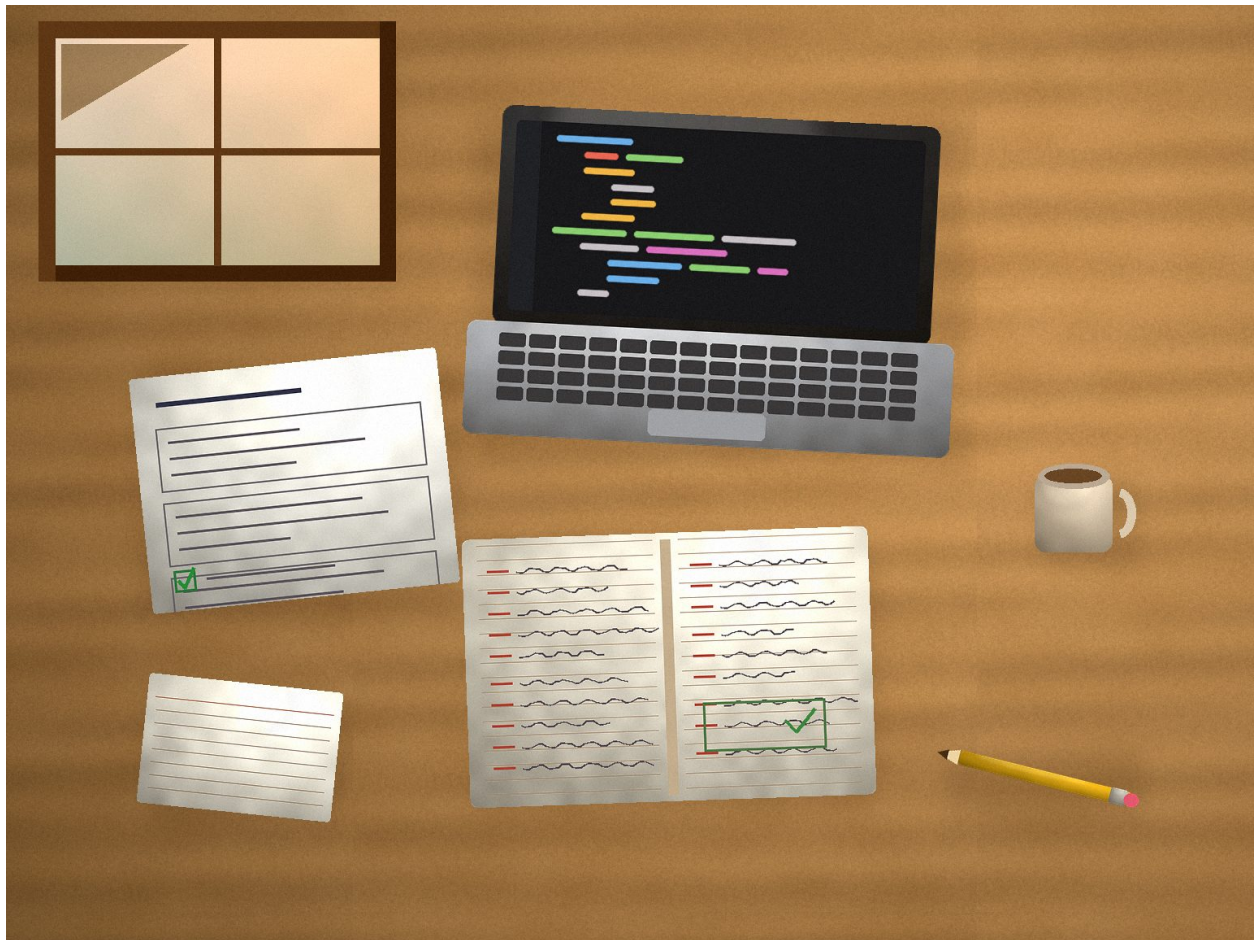
5. Security practice only on your own systems; AI coding help in the editor off through 16.

Your student practices security only on systems they own and toy apps they built, under a signed family security rule. Never anyone else's system, even "to help," even with a friend's say-so.

AI coding assistants inside the editor stay off through age 16. At 17, your student may turn one on for one named project, after passing a closed-book Build From a Blank (AS-04). AI is off for every check, at every age. The full rule is in the AI box in **The Computing Hour**.

That's it. Specify, predict, debug, explain, and keep honest records. You can do this.

The Computing Hour



The Computing Hour is the session this whole book runs on. It has six steps, the same six every time, and your student runs them. Learn it this week, and the rest of the book will make sense.

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

The six steps

Step	15–16	17–18	What the student does	What the parent does
Specify	5–10 min	5–10 min	Writes or updates today’s spec: goal, inputs and outputs, two examples, the done line	15–16: reads it and asks one question; initials Monday’s plan
Predict	2–5 min	2–5 min	Writes what the next run will do	Asks “Why?” and waits
Build and Test	20–35 min	30–50 min	Writes code in small steps; runs the tests after each change	Stays off the keyboard; stays available
Debug and Log	5–15 min	5–15 min	Expected/Actual, a guess, one change, a Debug Log line	Uses the rescue order, only if asked
Review	5–10 min	5–10 min	Rereads the change against the spec; saves with a real message; notes the next test	15–16: first user every session
Ship and Explain	3–5 min	3–5 min	Says or writes the exit line	Listens; 17–18: reads the log the same day

At 15–16 the Hour runs about 50 to 70 minutes with you in the room. At 17–18 it runs about 60 to 90 minutes, and you read the log the same day. Project days may run longer with a movement break in the middle. Stop while it’s going well; a passing test is a fine place to end.

Specify. The spec is a short written description of what the program must do, with examples and a line that says what “done” means. Later in a project, this step is a two-minute update. Chapter 1 teaches specs in full.

Predict. Before every run, your student writes one line: “I predict it will print...” (PY-02). A wrong prediction is the most useful thing that can happen in an Hour; it shows exactly where your student’s picture differs from the program.

Build and Test. Small steps: a few lines, a run, a check against the spec’s examples. From Chapter 3 on, that means real tests (written checks that compare what the program does with what it should do).

Debug and Log. When something surprises your student, the first move is a pencil: Ex-

pected/Actual, a guess, one change, a rerun, a Debug Log line.

Review. Your student rereads what changed today and holds it up against the spec. They save the work with a message that says what changed, such as “Format money to two decimals” rather than “stuff.” (In Chapter 6 that save becomes a Git commit.) They note the next test to write.

Ship and Explain. The Hour ends with the exit line, said aloud or written in the log:

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

and then

“Help I used: ...”

“Help I used” is said every time, even when the answer is “none.” A tutorial, a sibling, the documentation, you, an AI tool: all of it counts, and none of it is shameful.

Here’s how one Hour might go

This is an illustration, a composite, not a real family.

It’s Monday. A fifteen-year-old’s spec for a tip calculator says: put in a bill and a percent; get out the tip and the total, in dollars and cents. Examples: a bill of 40.00 at 20 percent gives a tip of 8.00 and a total of 48.00; a bill of 0 at 20 percent gives 0.00 and 0.00. Done when both examples pass and a stranger can use it without help.

Her father initials it and asks one question: “What happens if someone types 20%?” She adds a third example and writes “should say please type a number.”

Predict: “It’ll print 8.0, not 8.00.” Build and Test:

```
bill = 40.00
percent = 20
tip = bill * percent / 100
total = bill + tip
print(tip, total)
```

It prints 8.0 48.0. Her prediction was right, and the spec says dollars and cents, so the test fails. Debug and Log: Expected 8.00 and 48.00; Actual 8.0 and 48.0; guess: Python doesn’t know these are money. She looks up formatting, changes the last line to `print(f"{tip:.2f} {total:.2f}")`, and it prints 8.00 48.00. The Debug Log gets a line and 55 minutes.

Review: the change matches the spec. She saves with the message “Format money to two decimals.” Her father, as first user, types 20% and watches it crash, which she already knew. She writes it down as the next test.

Ship and Explain: “I specified a tip calculator with three examples. I predicted it would drop the zero. It did. Next I’d change it to reject a percent sign politely. Help I used: the Python documentation on formatting.”

The Debug Log

The Debug Log (BUG-05) is a notebook or a plain text file, kept all year. Each entry is dated and short:

- **Symptom:** what went wrong, in one line
- **Expected/Actual:** what your student expected, and what happened (BUG-01)
- **Guess:** why they think it happened
- **Test:** the one change they tried
- **Fix:** what worked, or “not yet”
- **Lesson:** one sentence, if there is one
- **Minutes:** how long the Hour ran

The minutes add up to hours, and the hours to credit. At 17–18, a five-minute conversation about one entry is often the best teaching of the week.

First user

The first user (TEST-05) is you. You sit down at the program and use it like a stranger would, someone who didn’t write it and doesn’t know what it’s supposed to do. Then you report what happened, plainly, without fixing anything.

Try this exact sentence: “I’m going to use it like a stranger. I won’t fix anything; I’ll just tell you what happened.” Then: “It asked for the bill. I typed forty dollars, in words. It crashed. Over to you.”

You don’t need to know any code to do this well; you’ll type the things real people type.

Who runs it

15–16. Your student is the session lead from the first week. They write the spec, keep the clock, keep the Debug Log, and say the exit line. You’re in the room. You initial Monday’s plan (PLAN-01), ask one question about each new spec, and you’re first user every Hour.

17–18. Your student plans the week and hands you or emails you the plan on Monday. They run the Hour on their own. You read the Debug Log the same day, ask about one entry, and act as first user once a week.

Place by skill, not birthday. A seventeen-year-old who is new to all of this may want the 15–16 shape for a semester. In the chapters, “ready 16” marks 17–18 moves some sixteen-year-olds can take.

The rescue order

When your student is stuck and asks for help, use the same order every time: **ask** → **wait** → **hint** → **short demo on your own copy** → **hand it back**.

Ask a real question: “What did you expect on that line?” Wait. Count to five in your head, and longer if they’re reading code. Offer a hint that points without solving: “Look at what the last line of the red text says.” If a demo is truly needed, show it on your own paper or your own copy of the file, then close it and hand the turn back. Your hands stay off your student’s keyboard. A program they fixed is theirs.

If you write code yourself, this is the hardest part of the book. Resist fixing.

When the Hour goes wrong

If the Hour looks like...	Try this
No spec, just typing	“Write three lines first: what goes in, what comes out, how you’ll know it’s done.” Then start.
Red text and panic	Read the last line of the error aloud first. What’s its name? Which line number? Write Expected/Actual.
“I’m deleting it and starting over”	“Before you start over, what does the Debug Log say you tried?” Save a copy, log first, then decide.
A chunk of AI-written code appeared	Stop. Fill in the Whose Work Is It? card (AI-06) for that part. Then walk through it line by line, or take it out.
The project is too big to finish	Scope it down (PLAN-03, Chapter 1): what version could ship in two weeks?
A flat, bored Hour	Switch to an unplugged card, or ask your student to teach you the hardest line. Boredom often means they’re ahead.
You took the keyboard	Hand it back and say so: “That was me fixing it. Your turn.”
It ran long and nobody wants to stop	Stop at the next passing test. Write the next step in the log for tomorrow.

The try-it shape

Every named activity in the book is a try-it, and every try-it box has the same shape. The heading carries its bank code and name, such as *In-home — SPEC-01 The One-Page Spec* or *Out-of-home — OUT-02 Library Computer Hour*. Then:

- **Time** — about how long it takes
- **Materials** — what to gather
- **Safety and privacy** — anything to watch
- **The fun** — why it’s worth doing

- **The skill** — what it builds
- **The crumb** — what to save for the record

Then **How it goes**, with band notes (15–16, 17–18, **ready 16**) where they matter. Each teaching chapter has at least one in-home try-it and one out-of-home try-it. Here's one worth knowing in week one.

Out-of-home — OUT-02 Library Computer Hour

- **Time:** about 50 minutes
- **Materials:** a library card, a USB stick or a family cloud folder, the spec on paper
- **Safety and privacy:** log out of everything before you leave; save nothing to the library computer
- **The fun:** a change of scene, and proof that the work doesn't depend on one machine
- **The skill:** running a whole Computing Hour on a public computer and leaving no trace
- **The crumb:** the file saved in two places

How it goes. Your student brings the spec on paper and runs the whole Hour on a library computer, using a browser-based Python editor or whatever the library offers. At the end they save the file to the USB stick or family folder, check that it opened, log out of every account, and close the browser. At 15–16 you sit beside them. At 17–18 they can go alone, and the crumb is the file arriving in its two places at home.

The talk-box shape

Every teaching chapter has one talk box, and its shape never changes.

The opening is a real question. It asks about your student's own spec, prediction, evidence, or judgment, and it has no answer to recite.

Five follow-ups, always in these five kinds:

1. **Prediction:** "What will it do?"
2. **Evidence:** "What did you see that makes you say that?"
3. **Location:** "Which line is the last one that did what you expected?"
4. **Change:** "What's the one thing you'd change?"
5. **Revision:** "You can change your answer. Run it again."

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at your student.

What a stuck silence usually means. Rarely that your student doesn't care. More often the question was vague or there was no wait; your student fears looking wrong in front of a parent (keep every comment about the code); they're waiting for you to fix it because you have before; there's no written spec or prediction to compare against; or the screen is louder than the question (lower the lid for a moment). Sometimes your student is ahead and bored; ask them to teach it.

Don't say / say instead. Each chapter gives one. For example: don't say "That's wrong"; say "Show me the line where it stops doing what you expected."

Family safety basics

These are ordinary and non-negotiable. Chapters point back here; Chapter 7 teaches security in depth and Chapter 8 teaches image safety.

- **The family security rule (SEC-01).** Your student practices security only on systems they own and toy apps they built. Never anyone else's system, even "to help," even with a friend's say-so. The law applies to teenagers. The rule is one page, signed by both of you, and it goes up in week one.
- **First names only.** Anything shared beyond the family carries first names only: no faces, no school, no town. Screenshots are cropped to the work.
- **Your student holds their own passwords,** on their own accounts, with long passphrases and two-step login (a password plus a second thing, such as a code or a key) turned on. You and your student make a recovery plan together, so a lost phone isn't a lost account.
- **Stop, Save, Tell (SAFE-02).** If anything online is threatening, pressuring, or sexual: stop replying, save it instead of deleting it, and tell a parent. Nobody is in trouble for telling. Chapter 8 rehearses it.
- **Library and shared computers.** Log out of everything; save to your own drive, not the machine.
- **Two places.** Every file lives in two places: the family drive or a USB stick, and one backup. Not a shared bot computer, and not a chatbot.
- **Publishing is a family decision.** Nothing goes public without your yes.
- **Screens out of the bedroom overnight** is a family call at this age, offered here, not imposed.

A note on screen time

The American Academy of Pediatrics asks families to make a media plan of their own rather than follow one number, putting the weight on content and on what screens crowd out: sleep, meals, movement, and homework.² The World Health Organization asks families to limit recreational screen time and to keep teenagers moving at least an hour a day, without giving a number for screens.³ On sleep, the American Academy of Sleep Medicine recommends 8 to 10 hours in every 24 for teens 13 to 18.⁴

These are guidelines, and they change as the research does. The Hour is purposeful school work; nearly half its activities are unplugged, and it ends with the screen closed. A late-night bug goes in the Debug Log and waits for morning.

The AI box

Working with AI in the Computing Hour

The rule underneath everything: your student specifies, predicts, writes, tests, and debugs. AI may explain, suggest, or check, within the service’s terms and your family’s rules. Anything it produces is made by a tool, not your student’s work.

At 15–16. Chat-style help is allowed, under the terms and with a parent nearby, *after* your student has written Expected/Actual and a guess: explain an error message, explain an idea, suggest practice problems with the answers hidden. **AI coding assistants inside the editor stay off.**

At 17. Your student may turn an editor assistant on for **one named project**, after passing a closed-book Build From a Blank (AS-04). The project’s README and Whose Work card say so. Every suggestion gets explained line by line before it stays (AI-05, Chapter 8).

At every age. AI is off for every check: Cold Trace, Debug on the Spot, Explain Your Code, Build From a Blank, Whose Work Audit, Spec-to-Test Check, and Portfolio Defense. “Help I used: ...” goes on every exit line. Generated code, text, and images are never filed as your student’s work.

Please don’t: paste private messages, photos, names, or your address into a chatbot; upload photos of family or friends to any AI tool; use companion or role-play apps; use an AI “detector” to judge your student, because “explain it to me” is the check; trust a citation until you’ve opened it.

Why the care? In one large study of high-school math students (not computing, and not homeschoolers), unrestricted chatbot help raised practice scores while lowering what students could do alone later; a version that gave hints instead of answers largely avoided the harm.⁵ This is a useful study, not a promise that every home will see the same result.

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.⁶

Age terms: 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 18.⁷ Check the current terms.

At 18: a family AI agreement, written together as adults in one household (SAFE-05, Chapter 8), replaces the age rules above.

Whose work is it? Your student’s work is what they specified, wrote, tested, and can explain. Every project carries a Whose Work Is It? card (AI-06), “Whose Work card” for short: who or what helped, what was asked, what came back, what was kept, what was changed, and how it was tested. A card that says “none” is a perfectly good card.

Dose honesty

There's no national rule for how many hours a homeschooler should code. A common homeschool guideline counts about 150 hours of work for one full high-school credit in a core subject, and about 120 hours for an elective; it's a guideline, not a law.⁸ In practice, about four Computing Hours a week across a 36-week year, plus try-its and projects, makes a full credit. Two Hours a week makes a half credit, and that's a real course too. Chapter 9 shows how to count.

The life of the habit

Keep the Debug Log going all year. Do a Cold Trace (AS-01) once a month: a short program traced on paper, alone, dated. Finish a project every four to six weeks. Hold a demo day once a semester for family, a co-op, or a grandparent on a video call.

None of this is a schoolroom invention. Specifying, testing, logging, reviewing, and explaining your work to a colleague is how working programmers already work. The Computing Hour is the same habit, in a smaller room.

Things to say

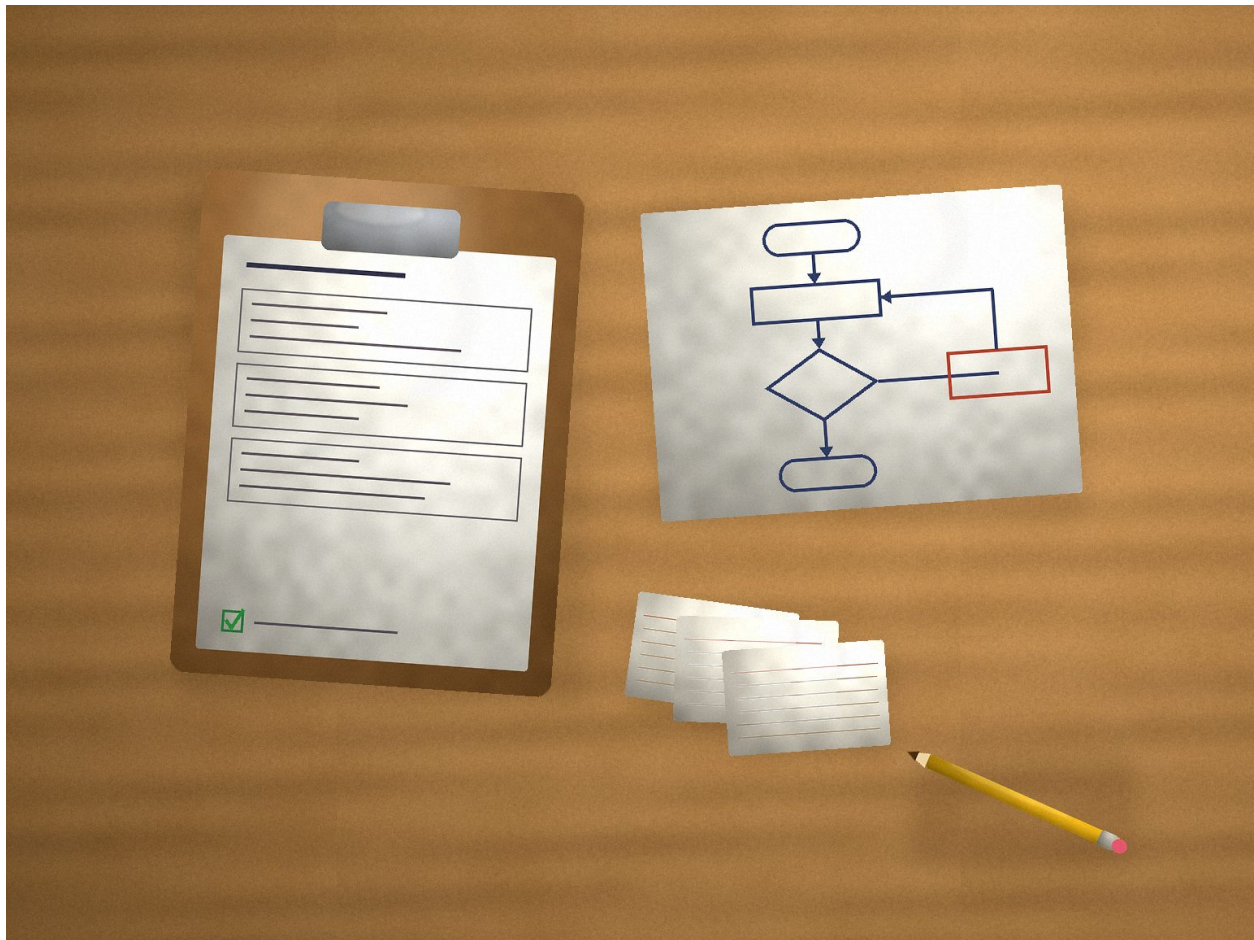
1. **Opening Specify (15–16).** “What's the goal today, and how will you know it's done?” Wait. Your student reads you the done line.
2. **Predict.** “Write down what you think it will print. Then run it.”
3. **Red text.** “Read me the last line. What's the error's name? Which line?” Wait. “What's your best guess?”
4. **The urge to rewrite.** “Before you start over, what does the Debug Log say you tried?”
5. **Review.** “Show me what changed since last time. Does the spec still say the same thing?”
6. **Ship and Explain.** Your student: “I specified... I predicted... It did... Next I'd change... Help I used: ...”

Ready to begin

Chapter 1 starts where every Hour starts: with a spec. Find a clean sheet of paper.

Chapter 1

What Computing Looks Like at 15–18: Specs, Algorithms, and Running Your Own Work



Why this matters

Here is a picture. It is an illustration, not a real family.

A seventeen-year-old has been asked to build a chore rotation tool for his younger sisters. He opens

a laptop, and his mother, who has read this chapter, closes it again. Gently. “Spec first,” she says, and slides a sheet of paper across the table.

He writes for twelve minutes. At the top: “A tool that assigns four chores to three sisters across seven days, fairly.” Then the inputs (names, chores, days), the output (a printed week), the rules (nobody gets dishes two days running), three examples, and a last line: “Done when every sister gets every chore at least once in a week, and the youngest can read the printout herself.”

His mother reads it the way a stranger would. “What does ‘fairly’ mean?” He crosses it out and writes what he actually means. “What happens when someone’s away for the weekend?” He hadn’t thought of that. He writes it down as a question for later.

By the end of the week the tool works. The spec has changed twice, and both changes are written on the page with a date. When his sisters argue that the rotation isn’t fair, he doesn’t argue back. He shows them the done line and asks which example they’d like him to add.

That is computing at fifteen to eighteen. The programs are bigger, the languages are real, and the tools are the ones professionals use. But the deepest change isn’t technical. At this age, the work becomes your student’s own. They decide what to build. They write down what it must do. They run the session, keep the history, and defend the result. You stop being the person who plans the lesson and become the person who asks, “How will you know it’s done?”

This chapter gives that change its first tools. The spec comes first: a short written description of what a program must do, with examples and a line saying what “done” means. Then algorithms: the exact steps inside the spec, which can be counted, compared, and traced on paper before a single line is typed. Almost all of it needs no screen. And all of it rests on the line the book runs on:

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

This chapter is about the first clause. The rest of the book is about the other three.

For the parent: understand it yourself

The plain picture

A spec is a promise written before the work starts. It says what the program is for, who will use it, what goes in, what comes out, what rules it follows, and how you’ll know it’s finished. It includes two or three examples: “if someone types this, the program should do that.” At least one example is an edge case (an input at the boundary, or an odd one a careless user might type).

An algorithm is the exact list of steps that gets from the input to the output. Recipes, driving directions, and the rule your family uses to decide who sits in the front seat are all algorithms, some better written than others. A computer needs an algorithm with no gaps, because it will do exactly what it’s told, in order, every time, and never guess what you meant.

People sometimes call this whole way of working computational thinking: breaking a problem down, spotting patterns, setting aside detail that doesn’t matter, and writing exact steps. This book uses plain words for it from here on.

The precise picture

A good spec describes **behavior, not code**. “The program shows the cheapest of three prices” is behavior. “Use a for loop” is code, and it belongs in the program, not the spec. A spec that names its examples precisely (“input: 4, 9, 2 → output: 2”) is already halfway to a set of tests, which is exactly what Chapter 3 will turn it into.

Algorithms can be compared. Some take more steps than others to do the same job, and the difference grows as the job grows. Looking for one card in a stack of sixteen by flipping each card in turn takes up to sixteen flips. If the stack is sorted, you can start in the middle and throw away half the stack with each flip: no more than five flips for sixteen cards, and only one more when the stack doubles to thirty-two. That’s the difference between linear search and binary search, and your student will count it by hand this week.

Some algorithms always give the same output for the same input (deterministic). Others involve chance and might give a different answer each time (probabilistic). A sorting rule is the first kind. A shuffled playlist is the second. That distinction matters a great deal in Chapter 8, because AI systems that write text are probabilistic.

The national standards many schools use for high-school computer science cover ages 14 to 18 and put it this way for algorithms: students should “Evaluate algorithms for efficiency, correctness, and clarity, using metrics or test cases.”⁹ That is the whole chapter in one sentence. Efficiency is the step count. Correctness is the spec’s examples. Clarity is whether someone else can follow it.

An honest word about the evidence

You’ll hear that learning to code improves thinking in general. The best summary of the research is more modest. A large review of studies found that learning to program does carry over to some other kinds of thinking, moderately, and most to skills close to programming itself; the effect was smaller when programming was compared with other active lessons, and smallest for school achievement and reading.¹⁰ This is a useful study, not a promise that every home will see the same result. Learning to program teaches programming, and that is reason enough. Specs, in particular, are a craft. They make your student a better builder of software, and the book promises nothing beyond that.

Wrong answers you should be able to hear

You don’t need to know any code to hear these. Each one has a question that turns it around.

- **“It should just work.”** There’s no done line. Ask: “How will you know it’s done? Write it down.”
- **“Use a loop to go through the list.”** The spec describes code, not behavior. Ask: “What will the person using it see?”
- **Examples that are all ordinary.** No edge case. Ask: “What would a stranger type? What about nothing at all?”
- **“Binary search is always better.”** Only if the data is sorted first. Ask: “What does it need before it can start?”

- **“I’ll figure out the steps as I type.”** The algorithm lives only in their head. Ask: “Walk me through it on paper, one step at a time.”

A five-minute warm-up for you

Before you teach this, write a spec for something you do every day: making coffee, starting the car, or the rule for whose turn it is to pick the movie. Include a done line and one edge case (“What if we’re out of milk?”). Then hand it to your student and ask them to follow it literally. You’ll laugh, and you’ll understand the chapter.

How to teach it this week

The first week of Chapter 1 fits inside the plan in “How to Use This Book This Week.” Here’s the session shape for a spec day, using the Hour’s step names.

Specify (most of the Hour on day one). Your student writes a one-page spec for something small. Hand them the sheet and say, “Before any code: what’s it for, who uses it, what goes in, what comes out, and how will we know it’s done?”

Predict. On a spec day, the prediction is about the spec itself: “Which part of this do you think I’ll get stuck on when I follow it?”

Build and Test. On paper. You follow the spec literally (SPEC-03, below), and your student watches where you stall.

Debug and Log. Every stall becomes a fix written in a second color, and one line in the Debug Log: “Expected: Mom would get two slices. Actual: she got the whole loaf. Fix: ‘take out exactly two slices.’”

Review. Your student reads the spec start to finish one more time. Is there still a done line? Is there an edge case?

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

In-home — SPEC-01 The One-Page Spec

- **Time:** 20 minutes
- **Materials:** one sheet of paper, a pencil, a second-color pen for later
- **Safety and privacy:** if the program involves family members, first names only
- **The fun:** finding out how much you know about a program before it exists
- **The skill:** purpose, user, inputs, outputs, rules, and what “done” means, before any code
- **The crumb:** the signed spec page

How it goes. Your student picks a small program they’d actually use: a tip calculator, a reading-log totaler, a timer for practice sessions, a chore rotation. On one page they write six things, each with a short heading.

1. **Purpose:** one sentence. What is it for?

2. **User:** who will use it? (“My grandmother, on her tablet” is a better answer than “people.”)
3. **Inputs:** what goes in, and in what form?
4. **Outputs:** what comes out, and in what form?
5. **Rules:** what must always be true? What must never happen?
6. **Done when:** one line that a stranger could check.

They sign and date the bottom. You read it and ask one question. Not a correction: a question. “What happens if the bill is zero?” is a good one.

15–16: you ask one question, and your student revises. You initial the page. **17–18:** your student writes the spec alone and hands it to you on Monday with the week’s plan. You read it the way a stranger would and write your one question in the margin.

Fading your help

In week one, you ask the done-line question out loud. By week three, your student writes the done line before you get there. By the end of the semester, they’re asking you what you’d type as an edge case, and you’re the one who has to think. That’s the direction this book always moves: from your question, to their habit, to their judgment.

Band moves

15–16: one page; one question from you; the spec is written before the first Hour of each project and updated in each Specify step after that.

17–18: user stories with acceptance lines (SPEC-04) for anything someone else will use, and a spec diff (SPEC-05) at the end of each project, comparing what the spec said with what shipped.

Practice that actually builds learning

Practice in this chapter goes in two directions: making specs sharper, and making algorithms countable. Start each with one kind of card at a time (blocked practice) for a week or two, then mix them so your student has to decide which tool fits.

In-home — SPEC-02 Examples Before Code

- **Time:** 15 minutes
- **Materials:** the spec from SPEC-01, a ruled sheet for a table
- **Safety and privacy:** no real names or numbers from anyone outside the family
- **The fun:** inventing the inputs a careless or creative user might type
- **The skill:** writing input → expected output examples, including edge cases, as part of the spec
- **The crumb:** the example table

How it goes. Under the spec, your student draws a table with two columns: *If the input is...* and *The program should....* They fill in at least three rows. One is ordinary. One is at a boundary (zero,

the largest allowed value, an empty list). One is something a stranger might type (“twenty,” “20%,” a space at the end, nothing at all). Every row says exactly what should happen, including “politely say please type a number.”

15–16: three rows, one of them an edge case. **17–18:** five rows, and your student marks which rows would make good automatic tests later (Chapter 3).

In-home — SPEC-03 The Literal Reader

- **Time:** 15 minutes
- **Materials:** a written spec or set of instructions, a second-color pen; for the first round, a sandwich kit or a deck of cards
- **Safety and privacy:** no knives or heat in the literal-reader round; keep it to cold food or cards
- **The fun:** watching a parent follow instructions with total, deadpan literalness
- **The skill:** repairing every ambiguity in a written spec
- **The crumb:** the marked-up spec in two colors

How it goes. For the first round, your student writes instructions for something physical: “make a peanut butter sandwich” or “sort these ten cards from lowest to highest.” You follow them exactly as written. If the instructions say “put the peanut butter on the bread,” you put the jar on the loaf. Every place you stall or go wrong, your student writes a fix in the second color. Then swap to a real spec, the one from SPEC-01, and do it again on paper: you play the program and do only what the spec says.

15–16: the sandwich round first, then their own spec. **17–18:** skip the sandwich if they’ve done it before, and swap to a spec for a program another family member will actually use. Ask that family member to be the literal reader.

In-home — SPEC-04 User Story and Acceptance Test

- **Time:** 15 minutes
- **Materials:** index cards
- **Safety and privacy:** first names only on cards
- **The fun:** writing like a professional team does
- **The skill:** “As a ..., I want ..., so that ...” with a checkable acceptance line
- **The crumb:** the story card

How it goes. On each card, your student writes one user story in a fixed form: “As a [user], I want [something], so that [reason].” Underneath goes an acceptance line that anyone could check: “Given four names and seven days, every name appears at least once.” A project usually has three to six stories. Your student sorts them into “must have” and “nice to have” and builds the must-haves first.

17–18: for any project someone else will use. **Ready 16:** a sixteen-year-old who writes clean one-page specs can start here.

In-home — SPEC-05 Spec Diff

- **Time:** 15 minutes
- **Materials:** the original spec, the finished program, the Debug Log
- **Safety and privacy:** nothing new to watch
- **The fun:** finding out what you actually built
- **The skill:** comparing what the spec said with what shipped, and writing the change note
- **The crumb:** the spec-versus-shipped note

How it goes. At the end of a project, your student lays the first spec beside the finished program and goes line by line. What did the spec promise that shipped? What shipped that the spec never mentioned? What did the spec promise that didn't ship, and why? They write a short change note. Something that shipped without being in the spec gets a decision: add it to the spec with a reason, or take it out.

17–18: at the end of every project. This note goes in the portfolio (Chapter 9).

Here's how one Hour might go (17–18)

This is a composite, not a real family. Remember the chore rotation tool from the start of the chapter? Its user stories included “As the youngest sister, I want my chores printed in big letters, so that I can read them myself.” The acceptance line: “Given four names and seven days, every name appears at least once.”

At the end of the week, the spec diff turned something up. The tool now skipped anyone marked “away,” which no spec line had ever mentioned. He'd added it on Thursday, when a sister went to a sleepover. His first instinct was to delete it, because it wasn't in the spec. Instead he added it to the spec, with a date and a reason (“added after a real absence; needed for fairness”), and wrote a new example. His exit line: “I specified a rotation for three sisters. I predicted the spec diff would be clean. It wasn't: I'd added absences. Next I'd change the printout to show who was away. Help I used: none.”

In-home — ALG-01 Two Ways to Search

- **Time:** 20 minutes
- **Materials:** 32 index cards numbered 1 to 32 (or playing cards), a tally sheet
- **Safety and privacy:** none needed
- **The fun:** a race between two methods, with a clear winner and a catch
- **The skill:** linear versus binary search with cards; counting steps as the list doubles
- **The crumb:** the step-count table

How it goes. Use sixteen cards first. Shuffle them and lay them face down in a row. You choose a target number. Your student finds it by flipping cards one at a time from the left, and you tally the flips. That's linear search. Do it three times and write down each count.

Now sort the cards and lay them face down in order. Your student flips the middle card. If it's too high, the target must be to the left, so the whole right half goes away. Flip the middle of what's left. Keep halving. That's binary search. Tally again.

Before using all 32 cards, your student predicts both new counts in writing. Then they run both searches and compare. With sixteen cards, linear search can take up to sixteen flips, and binary search never more than five. With thirty-two, linear can take up to thirty-two, and binary never more than six. Doubling the cards doubles one count and adds one to the other. The catch: binary search only works on sorted cards. Ask, “What did we have to do first, and did that cost anything?”

15–16: sixteen cards, then thirty-two with a prediction. **17–18:** add a column to the table for “64 cards (predicted)” and have your student explain the pattern in one sentence.

Here’s the same idea in Python, for later in the year. Your student doesn’t need to understand every line yet; the point is that the counts match the cards.

```
def linear_steps(cards, target):
    steps = 0
    for card in cards:
        steps += 1
        if card == target:
            return steps
    return steps

cards = list(range(1, 17))
print(linear_steps(cards, 16))
```

This prints 16, the worst case for sixteen cards. With `range(1, 33)` and a target of 32, it prints 32.

In-home — ALG-02 Sort by Hand, Count the Work

- **Time:** 25 minutes
- **Materials:** ten playing cards, a tally sheet
- **Safety and privacy:** none needed
- **The fun:** discovering that “just sort them” hides a lot of work
- **The skill:** selection sort and merge sort with playing cards; comparisons tallied
- **The crumb:** the comparison tally

How it goes. Deal ten cards face up in a row. First, selection sort: your student scans the whole row for the smallest card, moves it to a new row, and repeats. You make a tally mark every time they compare two cards. Then shuffle and try merge sort: split the row into pairs and sort each pair, merge pairs into fours, merge fours into eights, and so on, comparing only the front cards of the two piles being merged. Tally again. Compare the two tallies. With ten cards the difference is noticeable; ask your student to predict what happens with a hundred.

15–16: selection sort first, merge sort on a second day. **17–18:** both in one session, and a written sentence on which they’d choose for a thousand cards and why.

In-home — ALG-03 Pseudocode to Trace Table

- **Time:** 15 minutes
- **Materials:** a short program on paper, a ruled sheet

- **Safety and privacy:** none needed
- **The fun:** being the computer
- **The skill:** tracing a loop with an accumulator line by line before running it
- **The crumb:** the trace table

How it goes. Give your student this program on paper:

```
total = 0
for n in [3, 5, 2]:
    total = total + n
    print(n, total)
print("done", total)
```

They draw a table with a column for each name that changes (`n` and `total`) and a column for what gets printed. Then they go one line at a time, writing each new value in a new row. When they're finished, they write their predicted output at the bottom and only then run it. It prints:

```
3 3
5 8
2 10
done 10
```

If the trace and the run disagree, the disagreement is the lesson. Find the row where they part ways.

15–16: loops with one accumulator, like this one. **17–18:** loops with an `if` inside, or two accumulators. Chapter 2 picks this card up again with real files.

In-home — ALG-04 Coin Flip or Rule?

- **Time:** 15 minutes
- **Materials:** a die, eight index cards describing procedures
- **Safety and privacy:** none needed
- **The fun:** proving something changes by doing it three times
- **The skill:** deterministic versus probabilistic algorithms with dice and a fixed rule
- **The crumb:** the two-column sort of examples

How it goes. On each card, write a procedure: a recipe, a shuffled playlist, a dice game, alphabetizing a shelf, a coin toss to pick who goes first, a bus timetable, a sorting rule, a “random fact of the day.” Your student sorts them into two columns: “same input, same output every time” and “might change.” Then they pick one card from the second column and run it three times to prove it. Ask: “Is a weather forecast in the first column or the second? What about a chatbot’s answer?” Chapter 8 comes back to that last one.

17–18: the full card set. **Ready 16:** fine for any sixteen-year-old who has done ALG-01 and ALG-03.

In-home — PLAN-03 Scope It Down

- **Time:** 15 minutes

- **Materials:** a project idea that's too big, two columns on paper
- **Safety and privacy:** none needed
- **The fun:** keeping the dream and shipping something real
- **The skill:** cutting a too-big project idea to a version that can ship in two weeks
- **The crumb:** the before-and-after scope list

How it goes. Every student eventually proposes a project the size of a small company: a game with online multiplayer, a social network for the co-op, an app that does everything. Don't say no. Say, "What's the version you could finish in two weeks?" Your student writes every feature on the left, then moves only the ones that a first user truly needs to the right. The right column becomes the spec. The left column stays on file as "later," and some of it will come back.

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

- **Time:** 20 minutes, plus travel
- **Materials:** a notebook, a pencil
- **Safety and privacy:** observe, don't film; no photos of strangers or their screens
- **The fun:** spying on a machine's rules in plain sight
- **The skill:** writing a spec for a self-checkout, an elevator, or a traffic signal from observation
- **The crumb:** the spec page

How it goes. Pick a machine with rules: a self-checkout, an elevator bank, a crosswalk signal, a parking-garage gate. Your student watches for ten minutes and writes down everything it does, especially what happens when something unusual occurs. Back home, they turn their notes into a one-page spec in the SPEC-01 format, as if they were the engineer who had to build it. Their edge-case example must be something they actually saw: the elevator that skipped a floor, the self-checkout that called for an attendant.

15–16: you go together, and you compare notes on the way home. **17–18:** your student goes alone, and adds two user stories for different users of the same machine (a person with a full cart, a person using a wheelchair).

Talk box (this chapter)

Opening: "What exactly should it do — and how will you know it's done?"

Follow-ups: 1. **Prediction:** "If I followed this spec exactly, where would I get stuck first?" 2. **Evidence:** "Which example in your table shows that rule?" 3. **Location:** "Which line of the spec says what happens when the input is empty?" 4. **Change:** "If you could add one more example, which would it be?" 5. **Revision:** "You can change the done line. Does it still say what you mean?"

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. In this chapter, usually that there's no done line yet, so there's nothing to check the question against. Or the question was about code when the spec is about behavior. Try again with "What will the person using it see?"

Don't say / say instead. Don't say "That spec is too vague"; say "Read me the line that tells me when I'm finished."

One wrong example to diagnose

Give your student this spec and ask what's wrong with it. Don't hint.

Purpose: A program that helps with homework. **User:** Students. **Inputs:** The homework. **Outputs:** Help. **Rules:** It should be good. **Examples:** If I put in my math homework, it helps me.

There's no done line, so nobody could ever say it's finished. The user is everyone, which means it's built for no one. Every heading is filled in, and nothing is specified. A good repair starts by picking one student, one subject, and one kind of help, and ends with a line a stranger could check.

Working with a partner

Siblings, co-op classes, and two students sharing one computer can work as a pair. Pair programming is a simple arrangement: one person types (the driver) while the other reads, questions, and spots problems (the navigator), and they swap roles every ten or fifteen minutes. A review of studies of pair programming in programming courses found positive results for pairs on programming assignments, exams, and passing rates, though not on students' attitudes.¹¹ Most of those studies were college classes, not homes, so treat it as promising, not proven for your family.

For pairing at home, a few rules help. Each pair writes one spec together. Swap roles on a timer. The navigator never grabs the keyboard; they point with words, just as you do. Both say the exit line, and both write their own "Help I used: ..." At 17–18, two students can also spec together and build separately, then compare results; Chapter 9 counts both.

For the student

This section is for you. Your parent may read it too, but it's written to you.

At this age, computing changes. Up to now, someone else probably decided what you'd build. A lesson, a tutorial, or a course told you what to make, and your job was to make it. From here on, the work is yours. You decide what to build. You write down what it must do. You run the session. Your parent's job becomes asking hard questions, using your program like a stranger, and signing your records. Your job is to make something real and be able to explain every part of it.

The first tool for that is the spec.

What a spec is

A spec is a short written description of what a program must do, with examples and a line saying what "done" means. You write it before you write any code. It's not homework about your home-

work. It's the cheapest place to find your mistakes. A wrong idea costs you a minute to cross out on paper and an afternoon to unpick in code.

A good spec has six parts: purpose, user, inputs, outputs, rules, and a done line. It also has examples, at least one of them odd. Here is a tiny one:

Purpose: Tell me how many minutes I've practiced guitar this week. **User:** Me, every evening. **Inputs:** A minutes number each day, typed in. **Outputs:** The week's total and the daily average, as whole numbers. **Rules:** Negative numbers aren't allowed. A day I skip counts as zero. **Examples:** 30, 20, 0, 45, 30, 0, 25 → total 150, average 21. Seven zeros → total 0, average 0. Someone types "-10" → it says "minutes can't be negative" and asks again. **Done when:** all three examples work and my brother can use it without asking me anything.

Notice what it doesn't say. It doesn't say "use a loop" or "store it in a list." Those are decisions about code, and you'll make them later. A spec is about what the person using it sees.

Why it's worth it

Professionals write specs because guessing is expensive. When you sit down to code with a spec, you know exactly what you're aiming at, and you know when to stop. When something goes wrong, you can check the program against the examples instead of arguing with yourself about what it should do. And when your parent, or a college instructor, or a future employer asks what you built, you can show them.

Algorithms are the steps inside

Once you know what the program must do, you need the exact steps to do it. That's an algorithm. Some algorithms are faster than others, and you can measure the difference by counting steps. This week you'll search a stack of cards two ways and find out that one method takes up to thirty-two flips for thirty-two cards while the other never takes more than six. That's a real difference, and it gets bigger the more data you have.

One to write today

Write a one-page spec for something you'd actually use this month. A homework timer. A tool that splits a pizza order among friends. A tracker for the books you're reading. Use the six parts. Include three examples, and make one of them strange. Sign it and date it. Then hand it to your parent and ask them to find the weakest line. Fix it in a different color.

Explain it back

When you're done, try this without looking: tell your parent, in three sentences, what your program does, who it's for, and how you'll know it's finished. If you can't, the spec isn't done yet.

This part is hard for everyone

Edge cases. Nearly everyone writes examples that are all ordinary, because you imagine yourself using the program, and you know what to type. Real users don't. They type words where numbers go, press Enter on an empty box, or paste in a whole paragraph. Every time you write a spec, ask yourself: "What would a stranger type?" Then write that down as an example.

A challenge

Go to a store with a self-checkout and watch it for ten minutes. (Don't film anyone.) Pay attention to what it does when someone puts something in the bagging area that it didn't expect. Write the spec for that one rule, the "unexpected item" rule, as if you were the engineer who had to build it. What are the inputs? What does it do? What should it do when a customer puts down their own bag? How would you know your version was done? You'll find that a rule that annoys millions of people every day is a genuinely hard problem.

If it isn't clicking

What you see	What it usually means	Next move
Specs that describe code ("use a for loop")	Behavior and method are mixed up	Rewrite each line as "what a user sees"; run SPEC-03 again
No edge-case example, ever	Hasn't imagined a careless or hostile user yet	Ask, "What would a stranger type?"; try the boundary hunt in Chapter 3 early
Step counts guessed, not counted	Algorithms still feel abstract	Cards on the table; ALG-01 again with all 32 cards
The spec is three pages long	Spec as homework, not as a tool	PLAN-03: what's the two-week version? One page only

When your student is stuck in the moment, use the rescue order: ask → wait → hint → short demo on your own copy → hand it back. The demo, in this chapter, is usually you writing a spec of your own on a separate sheet, then putting it away.

If specs keep feeling pointless, that's a sign the projects are too small for a spec to matter. Pick something that someone else will use. A grandparent, a younger sibling, or a co-op teacher makes a spec feel necessary very quickly.

If you'd like outside help, a working programmer will talk about specs for twenty minutes without much prompting. Chapter 6 has an activity for that.

Tools, including AI

This chapter needs paper, pencils, index cards, and a deck of playing cards. A screen is optional until the Python version of ALG-01, which any Python editor can run. Chapter 2 sets that up.

The AI box in **The Computing Hour** covers what AI may and may not do. One point applies here especially. A chatbot can generate a spec in seconds, and it will look good. It won't be your student's spec. It won't know who the user is or what "done" means in your house, and writing the spec is where your student learns what they're building. Specs in this chapter are written by hand, without AI. Check the current terms before using any service.

What “done enough” looks like

Your student is ready to move on when they can:

- ☐ write a one-page spec with all six parts, a done line, and at least one edge-case example (SPEC-01, SPEC-02);
- ☐ repair a spec after a literal reader stalls on it (SPEC-03);
- ☐ count the steps for linear and binary search with cards, and predict what happens when the stack doubles (ALG-01);
- ☐ trace a short loop on paper and get the output right before running it (ALG-03);
- ☐ sort procedures into “same every time” and “might change” (ALG-04, 17–18 or ready 16);
- ☐ write a spec from something observed out in the world (OUT-01);
- ☐ **AS-01 Cold Trace, baseline:** trace a short program alone, without AI or notes, and predict its output; dated and filed.

At 17–18, add: user stories with acceptance lines for one project (SPEC-04), and a spec diff at the end of it (SPEC-05).

The Cold Trace (AS-01) is a baseline, not a grade. Repeat it once a month. A student who comes into this book already writing clean specs and tracing loops easily can move fast through this chapter and start Chapter 2 in week two. A student who finds all of it new should take two or three weeks here. Place by skill, not birthday.

Chapter 2

Python as a Tool: Programs That Read, Compute, and Write



Why this matters

Here is a picture. It is an illustration, not a real family.

A fifteen-year-old keeps a reading log in a plain text file, one line per book: the title, a comma, and the number of pages. Her spec says the program should read the file and print the total pages for

the year. Her prediction, written in the Debug Log before the first run: “817.”

The first run crashes. The red text ends with a line that names a `ValueError`. She reads that last line first, the way **The Computing Hour** taught her, and then the line number above it. The problem is the last line of her file, which is empty. An empty line has no comma, so it can’t be split into a title and a page count. She adds two lines of code to skip empty lines, runs it again, and gets 817.

Then she notices something else. One of her lines says 312 with a space after the number, and that one worked fine. She’d expected it to crash too. She checks, and it turns out Python’s `int()` quietly ignores spaces around a number. She writes it in the log as a good surprise: “`int` forgives spaces. It does not forgive empty.”

That is Python at this age. In *Computing for Young Minds*, Python was a first language: printing, asking, looping, and drawing. Here it becomes a tool. Real programs read something, compute something useful, and write a result a person can use. They work with real files, real data, and the libraries that come with the language. And they break in ways that teach you something, if you read the red text.

Python is a good working tool and the most common first text language, which is why this book uses it for most of the year. It isn’t “the best language”; there’s no such thing, only good fits. Your student will meet Java in Chapter 4, SQL in Chapter 5, and the languages of the web in Chapter 6. Python is the one they’ll reach for when they need to get something done.

For the parent: understand it yourself

The plain picture

A Python program is a text file of instructions that the computer follows from top to bottom. Most useful programs do three things: they **read** (from a file, or from what someone types), they **compute** (add up, count, sort, compare), and they **write** (print to the screen, or save a new file). This chapter is built around that shape.

Programs are made of a few kinds of things:

- **Values** of different **types** (what kind of value something is): whole numbers like 312, text like “Salt Road”, lists like [195, 312, 310].
- **Names** that hold values: `total = 0`.
- **Functions**, named chunks of code you can use again. A function can hand back a **return value**, which is what it gives to the code that called it.
- **Libraries**, collections of ready-made functions. Python comes with a large standard library, and part of being a programmer is knowing how to find what’s in it.

The precise picture

Here are the ideas that trip up nearly every student at this stage. You don’t need to fix them; you need to recognize them.

Text that looks like a number isn't a number. Anything read from a file or typed by a user arrives as text. "7" + "3" is "73", because adding text joins it. `int("7") + int("3")` is 10. Python's `int()` turns text into a whole number, and it's forgiving about spaces around the digits: `int("7 ")` gives 7. It isn't forgiving about anything else. `int("7.5")` and `int("")` both stop the program with a `ValueError`.

print and return are different jobs. `print` shows something on the screen for a person. `return` hands a value back to the code that called the function. A function that prints its answer instead of returning it looks like it works, until another part of the program tries to use the answer and gets `None`, which is Python's word for "nothing."

range stops one short. `range(1, 5)` gives 1, 2, 3, and 4, not 5. This is deliberate, and it causes more off-by-one errors than any other line in Python.

Where the program runs matters. When a program opens `"reading_log.txt"`, Python looks for that file in the folder the program is running from, which isn't always the folder you think. "The file doesn't work" usually means "Python is looking in the wrong place."

Lists and dictionaries do different jobs. A list keeps things in order: `["Salt Road", "Winter Fox"]`. A dictionary looks things up by a key: `{"tacos": 3, "pizza": 2}`. Counting how many times each thing appears is a dictionary job. Keeping a ranking is a list job.

The high-school standards many schools use put the library skill plainly: students should "Use documentation, libraries, application programming interfaces (APIs), and other tools in program development."¹² Reading the documentation is not a fallback for when you're stuck. It's part of the work, and it's a skill you can teach without knowing the answer yourself.

Wrong answers you should be able to hear

- **"It printed the right number, so the function works."** Possibly `print` where `return` was needed. Ask: "If another function used this answer, what would it get?"
- **"It says 73 instead of 10."** Text treated as a number. Ask: "What type is that value when it comes in?"
- **"It skipped the last one."** Probably `range`. Ask: "Read me the range line. What's the last number it gives?"
- **"The file is empty" or "the file doesn't work."** Usually a path problem. Ask: "Which folder is the program running from? Print it."
- **"It removed some of them but not all."** Changing a list while looping over it. Ask: "What happens to the next item when you remove this one?"

A five-minute warm-up for you

On a sheet of paper, predict what each line prints. Then ask your student to run them and check you.

```
print("7" + "3")
print(int("7") + int("3"))
print(int("7 "))
```

The answers are 73, 10, and 7. If the third one surprised you, you're in good company, and you've just learned the most useful habit in this chapter: predict, then check.

How to teach it this week

Setting up

If your student is new to typed code, start with Thonny, a free Python editor built for learners. It runs offline, needs no account, and comes with Python included, so there's one thing to install.¹³ It also lets you run a program one step at a time and watch each name change, which makes tracing visible. If your student has used Python before, any editor that runs Python works; Chapter 6 introduces a general-purpose editor and the terminal. CodeAI (formerly Code.org) also offers a browser-based Python Lab designed for students aged 13 and up, which suits a library Hour.¹⁴

Make one folder for this chapter, inside the course folder, and keep every program and every data file for the chapter in it. Many path problems disappear when everything lives together.

The session shape

Here's a Computing Hour for a "read, compute, write" day.

Specify. Your student writes or updates the spec: what file goes in, what comes out, three examples (one of them an empty or odd file), and the done line.

Predict. Before each run, a written line: "I predict it will print 817." Use PY-02 (below) at the start of the Hour if your student needs a warm-up.

Build and Test. Small steps. First, a program that just opens the file and prints each line. Run it. Then split each line. Run it. Then add up. Run it. Each run gets checked against the spec's examples.

Debug and Log. When red text appears, try this exact sentence: "Read me the last line. What's the error's name? Which line?" Then wait. Your student writes Expected/Actual and a guess before changing anything.

Review. Your student rereads the program against the spec and saves it with a message: "Skip blank lines in the reading log."

Ship and Explain. "I specified... I predicted... It did... Next I'd change..." and "Help I used: ..."

In-home — PY-02 Predict the Output

- **Time:** 10 minutes
- **Materials:** three to five short programs on paper or on screen, a prediction sheet
- **Safety and privacy:** none needed
- **The fun:** betting against Python and sometimes winning
- **The skill:** a written prediction of a short program's output, then run and compare

- **The crumb:** the prediction-and-result lines

How it goes. Show your student a short program. They write, on paper, exactly what it will print, line for line. Then they run it and write what it actually printed beside the prediction. For each mismatch, they write one sentence: “I thought ____ but Python ____.” Use the three lines from your warm-up, then the print/return program below, then a range loop. Make a habit of it: five minutes at the start of any Hour.

15–16: one program at a time; you choose them. **17–18:** your student writes predict-the-output puzzles for you, and has to know the answer first.

In-home — PY-01 A Program That Reads a File

- **Time:** 30 minutes
- **Materials:** a computer with Python, a small text file your student makes
- **Safety and privacy:** use made-up or family-only data; no one else’s names
- **The fun:** a program that does something with your own information
- **The skill:** opening a text file, reading lines, stripping whitespace, and counting
- **The crumb:** the program and its sample input file

How it goes. Your student makes a text file called `reading_log.txt` with a few lines in the form title, pages, and leaves an empty line at the end on purpose. Here is a first version of the program, saved in the same folder:

```
total = 0
with open("reading_log.txt") as f:
    for line in f:
        title, pages = line.split(",")
        total = total + int(pages)
print("Pages read:", total)
```

With a file of three books (195, 312 with a trailing space, and 310 pages) and a blank last line, this crashes. The last line of the red text reads `ValueError: not enough values to unpack (expected 2, got 1)`, and the line above it points to the `split` line. The blank line split into one piece, not two. Here’s the fix:

```
total = 0
with open("reading_log.txt") as f:
    for line in f:
        line = line.strip()
        if line == "":
            continue
        title, pages = line.split(",")
        total = total + int(pages)
print("Pages read:", total)
```

This prints `Pages read: 817`. `strip()` removes spaces and the invisible end-of-line character from both ends; `continue` skips to the next line. Notice that the line with 312 never caused trouble: `int()` ignored the space.

15–16: start with the crashing version and let your student find the bug. That’s the lesson. **17–18:** start from a blank file and the spec, and add a second output: the longest book.

Fading your help

In the first week, you read the error’s last line aloud together. By the third week, your student reads it before you can ask. By the end of the chapter, they’re telling you what the error means before they’ve finished reading it, and they’re right more often than not. When that happens, say so in one specific sentence: “You went straight to the last line and found the blank line. That’s exactly how it’s done.”

Band moves

15–16: one file in, one result out; Thonny if new; you’re first user every session (try a file with a typo in it). **17–18:** read a file, compute, and write a report file; use one standard-library module and its documentation; first user weekly.

Practice that actually builds learning

In-home — PY-03 Functions With Return Values

- **Time:** 25 minutes
- **Materials:** a computer with Python
- **Safety and privacy:** none needed
- **The fun:** building a tool once and using it everywhere
- **The skill:** def, parameters, return, and a docstring; calling one function from another
- **The crumb:** the program file

How it goes. Start with a predict-the-output on this program:

```
def double_print(n):
    print(n * 2)

def double_return(n):
    return n * 2

a = double_print(4)
b = double_return(4)
print(a, b)
```

It prints 8 (from inside `double_print`) and then `None` 8. The first function showed the answer and handed back nothing. The second handed back the answer, so `b` holds it. Then your student writes two small functions of their own, each with a one-line docstring (a short description in triple quotes, just under the `def`), and one function that calls the other:

```
def average(numbers):
    """Return the mean of a list of numbers."""
```

```

    return sum(numbers) / len(numbers)

def describe(numbers):
    """Return a one-line summary."""
    return f"{len(numbers)} items, average {average(numbers):.1f}"

print(describe([195, 312, 310]))

```

This prints 3 items, average 272.3. Ask, “What happens if the list is empty?” and let them try it. (average([]) stops with a ZeroDivisionError, which is a fine edge case for the spec.)

In-home — PY-04 Write a Report File

- **Time:** 30 minutes
- **Materials:** a computer with Python, a text file with some words in it
- **Safety and privacy:** use your own writing or a passage you made up
- **The fun:** a program that leaves something behind for a person to read
- **The skill:** computing totals and writing a formatted report to a new file
- **The crumb:** the report output file

How it goes. This program counts the words in a file and writes the five most common to a new file called report.txt:

```

counts = {}
with open("story.txt") as f:
    for line in f:
        for word in line.lower().split():
            counts[word] = counts.get(word, 0) + 1

top = sorted(counts.items(), key=lambda pair: pair[1], reverse=True)[:5]

with open("report.txt", "w") as out:
    for word, n in top:
        out.write(f"{word}: {n}\n")
print("Wrote", len(top), "lines")

```

With a story.txt of three short lines (“the cat sat on the mat,” “the dog sat on the log,” “A cat and a dog”), it prints Wrote 5 lines, and report.txt begins the: 4. Several words tie at 2, and the order of tied words depends on which came first in the file, which is a good conversation about what “top five” means when there’s a tie. Ask your student to open report.txt and check it by hand against the story. The “w” means “write,” and it replaces any old file with the same name, so a spec line about that is worth adding.

15–16: run it, predict the report, then change it to the top three. **17–18:** add a rule to the spec (ignore words shorter than three letters) and build it.

In-home — PY-05 Use a Library, Read Its Docs

- **Time:** 25 minutes

- **Materials:** a computer with Python and internet access, or the documentation saved for offline use
- **Safety and privacy:** official documentation only; no downloads from unknown sites
- **The fun:** finding a tool someone already built
- **The skill:** importing a standard-library module (csv, random, datetime) and finding the answer in the documentation
- **The crumb:** a docs note with the line that answered it

How it goes. Give your student a question that a standard-library module answers, and have them find the answer in Python’s official documentation, not a search result or a chatbot.¹⁵ Good questions: “How many days since we started school this year?” (the datetime module), “Pick a random chore from a list” (random), “Add up the minutes column in a spreadsheet file” (csv). Here’s a datetime answer:

```
from datetime import date

start = date(2026, 9, 8)
today = date(2026, 10, 12)
print((today - start).days, "days since we started")
```

This prints 34 days since we started. The crumb is a short note: the question, the page of the documentation, and the exact sentence on that page that answered it. At first this will feel slower than asking someone. Within a month, it will be faster, and it’s the skill that lasts.

In-home — PY-06 Paper Parsons, Python Edition

- **Time:** 10 minutes
- **Materials:** a working function printed out and cut into strips, plus one extra strip
- **Safety and privacy:** keep scissors for the parent at 15 if you like; otherwise none
- **The fun:** a puzzle with exactly one right answer and one trap
- **The skill:** reordering cut-up Python lines into a working function
- **The crumb:** the taped strip

How it goes. Print this function, cut it into one strip per line, and add one distractor strip such as `return word` indented inside the `if`:

```
def longest_word(words):
    best = ""
    for word in words:
        if len(word) > len(best):
            best = word
    return best

print(longest_word(["map", "lantern", "fox"]))
```

Your student puts the strips in order, including the indentation, explains in one sentence why the distractor doesn’t belong (it would hand back the first word longer than nothing, which is the first word, and stop), and tapes the result down. Then they run the real thing. It prints `lantern`. Paper

Parsons is the best first move for a student who freezes at a blank file. Chapter 4 brings it back in Java.

In-home — DS-01 Lists, Dictionaries, and When

- **Time:** 15 minutes
- **Materials:** six task cards, a computer for the second half
- **Safety and privacy:** none needed
- **The fun:** arguing for a choice
- **The skill:** choosing a list or a dictionary for a job and saying why
- **The crumb:** the choice card with reasons

How it goes. On each card, write a job: “the order people finished a race,” “how many times each word appears,” “a phone list where you look up a name,” “the next five chores in order,” “each student’s favorite color,” “the high scores, best first.” Your student sorts them into “list” and “dictionary” and writes one reason for each. Then they build one of each in Python and print it. Some cards have two defensible answers, and the reason is what counts.

In-home — DS-02 Dictionary Counter

- **Time:** 25 minutes
- **Materials:** a computer with Python
- **Safety and privacy:** use made-up votes or family-only choices
- **The fun:** settling a family argument with data
- **The skill:** counting words or items with a dictionary, and sorting the result
- **The crumb:** the program file

How it goes. Take a family vote on something low-stakes, such as dinner, and type the votes into a list:

```
votes = ["tacos", "pizza", "tacos", "soup", "tacos", "pizza"]
counts = {}
for v in votes:
    counts[v] = counts.get(v, 0) + 1
print(counts)
for food, n in sorted(counts.items(), key=lambda p: p[1], reverse=True):
    print(food, n)
```

This prints `{'tacos': 3, 'pizza': 2, 'soup': 1}`, then `tacos 3`, `pizza 2`, and `soup 1` on separate lines. Ask your student to explain `counts.get(v, 0)` out loud, which is the heart of it: “Look up this food; if it isn’t there yet, start at zero.” Then change the program to read the votes from a file instead.

In-home — DS-03 Nested Data on Paper

- **Time:** 15 minutes
- **Materials:** paper, colored pencils

- **Safety and privacy:** none needed
- **The fun:** drawing a map of data
- **The skill:** drawing a list of dictionaries (records) and tracing access paths
- **The crumb:** the diagram

How it goes. Real data often comes as a list of records, where each record is a dictionary:

```
books = [
    {"title": "Salt Road", "pages": 195, "tags": ["travel", "history"]},
    {"title": "Winter Fox", "pages": 310, "tags": ["animals"]},
]
print(books[1]["title"])
print(books[0]["tags"][1])
print(len(books[0]["tags"]))
```

Your student draws it first: a long box for the list, two boxes inside for the dictionaries, labeled arrows for each key. Then they trace each print line along the arrows and predict: Winter Fox, history, 2. Then they run it. Chapter 5 uses this exact shape for data.

17–18: draw it, then write three access paths of your own for your parent to trace. **Ready 16:** for any sixteen-year-old comfortable with lists and dictionaries.

Back to the trace table

ALG-03 from Chapter 1 comes back here with real programs. Before running any new loop in this chapter, especially one that reads a file, your student traces the first three times around the loop on paper. It takes two minutes, and it catches the blank-line bug before it happens.

Out-of-home — OUT-02 Library Computer Hour

- **Time:** about 50 minutes
- **Materials:** a library card, a USB stick or family cloud folder, today’s spec on paper
- **Safety and privacy:** log out of everything; save nothing to the library computer
- **The fun:** proving the program runs on a computer that isn’t yours
- **The skill:** a Computing Hour on a public computer; logging out; saving to your own drive
- **The crumb:** the file saved in two places

How it goes. This try-it is described in **The Computing Hour**; here it gets a Python job. Your student brings a spec for a small “read, compute, write” program and the data file on a USB stick, and builds it on the library computer in a browser-based Python editor, such as CodeAI’s Python Lab, if the library allows it. Files in browser editors sometimes behave differently from files on your own computer, which is its own good lesson about paths. At the end: save to the stick, check it opened, log out, close the browser.

15–16: you go together. **17–18:** alone, with a text to you when the file is saved in two places.

Talk box (this chapter)

Opening: “What goes in, what comes out — and what will this line do to it?”

Follow-ups: 1. **Prediction:** “What will total be after the second time around the loop?” 2.

Evidence: “What did the last line of the red text say, exactly?” 3. **Location:** “Which line is the last one that did what you expected?” 4. **Change:** “What’s the one line you’d change, and what do you predict will happen?” 5. **Revision:** “You can change your prediction. Run it again.”

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. In this chapter, it’s often that your student is reading the error from the top, where it’s least useful. Point gently: “Start with the last line.” Or there’s no written prediction, so there’s nothing to compare. Ask for one before the next run.

Don’t say / say instead. Don’t say “You forgot to convert it”; say “What type is that value when it comes in from the file?”

One wrong example to diagnose

Show your student this program and its output, and ask what’s wrong.

```
scores = [4, 7, 7, 2]
for s in scores:
    if s == 7:
        scores.remove(s)
print(scores)
```

It’s supposed to remove every 7. It prints [4, 7, 2]. One 7 survived. When the loop removed the first 7, everything after it shifted left by one, and the loop moved on to the next position, skipping the second 7 entirely. The lesson isn’t this particular bug; it’s the rule behind it: don’t change a list while you’re looping over it. Build a new list instead. Ask your student to write the fix and prove it with a list that has three 7s in a row.

Blocked, then mixed

For the first two weeks, practice one card type at a time: a week of file reading (PY-01, PY-04), a week of functions (PY-03, PY-06). Then mix. Each Hour starts with a two-minute predict-the-output on something from a different card than the day’s main work. Mixing feels harder, and that’s the point; your student is learning to choose the tool, not just use the one in front of them.

For the student

This section is for you.

In this chapter, Python stops being something you’re learning and becomes something you use. The test of a tool is whether it does a job you care about. So let’s start with one.

Reading a file

Most useful programs start by reading something. A file is just text stored on your computer, and Python can open it, go through it one line at a time, and do something with each line. Here's the shape:

```
with open("reading_log.txt") as f:
    for line in f:
        print(line.strip())
```

`open` finds the file. `with` makes sure it gets closed when you're done. The `for` loop hands you one line at a time. `strip()` removes the spaces and the invisible end-of-line character from each end. Every file program you write this year will start something like this.

A worked example

Say your reading log has three books and a blank line at the end. You want the total pages. Your spec says "Done when it prints the correct total for my real file and doesn't crash on a blank line." You predict 817. You run the first version, and it crashes.

The red text is called a traceback (Python's report of where an error happened). Read it from the bottom up. The last line names the error and gives a short explanation. The line above it shows you which line of your code was running. Here, it says `ValueError: not enough values to unpack (expected 2, got 1)`, and it points at the line that splits on the comma. Expected two pieces; got one. A blank line has no comma. That's the bug.

Write it in your Debug Log before you change anything: "Expected 817. Actual: `ValueError` at the split line. Guess: the blank line. Try: skip empty lines." Then change one thing. Run it. 817.

One program to write today

Write a reading-log totaler for your own reading, or a practice-minutes totaler, or anything with "name, number" on each line. Write the spec first, with three examples: an ordinary file, a file with a blank line, and an empty file. Predict the output before every run. When it works, add one more feature from your own list: the longest book, the average, or a report written to a new file.

Explain it back

When it works, cover the screen and explain each line to your parent, out loud, in order. Anything you can't explain, look up in the documentation until you can. That's not a test you can fail. It's how the program becomes yours.

This part is hard for everyone

Paths and file names. You will, at some point this year, be completely certain the file is there, and Python will tell you it isn't. You'll see `FileNotFoundError`. Almost always, the program is running from a different folder than the one the file is in. Keep the program and its files in the same

folder. If it still fails, add `import os` and `print(os.getcwd())` at the top. That prints the folder Python is looking in. You'll often find the answer in one line.

Also: text from a file is always text. If you read "312" and want a number, you need `int()`. It'll forgive spaces around the number. It won't forgive an empty string or a decimal point.

Read the documentation

When you need to know how something works, try Python's official documentation before anything else. It's free, it's correct, and it's written for programmers. At first it'll feel dense. Look for the examples near the top of each section. Write down the sentence that answered your question. A month from now, you'll be faster at this than at asking anyone.

A challenge

Write a program that reads any text file and builds a dictionary of word counts, then writes the ten most common words to a report file, ignoring capital letters. Spec it first. What counts as a word? Is "dog," with a comma, the same as "dog"? What should happen with ties? Decide, write it in the spec, and build to it. When you're done, run it on something you wrote and find out which word you use most.

If it isn't clicking

What you see	What it usually means	Next move
print where return was needed	Output and value are mixed up	Trace two calls on paper; PY-03 with a check that uses the return value
"The file doesn't work"	The path or working folder is wrong	Print the folder with <code>os.getcwd()</code> ; keep program and data together; Chapter 6 draws the folder tree on paper
Can't start from a blank file	Reading has outpaced writing	Paper Parsons (PY-06), then modify a working program before writing one fresh
Reads the traceback from the top	The habit hasn't formed yet	Cover everything but the last line; read it aloud first, every time

When your student is stuck in the moment, use the rescue order: ask → wait → hint → short demo on your own copy → hand it back. In this chapter, the best hint is usually "Read me the last line of the red text." The demo, if you need one, is a three-line program on your own file that shows the same idea, closed before you hand back the turn.

If Python itself isn't clicking after several weeks, slow down rather than speed up. Spend a week on predict-the-output and Paper Parsons only. Reading code fluently comes before writing it fluently, for everyone. If you'd like outside help, a co-op class or a tutor for one session is ordinary; professionals learn from each other all the time.

Tools, including AI

Editors. Thonny for students new to text code; any Python editor after that. A browser-based editor, such as the Python Lab from CodeAI (formerly Code.org), for library Hours. The book never names a Python version; any current one works.

Documentation. Python's official documentation is free and is the first place to look.

AI. The AI box in **The Computing Hour** sets the rules. What's specific here: at 15–16, after your student has written Expected/Actual and a guess, a chatbot may explain a traceback or an idea. Ask it to explain, not to fix. The editor assistant stays off through 16, and AI is off for every check. Because the documentation is the skill this chapter teaches, have your student check anything a chatbot tells them about Python against the documentation. If they disagree, the documentation wins. Check the current terms before using any service.

What “done enough” looks like

Your student is ready to move on when they can:

- ☐ write a program that reads a file, computes something, and prints or writes a result, built to their own spec (PY-01, PY-04);
- ☐ predict the output of a short program in writing and explain any mismatch (PY-02);
- ☐ write a function that returns a value, with a docstring, and explain the difference between print and return (PY-03);
- ☐ find an answer in Python's documentation and write down the sentence that answered it (PY-05);
- ☐ reorder a cut-up function correctly (PY-06);
- ☐ choose a list or a dictionary for a job and say why, and count with a dictionary on purpose (DS-01, DS-02);
- ☐ run a Computing Hour on a library computer and leave the file in two places (OUT-02);
- ☐ **AS-01 Cold Trace:** this month's, alone, dated;
- ☐ **AS-03 Explain Your Code:** explain one of their own programs line by line to you, without notes or AI.

At 17–18 (or ready 16), add: draw a list of dictionaries on paper and trace three access paths (DS-03).

A student who moves through this chapter quickly and can explain every line is ready for Chapter 3, where tests make all of this provable. A student who's still unsure about files and functions should stay another week or two. Place by skill, not birthday.

Chapter 3

Testing and Debugging: Proof Before Pride



Why this matters

Here is an illustration, not a real family. Picture a seventeen-year-old who has written a function that grades a quiz. She has run it four times on four answer sheets she typed herself, and it gave the right percent every time. She is proud of it, and she should be. Then her father sits down as first user, leaves every answer blank, presses Enter, and the screen fills with red text. Her face falls.

Thirty seconds ago the program was finished. Now it feels like it was never any good.

Nothing about the program changed in those thirty seconds. What changed is the evidence. Four hand-picked inputs proved that the function works on four hand-picked inputs. They didn't prove it does what the spec says, because the spec (if she wrote it the way Chapter 1 taught) promised something about every answer sheet a real user might hand in, including an empty one.

This chapter is about the difference between “it ran” and “it's right.” A **test** is a check, written down, that compares what the program does with what it should do. A set of tests is written evidence. Debugging is what you do when the evidence says no. The chapter's subtitle is the order of events: proof first, then pride. Pride that comes after proof lasts. Pride that comes before it breaks the first time a stranger types something odd.

It is also the third clause of the book's focus line, **Specify it before you build it; predict before you run; debug before you rewrite; explain it before you call it yours.** Chapters 1 and 2 built the first two habits. This one builds the third. A student who meets red text by deleting the file and starting over learns almost nothing from the bug. A student who writes what they expected, reads what happened, makes one guess, changes one thing, and logs it learns something every time, and gets faster every month.

There's a feeling side to this, and at 15–18 it's real. Perfectionists hear every failing test as a verdict; quick students change five lines at once because waiting feels like losing; students praised as “good at computers” quietly fear the day the computer proves otherwise. A failing test isn't a grade. It's the bug's address. Your calm is part of the method, and it may be the most important part.

Testing is craft, not a learning trick. This book doesn't promise it makes anything else easier, only that your student's programs will do what they say.

For the parent: understand it yourself

You don't need to program to run this chapter. You need to recognize three kinds of error, know how to read the red text in the right order, and be able to ask “What test would prove that?” in a calm voice. All three take about an evening.

The everyday picture. A recipe card says “bake until done.” Two cooks get two different cakes. Now the card says “bake 35 minutes at 350°F; 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's holding the toothpick. A test table is a page of toothpicks for a program: for this input, it should give this output. If it doesn't, you know exactly which promise was broken.

The precise picture. The national high-school standards from the Computer Science Teachers Association describe the goal in one sentence: “Evaluate a computing technology's alignment with design specifications and responsible design values, including its correctness, effectiveness, and user experience.”¹⁶ Read the middle of that slowly. *Alignment with design specifications.* The spec from Chapter 1 is the measuring stick, and the tests are how you hold the program up against it. A companion standard asks students to evaluate algorithms “for efficiency, correctness, and clarity, using metrics or test cases.”¹⁷ Test cases are the everyday tool for correctness.

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 colon or a bracket is missing, 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 it runs, finishes, and gives the wrong answer. No red text. Only a prediction or a test catches it.

Here is a syntax error. The colon at the end of the first line is missing:

```
def total(prices)
    return sum(prices)
```

Python stops before running anything and reports `SyntaxError: expected ': ', with a small caret under the spot where it gave up. Add the colon and the error goes away.`

Here is the quiz grader from the opening, with its runtime error. This first version divides by the number of answers the student gave:

```
def score_quiz(answers, key):
    """Return the percent of answers that match the key."""
    right = 0
    for given, correct in zip(answers, key):
        if given == correct:
            right += 1
    return round(100 * right / len(answers))

print(score_quiz(["b", "c", "a"], ["b", "c", "d"]))
print(score_quiz([], ["b", "c", "d"]))
```

The first call prints 67. The second stops the program with a traceback whose last line is `ZeroDivisionError: division by zero`.

And here is a logic error, labeled as a deliberate bug because it's the one the boundary hunt below is built to find:

```
def in_band(age):
    """True for ages 15 to 18, inclusive."""
    return 15 < age <= 18 # DELIBERATE BUG: 15 is left out
```

It runs without complaint. `in_band(16)` gives `True`, `in_band(18)` gives `True`, and `in_band(15)` gives `False`, which the docstring says is wrong. Nothing turns red. Only someone who tested the boundary would know.

How to read a traceback

A **traceback** is Python's report of where an error happened, read from the bottom up. Here is the grader's, exactly as it appeared when we ran it (the file path at the top will match your own folder,

not ours):

```
Traceback (most recent call last):
  File "grader.py", line 10, in <module>
    print(score_quiz([], ["b", "c", "d"]))
  File "grader.py", line 7, in score_quiz
    return round(100 * right / len(answers))
ZeroDivisionError: division by zero
```

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

1. **The last line** says what happened: the error’s name (`ZeroDivisionError`) and a message (division by zero).
2. **The line just above it** is where it happened: line 7, inside `score_quiz`.
3. **Further up** is how the program got there: line 10 called the function with an empty list.

The Python tutorial says this plainly: the last line of the error message indicates what happened, and the place Python reports isn’t always the place that needs fixing.¹⁸ A missing bracket on one line is often reported on the next. So the habit is: last line, then the line number, then look at that line and the one above it.

Tests in plain Python

Your student doesn’t need a testing framework this year. Python has a built-in statement called `assert`: it checks that something is true and stops the program loudly if it isn’t. Here’s the boundary bug caught by three asserts:

```
assert in_band(16) == True
assert in_band(18) == True
assert in_band(15) == True, "15 is in the band"
print("All tests passed.")
```

The first two pass silently. The third stops the program with `AssertionError: 15 is in the band`, and “All tests passed.” never prints. That’s a test doing its job: it failed loudly, on the exact promise that was broken. Changing `15 < age` to `15 <= age` makes all three pass and the last line print.

At 17–18 (ready 16), the asserts move into their own file, named something like `test_grader.py`, and the student runs that file after every change. Python also ships with a testing framework called `unittest` that does the same with nicer reports; your student may meet it later.¹⁹

The debugging loop

Professional debugging is a short loop, and every step is teachable:

1. **Expected/Actual.** Write it before touching anything: “I expected 0. It crashed.”
2. **Read the clue.** The traceback’s last line, or the wrong output.
3. **One hypothesis.** “It divides by the number of answers, and there are none.”

4. **One change.** Only one.
5. **Run the tests again.** All of them, not just the one you were fixing.
6. **Log it.** One Debug Log entry.

The “one change” rule comes from working practice. A well-known book for professional developers notes that if you changed only one thing and the system stopped working, that one thing is likely responsible.²⁰ Change three things and you’ve learned nothing, even if it now works.

The Debug Log is the record of that loop. You met it in **The Computing Hour**; this chapter is where it earns its keep. Each entry is dated and holds the symptom, the Expected/Actual line, the hypothesis, the test that checks it, the fix, the lesson, and the minutes spent. The minutes feed the hours log that Chapter 9 turns into credit.

Bisecting. When a bug hides in a long program, or appeared somewhere in last week’s changes, the fastest method is halving. Comment out (or skip) the second half: does the bug still happen? Then it’s in the first half. Halve again. Ten halvings narrow a thousand lines to one. At 17–18, the same idea works on history: Chapter 6 shows how Git can jump to the midpoint of a week of commits and ask “good or bad?”

The rubber duck. Many programmers explain their code, line by line, out loud, to a listener who doesn’t need to understand it, even a rubber duck on the desk. It’s working practice, not a research finding; the same developers’ book says that explaining what the code is supposed to do “often causes the problem to leap off the screen and announce itself.”²⁰ You’re a very good rubber duck. You just have to listen without fixing.

Wrong answers you should be able to hear

- **“It works on my input.”** It ran once, on one example the student chose. Say: “Show me the test that proves it. What about the empty one?”
- **“That test is wrong, so I deleted it.”** Sometimes a test really is wrong, but usually it’s inconvenient. Say: “What did the spec promise for that input? If the spec changed, change the spec first, then the test.”
- **“I changed a few things and now it works.”** No one knows which change mattered. Say: “Which one fixed it? Put the others back and find out.”
- **“The computer is wrong.”** It did exactly what it was told. Say: “Read me the last line of the red text.”
- **“I’ll just rewrite the whole file.”** One error has become a reason to throw away an hour of work and the clue with it. Say: “What’s in today’s Debug Log so far?”

Your five-minute warm-up

Before tomorrow’s Hour, type the quiz grader above into the student’s editor and run it. Find the last line of the traceback and read it aloud. Then find line 7. Then say this chapter’s opening question once, out loud, to an empty room: “What did you expect, what happened instead — and what test would prove the fix?” Practice the silence after it, counting to five in your head.

How to teach it this week

Run the week inside the six steps of **The Computing Hour**. Here's how each step leans toward testing.

- **Specify (5–10 minutes)**. The spec gets a test table: three or more rows of input and expected output, one of them an edge case. If last week's spec had examples (SPEC-02 Examples Before Code from Chapter 1), they become the first rows.
- **Predict (2–5)**. “Which of these tests will fail first, and why?” Written.
- **Build and Test (15–16: 20–35; 17–18: 30–50)**. Code in small steps. Run the tests after every change, not only at the end.
- **Debug and Log (5–15)**. Expected/Actual, hypothesis, one change, a Debug Log entry.
- **Review (5–10)**. Read the change against the spec. Did a test get deleted? Did the spec quietly change? Commit test and fix together with a real message (GIT-03 Good Commit Messages, Chapter 6, previews the habit).
- **Ship and Explain (3–5)**. “I specified... I predicted... It did... Next I'd change...” and “Help I used: ...”

Monday: the test table before code. Start on paper, with no screen at all.

In-home — TEST-01 Test Table Before Code

- **Time:** 15 minutes
- **Materials:** the student's current spec, a sheet of paper ruled into four columns (input, expected, actual, pass?), a pencil
- **Safety and privacy:** none needed; use made-up data
- **The fun:** you get to try to break something before it exists
- **The skill:** listing normal, boundary, and bad inputs with expected results, written first
- **The crumb:** a test table, dated

How it goes. The student reads the spec's done line aloud. Then they fill the first two columns: at least two normal inputs, at least one boundary (the smallest, the largest, the empty), and at least one bad input a careless user might type. The “actual” and “pass?” columns stay blank. They're filled in later, on the screen.

Try this exact sentence: “Before you write the function, write what it should say for an empty sheet.” Then wait.

15–16: You read the table and ask one question, usually “What would a stranger type here?” Initial it with the spec. **17–18:** The student writes it alone; you read it the same day and add one row they didn't think of. If they can't decide what the right answer *should* be for a row, that's a hole in the spec, not in the table. Send them back to fix the spec.

Tuesday: asserts. The test table goes into code.

In-home — TEST-02 Asserts That Fail Loudly

- **Time:** 20 minutes

- **Materials:** the editor, Monday’s test table
- **Safety and privacy:** none needed
- **The fun:** watching a test catch a mistake you made on purpose
- **The skill:** writing an assert for each test-table row and reading a failing assert
- **The crumb:** a test file, or the asserts at the bottom of the program

How it goes. One row of the table becomes one assert line. The student writes all of them before running anything, then predicts which will fail. Then they break the function on purpose (change `<=` to `<`, say), run again, and read the `AssertionError`. A test that has never been seen failing hasn’t been proven to work.

15–16: asserts at the bottom of the program file are fine. **17–18:** asserts start moving into their own file (TEST-03, below).

Wednesday: the traceback. Use a program with a real crash, ideally one from the student’s own week.

In-home — BUG-02 Read the Traceback

- **Time:** 15 minutes
- **Materials:** a printed traceback (or one on screen), an index card, a colored pencil
- **Safety and privacy:** none needed
- **The fun:** decoding a message that looked scary a minute ago
- **The skill:** reading a Python traceback bottom-up: error type, message, file, line
- **The crumb:** a traceback decoder card

How it goes. On the card the student writes four labels: *error type*, *message*, *file* and *line*, *how we got here*. They circle each one on the printed traceback in a different color, starting from the bottom. Then they say, in one sentence of their own, what Python is complaining about. Only then do they open the file. Keep the card by the keyboard for the rest of the year.

15–16: do it together the first time, with you reading the last line aloud. **17–18:** the student explains a traceback to you, as if you were a new programmer on their team.

Thursday: the planted bugs. Someone else plants the bugs. This is where siblings are wonderful.

Friday: first user. You sit at the keyboard as a stranger (see the try-it below). Then the student turns every surprise into a new test row and makes it pass.

Fading help. In week one, say “Expected/Actual first” every time; by week three, point at the card; by week five, say nothing. When they reach for the rescue, use the rescue order from **The Computing Hour**: ask → wait → hint → short demo on your own copy → hand it back. A “hint” in this chapter is almost always one of three questions: “What does the last line say?”, “Which test fails?”, or “What’s the smallest input that breaks it?”

Practice that actually builds learning

Practice here has a rhythm: blocked first (one kind of test, one kind of bug, several times), then mixed (a program where the student has to decide what kind of problem they're facing).

In-home — TEST-04 Boundary Hunt

- **Time:** 15 minutes
- **Materials:** a spec with a range in it, paper, pencil
- **Safety and privacy:** none needed
- **The fun:** finding the input the author forgot
- **The skill:** finding off-by-one and empty-input boundaries in a spec
- **The crumb:** a boundary list

How it goes. Give the student a one-line spec: “A program says whether a person is in the book’s age range, 15 to 18 inclusive.” Ask: which inputs matter? A good list has 14, 15, 18, and 19 (each side of each edge); a typical middle value like 16; an empty entry; a word (“sixteen”); and a negative number. Then hand them the deliberately buggy `in_band` function from the parent section and let them find which item on their list catches the bug. (It’s 15.) When we ran the corrected version on 14, 15, 18, 19, and -1 , it gave False, True, True, False, False. The word “sixteen” makes it crash with a `TypeError`, because Python won’t compare a number and text. That’s a second finding: the spec never said what to do with words. Fix the spec first.

15–16: the parent reads the spec aloud and the student calls out boundaries. **17–18:** the student writes the boundary list for a spec someone else wrote.

In-home — BUG-03 Planted Bugs Swap

- **Time:** 20 minutes
- **Materials:** a working copy of a program the student wrote, a second person (parent, sibling, co-op friend)
- **Safety and privacy:** always plant bugs in a copy, never the only version
- **The fun:** revenge, gently
- **The skill:** finding three bugs someone else planted in working code
- **The crumb:** a bug list with line numbers

How it goes. The planter makes a copy of a working program and plants exactly three bugs: one syntax error, one runtime error, and one logic error. (A parent who doesn’t code can still do this: delete a colon, change a `len(key)` to `len([ ])`, change a `<=` to `<`.) The student finds all three, writing an Expected/Actual line and a Debug Log entry for each. The logic bug is always the hardest, which is the whole lesson: the bugs that don’t make noise need tests.

15–16: you plant; they hunt. **17–18:** the student plants bugs for *you*, then coaches you through finding them using the rescue order.

In-home — BUG-01 Expected/Actual Line

- **Time:** 5 minutes, every time something surprises anyone
- **Materials:** the Debug Log
- **Safety and privacy:** none needed
- **The fun:** it turns “ugh” into a sentence
- **The skill:** writing what you expected and what happened before touching code
- **The crumb:** a Debug Log line

How it goes. You know this card from **The Computing Hour**. This week it becomes a rule: no keyboard until the Expected/Actual line is written.

In-home — BUG-05 The Debug Log

- **Time:** 5 minutes at the end of each Hour
- **Materials:** a notebook or a single plain file, dated entries
- **Safety and privacy:** keep names of friends and family out of it; it may end up in a portfolio
- **The fun:** in a month it reads like a detective’s casebook
- **The skill:** a dated log of symptom, Expected/Actual, hypothesis, test, fix, lesson, and minutes
- **The crumb:** a monthly log page

How it goes. Here is one entry, as an illustration:

14 Oct. Symptom: grader crashes on an empty answer sheet. Expected: 0. Actual: ZeroDivisionError on line 7. Hypothesis: it divides by the number of answers, which is zero. Test: `assert score_quiz([], KEY) == 0` (fails before the fix). Fix: divide by the number of questions in the key. Lesson: the percent is out of the questions, not out of what the student wrote. 55 minutes.

That last line, the lesson, is what makes a log worth keeping. At the end of each month the student reads the month’s entries and writes one sentence about the bug they meet most often. Chapter 9 uses those pages as evidence for credit.

In-home — TEST-03 A Real Test File

- **Time:** 30 minutes
- **Materials:** the editor, a terminal or the editor’s Run button
- **Safety and privacy:** none needed
- **The fun:** one command, a whole page of promises checked
- **The skill:** moving asserts into a test file and running them every change
- **The crumb:** a `test_` file and a short run log

How it goes. The function lives in `grader.py`, with no `print` calls at the bottom. Here’s the fixed version:

```
def score_quiz(answers, key):
    """Return the percent of questions answered correctly."""
```

```

right = 0
for given, correct in zip(answers, key):
    if given == correct:
        right += 1
return round(100 * right / len(key))

```

The tests live in `test_grader.py`, in the same folder:

```
from grader import score_quiz
```

```
KEY = ["b", "c", "d"]
```

```

assert score_quiz(["b", "c", "d"], KEY) == 100    # all right
assert score_quiz(["b", "c", "a"], KEY) == 67     # two of three
assert score_quiz(["a", "a", "a"], KEY) == 0      # none right
assert score_quiz([], KEY) == 0                  # empty sheet
assert score_quiz(["b"], KEY) == 33              # stopped early

```

```
print("All 5 tests passed.")
```

Running `python3 test_grader.py` (or pressing Run with the test file open) prints All 5 tests passed. With the first version of the grader, the same file stops on the empty-sheet test with `ZeroDivisionError`. That’s the order to teach: write the test that reproduces the bug, watch it fail, fix the code, watch it pass, and keep the test forever so the bug can’t sneak back.

17–18: every project from here on has a test file, and “done” includes “the test file passes.” **Ready 16:** a 16-year-old who’s comfortable with files from Chapter 2 can start this now. One thing to notice together: what should happen if the *key* is empty? The spec doesn’t say. That’s the next row.

In-home — TEST-05 First User, Weekly

- **Time:** 15 minutes
- **Materials:** the student’s program, you, a note card
- **Safety and privacy:** use made-up data
- **The fun:** you get to be the most confusing user in the world
- **The skill:** the parent uses the program like a stranger and reports, without fixing
- **The crumb:** a first-user note

How it goes. You’ve been first user since **The Computing Hour**. This week, add one rule: everything surprising you find becomes a test row before it becomes a fix. Use it like a stranger: press Enter with nothing typed; type a word where a number belongs; type a huge number. Report what you saw in one plain sentence (“When I left it blank, red text filled the screen”). Don’t diagnose. Don’t touch the keyboard after your turn. The student writes the test, then the fix.

15–16: every session. **17–18:** weekly, with the student choosing which feature you test.

In-home — BUG-04 Bisect the Bug

- **Time:** 20 minutes

- **Materials:** a program of 40 lines or more with a bug the student can't find, or a week of saved versions
- **Safety and privacy:** work on a copy
- **The fun:** a bug in a thousand lines found in ten steps
- **The skill:** halving the code or the history to find where the bug starts
- **The crumb:** bisect notes

How it goes. The student writes down a test that shows the bug. Then they disable the second half of the program (comment it out, or return early from the function) and run the test. Still broken? The bug is in the first half. Fixed? It's in the second. They write down each step: "Lines 1–40: bug. Lines 1–20: no bug. Lines 21–30: bug..." until one line is left. With saved versions, the same idea works on time: test Monday's version and Friday's, then Wednesday's, and so on. Chapter 6 shows the Git command that automates this.

17–18: the default method for any bug that survives one Hour. **Ready 16:** a 16-year-old with tidy, numbered saved versions can do the history version now.

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

- **Time:** 20 minutes
- **Materials:** a pocket notebook
- **Safety and privacy:** watch machines, not people; no photos of strangers; first names only if you talk to staff
- **The fun:** finding bugs in the real world
- **The skill:** writing a spec for a self-checkout, an elevator, or a traffic signal from observation
- **The crumb:** a spec page

How it goes. In Chapter 1 the student watched a public machine and wrote its spec. This week they go back out hunting for a *bug in the world*: a confusing sign, a parking-payment kiosk that rejects a valid card, or a form that asks for a middle name and won't accept "none". They write it as Expected/Actual ("I expected the kiosk to take a card after I chose 'pay.' It asked me to choose 'pay' again"), then a guess about the cause, then a test that would prove the fix. The point is noticing that every confusing thing in public is a failed test someone didn't write.

15–16: go together and compare notes. **17–18:** the student goes alone and brings back two.

Talk box (this chapter)

Opening: "What did you expect, what happened instead — and what test would prove the fix?"

Follow-ups: 1. **Prediction:** "Before you run it again, which test do you think will fail now?" 2. **Evidence:** "What does the last line of the traceback say, in your own words?" 3. **Location:** "Which line is it pointing at, and what's on the line above?" 4. **Change:** "What's the one change you'll make first? Just one." 5. **Revision:** "Now that it passes, what new test would break it?"

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. At this age, often that the student is embarrassed the bug was simple. Sometimes it means they haven't written the Expected/Actual line yet, so there's nothing to compare. Sometimes the screen is louder than the question; lower the lid for a moment and ask again. And sometimes it means they already know the answer and are waiting to see whether you'll fix it for them, because you have before.

Don't say / say instead. Don't say "Did you even test it?"; say "Which test would have caught this? Let's write it now."

An incorrect example to diagnose

Show the student this test, labeled as a broken test:

```
def double(n):
    return n + n

assert double(3) == double(3)  # BROKEN TEST: it can never fail
print("Passed.")
```

It prints Passed. every time. Ask: "If double were completely wrong, say it returned $n * 3$, would this test catch it?" It wouldn't. The test compares the function with itself. A test has to compare against an answer the student worked out independently, on paper, before running: `assert double(3) == 6`. A test that can't fail proves nothing. Spotting it is a real skill.

Mixed practice

Once each card has been done once, mix them. Hand the student a short program with no hint about what's wrong, and let them decide: is this a syntax error, a runtime error, or a logic error? Do I need the traceback card, or a new test, or both? Keep the monthly AS-01 Cold Trace from Chapter 1, and once a week ask for the three kinds of error and the traceback reading order from memory.

For the student

You will spend a lot of your programming life finding mistakes. Everyone does; it's what the job is. The people who get good aren't the ones who make fewer mistakes. They find them faster, because they have a method.

The method has two halves. **Tests** are how you know whether there's a problem. **Debugging** is how you find and fix it once you know.

Tests first. A test is a written promise: "For this input, my program should give this output." You write it before you write the code, while your head is still on what the program *should* do instead of what it *does* do. That order matters. Once you've written the code, your brain starts defending it. Before, you're still honest.

A test table looks like this:

Input	Expected	Actual	Pass?
["b", "c", "d"]	100		
["b", "c", "a"]	67		
[] (empty sheet)	0		
["b"] (stopped early)	33		

Leave the last two columns blank until you run it. The empty sheet is the edge case, the input at the boundary or the odd one a careless user might type. It's the row that catches the bug, almost every time.

Then asserts. Each row becomes a line of Python:

```
assert score_quiz([], KEY) == 0      # empty sheet
```

If it's true, nothing happens. If it's false, the program stops and tells you which promise you broke. Write all your asserts, predict which will fail, then run them.

Try this today. Pick a function you wrote in Chapter 2. Write four test rows for it: two normal, one boundary, one bad input. Turn them into asserts. Then break the function on purpose (change one symbol) and check that at least one test catches it. If none do, your tests aren't testing anything yet. Fix the tests, not the function.

When it breaks. You'll see red text. Here's the order:

1. Write the Expected/Actual line before you touch anything. "I expected 0. It crashed."
2. Read the **last line** of the traceback. That's what happened.
3. Read the line just above it. That's where.
4. Make one guess about why.
5. Change **one thing**. Run *all* the tests.
6. Write a Debug Log entry: symptom, Expected/Actual, hypothesis, test, fix, lesson, minutes.

Step 5 is the one people skip. Change three things at once and, even if it works, you won't know which change fixed it, and one of the others may have quietly broken something. One change, one run. It feels slower. It's much faster.

Before you rewrite. At some point you'll want to delete the file and start over. Sometimes that really is the right call. But make it a decision, not a reaction. Read your Debug Log first. If you have a hypothesis you haven't tested, test it. If you've tested three and they all failed, then maybe a rewrite makes sense, and now you know three things the new version has to get right.

Explain it back. Pick one bug you fixed this week. Explain to someone (a parent, a sibling, or a rubber duck on your desk) what the bug was, how you found it, and why your fix works. If you can't say why the fix works, you got lucky, and luck doesn't repeat.

This part is hard for everyone. Thinking of edge cases is hard. Your brain wants to test the inputs you were already imagining when you wrote the code, which are exactly the inputs that work. Use your first user. When a parent types nothing and presses Enter, they're doing you a favor.

About pride. It feels great when all the tests pass. Enjoy it. That feeling is earned. But notice the order: proof, then pride. If a test fails and it stings, remember that the test just saved you from a stranger finding it. A failing test is the bug's address.

If you're 17 or older: keep your tests in their own file, `test_` plus the name of the file it tests, and run it after every change. When you fix a bug, write the test that reproduces it *first*, watch it fail, then fix the code and watch it pass. Commit the test and the fix together. If you ever use an AI coding assistant on your one named project (see the AI box in **The Computing Hour**), every line it suggests gets the same treatment as your own code, and more: you explain each line before you keep it, and your tests decide whether it works, not the tool's confidence.

Challenge. Take the best function you've written so far. Write three new tests designed to break it. If you can't break it, ask your first user to try. If they break it, write the test, fix the function, and log it. That's a good week.

What to save: your test table (dated), your test file or asserts, and this week's Debug Log pages. Save them in two places.

If it isn't clicking

The student deletes failing tests. Usually tests feel like grades, and a failing one feels like a bad mark. Reframe it out loud: "A failing test isn't a grade. It's the bug's address." Make a family rule that no test gets deleted without a one-line note in the Debug Log saying why, and a matching change to the spec if the expected answer changed.

The student changes several lines at once. That's usually panic, impatience, or both. Slow the loop down by saying it out loud together for a week: "One change. Run. What happened? Log it." If they really want to try several ideas, they can list them first in the log, then try them one at a time.

The same bug keeps coming back. There's no regression test, a test that stays in the file after a fix so the bug can't quietly return. Next time it appears, the first move is to write the test that would have caught it, watch it fail, and keep it forever.

Red text causes a shutdown. The student freezes, or walks away, or says "It's broken." Go back to BUG-02 with a printed traceback and the colored pencils. Read only the last line together. Say, in your calmest voice, "That's a clue, not a verdict." Then follow the rescue order: ask → wait → hint → short demo on your own copy → hand it back. If you demo, demo on your own copy of the file, and hand back the student's copy untouched.

When to get outside help. Three Hours stuck on one bug, with honest log entries, is a good moment for a second pair of eyes: a co-op class, a tutor, or an adult who codes at work (OUT-03 in Chapter 6). Bring the Debug Log.

Tools, including AI

Your student's editor is enough: Thonny for 15–16 if it's still new, or any free code editor with a terminal at 17–18. Thonny's step-through mode, which runs a program one step at a time so you can watch the variables change, is a fine debugging tool at any age.¹³ Adding temporary `print()` lines to show what a variable holds is also real debugging, as long as the student removes them afterward.

For AI, the rules are the ones in the AI box in **The Computing Hour**, and they matter especially here. Chat-style help may explain an error message only after the student has written their own Expected/Actual line and guess. At 17, on the one named project, a suggested fix stays only after the student explains every line of it (AI-05 Explain Every Line, Chapter 8) and the full test file passes. AI is off for every check, including AS-02 and AS-06 below. The tests, not the tool, decide whether code works. Check the current terms.

What “done enough” looks like

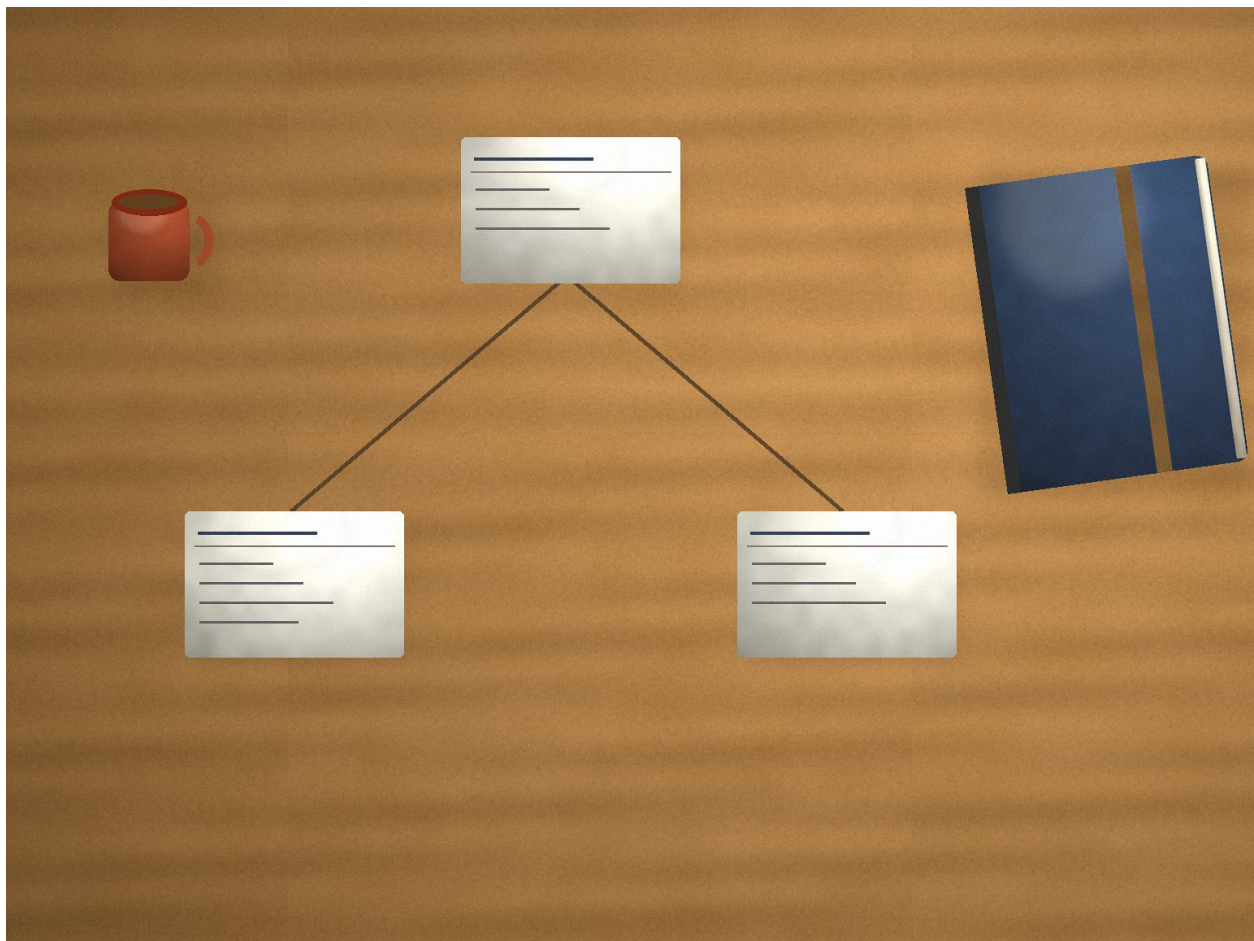
Move on when most of these are “seen,” a few are “growing,” and none of the core four (the first four) is “not yet”:

- ☐ Writes a test table, including a boundary and a bad input, before writing code (TEST-01, TEST-04)
- ☐ Turns test rows into asserts and has seen a test fail on purpose (TEST-02)
- ☐ Reads a traceback bottom-up and names the error type, message, and line (BUG-02)
- ☐ Writes Expected/Actual before touching code, and keeps a Debug Log with lessons (BUG-01, BUG-05)
- ☐ Has found three planted bugs, one of each kind (BUG-03)
- ☐ Turns first-user surprises into tests before fixes (TEST-05)
- ☐ **AS-02 Debug on the Spot:** fixes a bug you planted while explaining aloud, with no AI and no help. You write one line about how it went.
- ☐ **AS-06 Spec-to-Test Check:** given a short spec written by someone else, writes a test table for it, unaided, in 15 minutes. Date it and keep it for Chapter 9.
- ☐ **17–18:** keeps a test file for every project and runs it after every change (TEST-03); has bisected at least one bug by halves (BUG-04, below).

Placement. Place by skill, not birthday. A 15-year-old who writes tests first and logs calmly is ready for Chapter 4. A 17-year-old who still deletes failing tests needs another week here, and that's not a failure.

Chapter 4

Objects and Classes: The Bridge to Java



Why this matters

Here is an illustration, not a real family. Picture a sixteen-year-old who has spent three weeks on a program that tracks the family's library books. It works. It also has six separate lists (titles, authors, pages, due dates, who borrowed it, whether it's back), and every function takes all six as arguments. When he adds a seventh list for "rating," he has to change eleven lines in nine places, and he misses

one. The program doesn't crash. It just quietly starts pairing the wrong rating with the wrong book.

He doesn't need to work harder. He needs a better shape. What he's missing is the idea that a book is one *thing*, with its own facts and its own actions, and that the program should be built out of things like that. That idea is called an **object**, and the blueprint for making objects of one kind is called a **class**. A class is a blueprint; an object is one thing built from it. The `Book` class says every book has a title and a page count and can describe itself. Each actual book on the shelf is an object.

Classes are the biggest new idea of high-school programming. They change how a student thinks about a program: not a long list of steps, but a small cast of characters, each responsible for what it knows. They make big programs readable, and they make changes safer, because the code that knows about pages lives in one place.

Classes are also the bridge to a second language. This chapter builds the same small program in Python, then in **Java**. Java is a real strand of this book, not a sidebar. It's widely used in industry and in college courses, and it makes visible things that Python keeps quiet: what **type** each value is (a whole number, text, a true-or-false), where each block of code starts and stops, and the moment when the whole program is checked before it runs. Java **compiles**: it translates the whole program first, and refuses to run if something doesn't fit. Python doesn't; it finds problems as it reaches them. Learning both teaches a student what a programming language *is*, which is something no single language can teach.

There's an honest limit to say once. Object-oriented programming isn't "the right way" to program. It's one way, common in industry and central to the AP Computer Science A course, which is taught in Java. Other styles exist and are good. A student who understands objects well can learn those too.

And AP CSA? It's an option, not the point of this chapter. Some families will want it; many won't. This chapter teaches Java because Java is worth learning. If your student later decides to sit the AP exam, the "Tools" section and Chapter 9 explain how that works for homeschoolers.

For the parent: understand it yourself

You don't need to write Java to teach this chapter. You need to tell a class from an object, recognize the three or four parts of a class, and know what a compiler error looks like so you can say "Read me the first one" instead of "Oh no."

The everyday picture. A recipe card for muffins is a class. It says what every batch has (flour, sugar, a number of muffins) and what you can do with it (bake, count, describe). The actual tray of muffins on your counter is an object. You can make three trays from one card, and each tray has its own count: one has twelve, one has eleven because somebody tested one. Changing the count on one tray doesn't change the card or the other trays.

The precise picture. The national high-school standards from the Computer Science Teachers Association put it this way: "Create a modular program that uses procedures, external libraries, or objects to improve reusability and readability."²¹ Notice the purpose: reusability and readability. Objects are a way to write code other people (including your student, next month) can read and

reuse.

A class in Python

Here is a small Book class:

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def is_long(self):
        return self.pages > 400

    def describe(self):
        return f"{self.title}, {self.pages} pages"

b = Book("The River Map", 312)
print(b.describe())
print(b.is_long())
```

It prints The River Map, 312 pages, then False. Read it in parts:

- **class Book:** starts the blueprint.
- **__init__** (said “init,” with two underscores on each side) is the setup step. It runs once, when a new book is made, and stores the title and page count. These stored facts are called **attributes**.
- **self** means “this particular book.” When the code says `self.pages`, it means *this* book’s pages, not some other book’s.
- **is_long** and **describe** are **methods**: things a book can do. A method is a function that belongs to a class.
- **b = Book("The River Map", 312)** makes one object from the blueprint. You can make as many as you like.

The same class in Java

Now the same idea in Java. Java likes each class in its own file, so Book goes in `Book.java`:

```
public class Book {
    private String title;
    private int pages;

    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }

    public boolean isLong() { return pages > 400; }
```

```

    public String describe() { return title + ", " + pages + " pages"; }
}

```

And a second file, `Main.java`, holds the starting point:

```

public class Main {
    public static void main(String[] args) {
        Book b = new Book("The River Map", 312);
        System.out.println(b.describe());
        System.out.println(b.isLong());
    }
}

```

In a terminal, `javac Book.java Main.java` compiles both files, and `java Main` runs the program. It prints `The River Map, 312 pages, then false` (Java writes `true` and `false` in lowercase). Same program. Here's what Java makes you say out loud that Python didn't:

- **Types.** `String title` and `int pages` say exactly what kind of value each attribute holds. `boolean isLong()` says the method hands back a `true-or-false`.
- **private** means only the `Book` class itself can change the title and pages. Other code has to ask a method. This is the “who gets to change it?” in the talk box.
- **The constructor**, `public Book(...)`, plays the part of Python's `__init__`. It has the same name as the class.
- **new** is required to make an object: `new Book(...)`.
- **Braces and semicolons.** `{` and `}` mark where each block starts and stops; Python uses indentation instead. Every statement ends with `;`.
- **main.** A Java program starts at a method called `main`. Its first line looks like a spell (`public static void main(String[] args)`); for now, it's just “the door the program comes in by.”

Compile, then run

The biggest difference isn't the punctuation. It's *when* mistakes show up. Here's a short Java program with two deliberate bugs, labeled:

```

public class Main {
    public static void main(String[] args) {
        int pages = "300";                // DELIBERATE BUG 1: text, not a number
        System.out.println(pages)         // DELIBERATE BUG 2: missing semicolon
    }
}

```

When we compiled it, Java reported only one error, the missing semicolon: `Main.java:4: error: ';' expected`. Nothing ran. After the semicolon was added and the file compiled again, Java reported the next one: `Main.java:3: error: incompatible types: String cannot be converted to int`. Fix that (`int pages = 300;`) and it compiles and runs.

That's a lesson worth a whole Hour. Java checks before it runs, and it often reports problems one

layer at a time: fix the first error, compile again, read the next. A wall of red after compiling isn't a hundred problems. Usually it's one, plus its echoes. The habit is: **first error first**, then compile again. Python, by contrast, would have run happily until it reached the bad line.

Wrong answers you should be able to hear

- **“A class and an object are the same thing.”** One example has blurred them. Ask: “If Book is the card, how many books can you make from it? Make me three.”
- **“It says takes 0 positional arguments but 1 was given.”** In Python, the method forgot self as its first parameter. When we ran a version of `is_long` written as `def is_long():`, Python said exactly that: `TypeError: Book.is_long() takes 0 positional arguments but 1 was given`. Python hands every method the object itself, and the method has to have a place to put it.
- **“Java says cannot find symbol.”** Often new is missing. `Book b = Book("The River Map", 312);` gives error: `cannot find symbol`, because without new, Java goes looking for a method named `Book` and doesn't find one. (The same error also appears for typos in names, so read the line it points at.)
- **“Java is just harder Python.”** Types and structure feel like noise. Ask: “What does the type tell you that Python didn't?”
- **“Seven divided by two is three?”** In Java, dividing two whole numbers gives a whole number. `7 / 2` prints 3; `7 / 2.0` prints 3.5. Python's `7 / 2` gives 3.5, and Python has a separate `//` for whole-number division (`7 // 2` gives 3). Neither is wrong. They're different promises.
- **“The two strings are equal but Java says false.”** In Java, `==` on text checks whether two variables point at the very same object, not whether the letters match. Use `.equals`. When we ran it, comparing “yes” with `new String("yes")` gave false with `==` and true with `.equals`. In Python, `==` compares the letters. So there are three ideas to keep apart: `=` stores a value, `==` compares, and in Java, `.equals` compares text.

Your five-minute warm-up

Take three index cards. Write “Book” on the first and list two attributes (title, pages) and two methods (is long?, describe). That's the class. On the other two cards, write two actual books from your shelf, with their own titles and page counts. Those are objects. Put the class card above the two object cards. Then say this chapter's opening question once, out loud: “What does this object know, what can it do — and who gets to change it?” That's the whole preparation.

How to teach it this week

Run the week inside the six steps of **The Computing Hour**, and expect it to take two weeks rather than one. Classes are new, and Java is a second language. Both deserve time.

- **Specify.** The spec now names the *things* in the program before it names the steps: “A Book knows its title and pages and can say whether it's long.” Each class gets one line of respon-

sibility.

- **Predict.** “What will `describe()` print for this book?” At 17–18 in Java: “Will this compile? If not, which line will Java complain about first?”
- **Build and Test.** One class at a time. Make one object, call one method, check the result. Tests from Chapter 3 continue: an `assert` on `describe()` in Python, a printed check in Java.
- **Debug and Log.** Java compile errors go in the Debug Log like any other: Expected/Actual, the first error’s line and message, one change, compile again.
- **Review.** Look at the class with fresh eyes: is anything in it that doesn’t belong to a book? Is anything outside it that does?
- **Ship and Explain.** “I specified... I predicted... It did... Next I’d change...” and “Help I used: ...”

Day 1: cards before code. The whole first Hour can be unplugged.

In-home — OOP-01 Index-Card Objects

- **Time:** 15 minutes
- **Materials:** index cards, two colors of pen
- **Safety and privacy:** use made-up names or objects from around the house
- **The fun:** you become a method
- **The skill:** cards as objects, with attributes on the front and methods on the back
- **The crumb:** a card set

How it goes. The student picks something the family owns many of: books, plants, bikes, board games. On one card they write the class: its name, the attributes every one of them has, and two or three things it can do. Then they make three object cards from it, each with its own values on the front and the same methods on the back. To “call a method,” the student hands you an object card and says “describe.” You do exactly what the back of the card says, using only the facts on the front. If the back says “say the title and pages,” you say exactly that, nothing more. Every time you have to guess, there’s a gap in the class.

15–16: you play the method; the student designs the cards. **17–18:** add a second class that uses the first (a `Shelf` that holds several `Book` cards and can count their pages) and play it out with real cards on the table.

Day 2: the first class in Python.

In-home — OOP-02 Your First Class

- **Time:** 30 minutes
- **Materials:** the editor, yesterday’s cards
- **Safety and privacy:** no real people’s names in the data
- **The fun:** the card set comes to life
- **The skill:** a Python class with `__init__`, attributes, and two methods
- **The crumb:** the program file

How it goes. The student types the class from their card set, not from the book. Start with `__init__` and one method. Make one object. Print one thing. Then add the second method and a second object. Each run is preceded by a written prediction. Add two `assert` lines from Chapter 3 at the bottom, one for each method. If they're using the Book example, a good test is `assert b.describe() == "The River Map, 312 pages"`.

15–16: sit nearby and be first user: make an object with a page count of 0, or of 401, and report what you saw. **17–18:** the student adds a method that changes an attribute (a `mark_returned()` that sets `self.returned = True`) and writes a test that proves it changed only that one object.

Days 3–4: the same program in Java. This is the bridge.

In-home — OOP-03 Same Program, in Java

- **Time:** 40 minutes (often split over two Hours)
- **Materials:** a free Java Development Kit (JDK) and a plain editor, or a browser-based Java editor if the family's rules allow it; Tuesday's Python file
- **Safety and privacy:** install only from the official source for the JDK; a parent decides on any browser tool and its account
- **The fun:** watching the same idea wear different clothes
- **The skill:** rewriting a small Python class in Java: types, braces, main, compile then run
- **The crumb:** two files side by side

How it goes. The Python file stays open on one side. The student writes the Java version beside it, line by line matching line, saying the translation out loud: “`__init__` becomes the constructor. `self.pages` becomes `this.pages`. And I have to say `pages` is an `int`.” Then compile. Expect errors, and plan for them. The first compile almost always fails, and that's the lesson. Read the first error only. Fix it. Compile again. Log each one. Picture a fifteen-year-old whose log entry after the first Java Hour reads: “Java tells me before it runs. Python told me when it ran.” That sentence is most of this chapter.

15–16: the student copies the shape of `Book.java` and `Main.java` from this chapter, then changes them to match their own card set. **17–18:** the student writes the Java version without looking at the book's, using the Python file and Java's own error messages as the only guides.

Day 5: the diagram.

In-home — OOP-04 Class Diagram on Paper

- **Time:** 15 minutes
- **Materials:** paper, pencil, a ruler if you like straight lines
- **Safety and privacy:** none needed
- **The fun:** seeing the whole program on one page
- **The skill:** boxes and arrows for two classes that work together
- **The crumb:** the diagram

How it goes. Each class is a box with three parts: its name at the top, its attributes in the middle (with types, at 17–18), its methods at the bottom. An arrow from `Shelf` to `Book` says “a shelf holds

books.” The student draws the diagram for the program they’re building, then answers out loud: “Who knows the page count? Who knows the total? If the total is wrong, which box do I open first?”

15–16: two classes. **17–18:** three classes, and a sentence under each box saying what it’s responsible for, and what it’s *not*.

Fading help. Week one, you may need to say “first error first” every time Java complains. Week two, point at the first line of the error. By week three, the student reads it to you. When they ask for rescue, use the rescue order from **The Computing Hour**: ask → wait → hint → short demo on your own copy → hand it back. Good hints in this chapter: “Which card would know that?” and “What type is it?”

Practice that actually builds learning

Practice starts blocked (one class, several objects; then the same class in both languages) and becomes mixed (deciding *which* class a new piece of information belongs to).

In-home — PY-06 Paper Parsons, Python Edition

- **Time:** 10 minutes
- **Materials:** a working Java method cut into strips, plus one distractor strip
- **Safety and privacy:** none needed
- **The fun:** a puzzle with a trap in it
- **The skill:** reordering cut-up lines into a working function
- **The crumb:** the taped strip

How it goes. You met Parsons puzzles with Python in Chapter 2. This week, print the Book constructor and its describe method in Java, cut it into one line per strip, shuffle, and add one distractor (a Python line like `self.pages = pages`, or a Java line with a missing semicolon). The student orders the strips, tapes them down, and says why the distractor doesn’t belong. Matching braces is half the puzzle, which is the point.

In-home — OOP-05 Arrays and ArrayLists

- **Time:** 30 minutes
- **Materials:** the editor and JDK
- **Safety and privacy:** none needed
- **The fun:** a game board in nine lines
- **The skill:** Java arrays and ArrayLists, loops over them, a 2D grid
- **The crumb:** a Java file plus a trace on paper

How it goes. A Java **array** holds a fixed number of values of one type. Its size is set when it’s made. An **ArrayList** is a list that can grow, like a Python list. Start with the array:

```
public class Arr {
```

```

    public static void main(String[] args) {
        int[] scores = {88, 92, 75};
        System.out.println(scores.length);
        System.out.println(scores[3]);    // DELIBERATE BUG: there is no index 3
    }
}

```

It prints 3, then crashes with `ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3`. Three items are numbered 0, 1, and 2, just as in Python. (This is a runtime error: Java compiled it happily, because the compiler checks types, not which numbers you'll ask for.)

Then an `ArrayList` that grows:

```

import java.util.ArrayList;

public class Shelf {
    public static void main(String[] args) {
        ArrayList<Integer> pages = new ArrayList<>();
        pages.add(312);
        pages.add(95);
        pages.add(540);
        int total = 0;
        for (int p : pages) {
            total += p;
        }
        System.out.println(pages.size() + " books, " + total + " pages");
    }
}

```

It prints `3 books, 947 pages`. Notice `Integer`, not `int`, inside the angle brackets; `ArrayLists` hold objects, and `Integer` is the object version of a whole number. That's a fact to know, not to understand fully this week.

Finally, a **2D array**, a grid of rows and columns, which is how many board games are stored:

```

public class Grid {
    public static void main(String[] args) {
        char[][] board = new char[3][3];
        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                board[r][c] = '.';
            }
        }
        board[1][2] = 'X';
        for (char[] row : board) {
            System.out.println(new String(row));
        }
    }
}

```

It prints three rows: `...`, then `..X`, then `...`. Before running it, the student traces it on paper (a 3-

by-3 grid of boxes) and predicts where the X lands. Row 1, column 2 is the middle row, right-hand column, because both count from zero.

17–18: all three. **Ready 16:** a 16-year-old who’s comfortable with Python lists and nested loops from Chapter 2 can take this on now.

Out-of-home — OUT-03 Ask a Working Programmer

- **Time:** 30 minutes
- **Materials:** five questions written in advance; a notebook
- **Safety and privacy:** an adult the family knows (a relative, a co-op parent, a neighbor); meet in a public place or by video with a parent present; first names only in notes
- **The fun:** finding out what the job is actually like
- **The skill:** a short interview with an adult who writes code at work
- **The crumb:** interview notes (first name only)

How it goes. The full version of this try-it lives in Chapter 6, and many families will do it there. If your student is curious now, do a short version this week with one question about objects: “How does your team decide what goes in a class?” or “What’s a class in your code that got too big, and what did you do?” Most working programmers have a story. Write down the answer in the student’s words, and add one follow-up question they’d ask next time.

Talk box (this chapter)

Opening: “What does this object know, what can it do — and who gets to change it?”

Follow-ups: 1. **Prediction:** “If I make a second book and change its pages, what happens to the first one?” 2. **Evidence:** “Which line in the class makes that true?” 3. **Location:** “Where does the page count live? Point to it.” 4. **Change:** “If we added a due date, which class would get it, and what would change in the Java version?” 5. **Revision:** “Is there anything in this class that doesn’t belong to a book?”

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. Often that the student is translating between languages in their head and lost track of which one you asked about. Point at the file. Sometimes it means the class and object are still blurred; go back to the cards on the table. And sometimes the student knows the answer and is checking whether you’ll accept a Python answer to a Java question. Ask for both.

Don’t say / say instead. Don’t say “Java is just picky”; say “Java is telling you before it runs. What’s it telling you?”

An incorrect example to diagnose

Show the student this Python class, labeled as a poor design:

```
class Library:
    def __init__(self):
        self.titles = []
        self.pages = []
        self.due_dates = []
        self.borrowers = []
```

It runs. Nothing is wrong, exactly. Ask: “If someone adds a book’s title but forgets its page count, what happens to every book after it?” The lists drift out of line, which is the bug from this chapter’s opening. Then ask: “What would a Book class fix?” Each book keeps its own facts together, so they can’t drift. Finding the *shape* problem, not just the line problem, is what this chapter is for.

Mixed practice

Once the cards and the first classes are done, mix it up. Hand the student a list of facts and actions (“a due date,” “renew,” “count all overdue books,” “the borrower’s first name”) and have them decide which class each belongs to, on paper, before any code. Mix in Java and Python: one Hour in each, same project. Keep the monthly AS-01 Cold Trace, using a short class this time: “What will this print?”

For the student

So far your programs have been lists of steps with some data floating around them. This chapter changes the shape. From now on, you’ll build programs out of **objects**: things that hold their own facts and know how to do their own jobs.

The idea. A class is a blueprint. An object is one thing built from it. Book is the class: every book has a title and a page count and can describe itself. “The River Map, 312 pages” is one object. Make a hundred objects from one class, and each keeps its own facts.

Here’s the Python version:

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def is_long(self):
        return self.pages > 400
```

`__init__` runs when a new book is made. `self` means *this* book. `is_long` is a method, a function that belongs to the class. `Book("The River Map", 312).is_long()` gives False.

Now Java. Java says the same thing with more words, and every extra word means something:

```
public class Book {
    private String title;
    private int pages;
```

```

    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }

    public boolean isLong() { return pages > 400; }
}

```

String and int are types. private means only Book can change these. The constructor has the class's name. Braces show where blocks start and stop. Semicolons end statements. To make a book, you write new Book("The River Map", 312).

Compile, then run. Java checks the whole program before it runs a single line. That's annoying for about a week, and then it's useful: Java catches a whole family of mistakes that Python only finds when it reaches them. When Java reports errors, read the **first** one, fix it, and compile again. The others are often echoes, and sometimes the next error is hiding behind the first.

Try this today. Pick something you own several of (books, games, plants, tools). On an index card, write its class: two attributes and two methods. Write it in Python with two objects and a test for each method. Then write it in Java. Keep both files open side by side. Your first Java compile will probably fail. Good. Log every error: line, message, fix.

Explain it back. Point at your Java file and explain, to a parent or a sibling, what each of these does: private, the constructor, new, and the type in front of each method. Then explain what Python did instead for each one. If you can do that, you understand both languages better than if you'd only learned one.

This part is hard for everyone. Types and semicolons. You'll forget a semicolon. You'll put text where a number belongs. You'll write == when you meant .equals on two strings and get false for two words that look identical. Everyone who learns Java does all of these, usually in the first week. The fix is the same as in Chapter 3: read the message, change one thing, log it. One more trap: in Java, 7 / 2 is 3, because two whole numbers make a whole number. Use 7 / 2.0 when you want 3.5.

Who gets to change it? This is the question that makes classes worth it. When pages is private, the only code that can change it is inside Book. If the page count ever goes wrong, you know exactly where to look. That's a big deal in a program with a thousand lines, and it's how professional teams keep big programs from turning into tangles.

If you're 17 or older: learn arrays, ArrayLists, and 2D arrays this month (OOP-05). Then build a small grid game: a 3-by-3 or 5-by-5 board, a class for the board, a method to place a piece, and a method that checks whether a move is allowed. Write tests for the edges: what happens at row 0? At the last column? Outside the board? If you're thinking about the AP Computer Science A exam, know that it's in Java and that its written section asks you to write methods, design a class, work with an ArrayList, and work with a 2D array. Your grid game touches all four. AP is a choice, not a requirement; talk with your parent about whether it fits your plans.

A worked translation. Here's how one line moves between the languages, said the way you'd say it to a teammate. Python: self.pages = pages. Java: this.pages = pages;. Same idea: store

the value you were handed in this object's own attribute. Python: `def is_long(self):`. Java: `public boolean isLong() {`. Java adds who may call it (public), what it hands back (boolean), and braces where Python used a colon and indentation. Python names use underscores (`is_long`); Java's custom is capitals in the middle (`isLong`). Neither is right; each language has its habits, and following them is part of writing code other people can read.

Reading a Java error. Java's messages have a shape: the file, a colon, the line number, the word error, and a short description, then the line itself with a caret under the spot. `Main.java:4: error: ';' expected` means “file Main.java, line 4, I expected a semicolon.” Copy that first line into your Debug Log before you change anything. After a week you'll recognize the five or six messages you see most, and they'll stop feeling like insults.

Challenge. Build a 2D grid game board as a class, in Java, with a `place(row, col)` method that refuses moves outside the board instead of crashing. Write three tests, one of them for a move that should be refused. Then write the same class in Python and compare. Which language caught more of your mistakes before you ran it?

What to save: your index cards (a photo is fine), both program files, your class diagram, and the Debug Log pages with every compile error you met. Save them in two places.

If it isn't clicking

Class and object are still confused. One example blurred them. Go back to the cards: one class card, three object cards, each with different values. Ask: “If I change this one's pages, what happens to the others?” Then make three objects in code and print each.

Java errors feel hostile. The compiler's messages aren't being read; they're being looked at. Read them top-down, first error first, out loud. Copy the first error's line number and message into the Debug Log before touching anything. Within a week, most students find Java's messages more helpful than Python's, because they arrive before anything runs.

Everything ends up in one giant class. There's no sense yet of who's responsible for what. Back to OOP-04: draw the boxes and ask, for each fact, “Who would know this?” A `Shelf` knows how many books it holds. A `Book` knows its own pages. Nobody else should.

Java itself is the obstacle. Some students hit a wall with the second language. That's fine. Slow down, stay in Python for classes for another two weeks, and come back to Java with one tiny class. If Java still doesn't stick, a co-op class, a tutor who knows Java, or a dual-enrollment course (Chapter 9) can carry the strand. When you ask for rescue, the order is the same as ever: ask → wait → hint → short demo on your own copy → hand it back.

Tools, including AI

For Java, a free Java Development Kit (the JDK) plus a plain editor is enough: `javac` compiles, `java` runs. Install the JDK only from its official source. A browser-based Java editor is an option

if the family’s rules allow it; a parent decides on the account and checks the current terms. Any free code editor that handles Python will handle Java too. No brand matters here.²²

The AI rules are the ones in the AI box in **The Computing Hour**. At 15–16, chat-style help may explain a Java error message after the student has written their own Expected/Actual line and guess. The editor coding assistant stays off through 16, and for every check. Translating Python into Java is exactly the kind of work an assistant does instantly and a student learns from by doing slowly, so keep this chapter’s translation work the student’s own. At 17, on one named project after AS-04 Build From a Blank, every suggested line still gets explained before it’s kept.

AP Computer Science A, if your family is considering it. The course and exam are built around Java. The current framework took effect in fall 2025. The exam has a multiple-choice section and four free-response questions: one on methods and control structures, one on class design, one on data analysis with an ArrayList, and one on a 2D array.²³ Homeschooled students take AP exams at an authorized school, arranged through that school’s AP coordinator. Ask a coordinator by early fall; the final ordering deadline falls in mid-November, and exams are given in the first two weeks of May.²⁴ Putting “AP” in a course title on your transcript is a separate step, the AP Course Audit (Chapter 9). This chapter covers the Java a student needs for much of the course, but it isn’t an AP prep book, and the exam is optional.

What “done enough” looks like

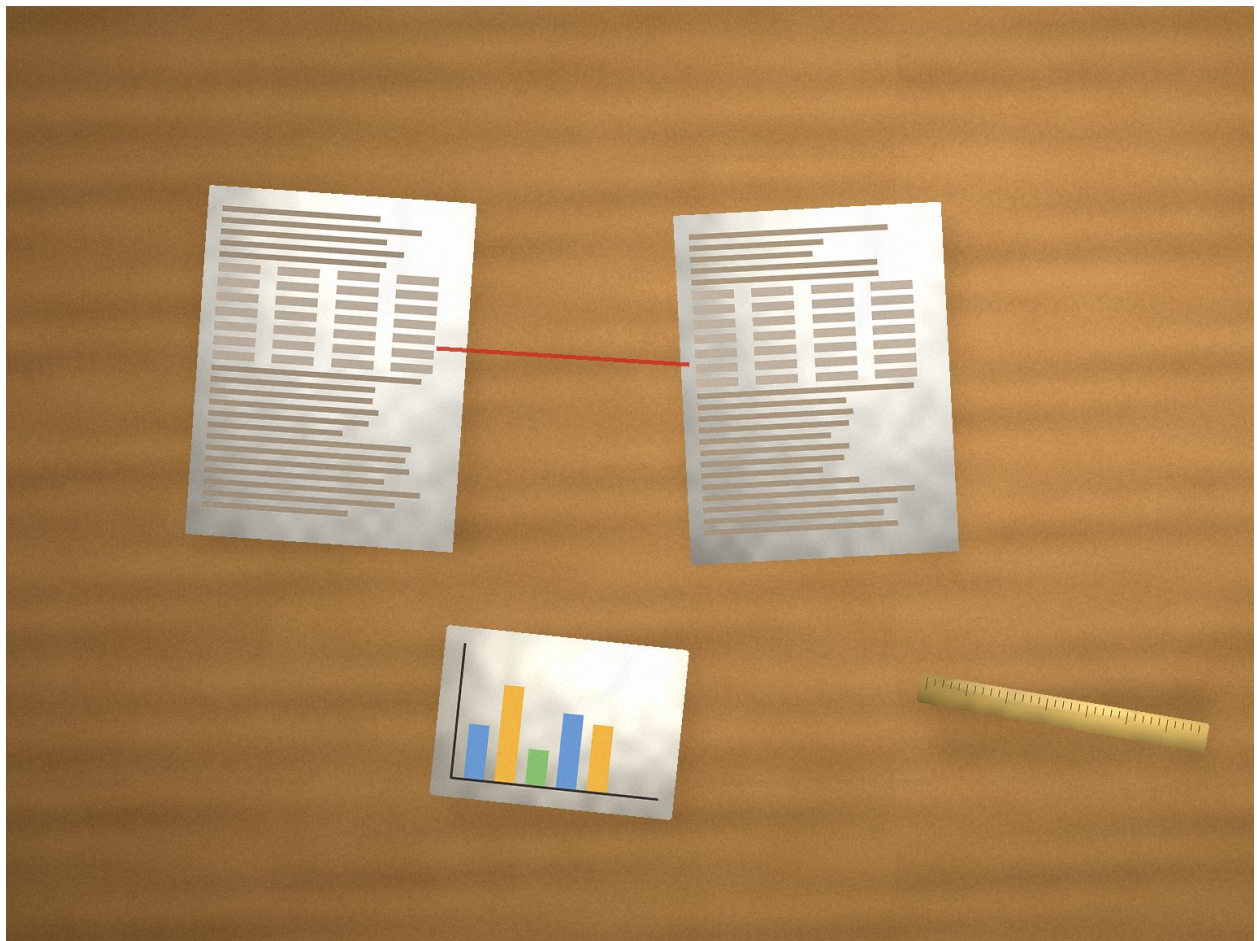
Move on when most of these are “seen,” a few are “growing,” and none of the first three is “not yet”:

- ☐ Explains the difference between a class and an object, using a real example (OOP-01)
- ☐ Writes a Python class with `__init__`, two attributes, and two methods, with tests (OOP-02)
- ☐ Writes the same class in Java, compiles it, and reads compiler errors first-error-first (OOP-03)
- ☐ Draws a class diagram for two classes that work together, saying what each is responsible for (OOP-04)
- ☐ Explains `=` versus `==` versus `.equals`, and why `7 / 2` is 3 in Java
- ☐ **AS-03 Explain Your Code:** teaches back their own Java class, line by line, to you. You write one line about how it went.
- ☐ **AS-01 Cold Trace:** traces a short class and its method calls on paper and predicts the output, unaided.
- ☐ **AS-02 Debug on the Spot:** fixes a Java compile error you planted (a missing semicolon or a wrong type) while explaining aloud.
- ☐ **17–18 (ready 16):** uses an array, an ArrayList, and a 2D array in Java, and traces the 2D grid on paper (OOP-05).

Placement. Place by skill, not birthday. A 15-year-old who writes classes in both languages and can explain `private` is ready for Chapter 5. A 17-year-old who still mixes up class and object needs another week with the cards, and that’s normal. Java is a second language; give it a second month if it needs one.

Chapter 5

Data: Questions, Tables, and Databases



Why this matters

Here is an illustration, not a real family. Picture a seventeen-year-old who wants to know which month the family uses the most water. She could open a spreadsheet, paste in the meter readings, click the chart button, and get a colorful picture in thirty seconds. It would look convincing. It would also be wrong, because one month's reading is missing, another was typed with a space in

it, and the chart's axis starts at 400, which makes a small difference look enormous.

She does it differently. She writes the question on a card first. She writes down what each column means and what values are allowed before she looks at a single number. She cleans the file in code and keeps a log of everything she changed. She makes one chart, with an honest axis, and writes a caption that says what the data *can't* show. It takes an afternoon instead of thirty seconds. And when her parent asks "Are you sure?", she can show exactly why.

That's what this chapter teaches. Data work starts with a question, not a spreadsheet. The student writes the question, describes the data before collecting it, cleans it in code, answers with one honest chart, and learns why serious data lives in a **database**: organized tables, linked by **keys**, the columns that connect one table to another. A **query** is a question asked of those tables, in a language called SQL (often said "sequel"). Databases are behind nearly every app your student uses, from the library catalog to a shopping cart, and SQL has been the common way to ask them questions for decades.

It also matters beyond computing. Teens meet charts constantly: in news stories, in ads, in posts that want them angry or afraid. A student who has made an honest chart, and cut an axis once on purpose to see what it does, reads every other chart differently. That's a real gain. It's a habit of reading, though, not a promise about anything else.

One caveat, said once and plainly: two things rising together doesn't mean one causes the other. Ice-cream sales and sunburns rise together, and ice cream doesn't cause sunburn; summer causes both. The chapter comes back to this habit in the talk box. After that, it just asks the question: "What else could explain it?"

For the parent: understand it yourself

You don't need statistics to run this chapter. You need to know what a clean table looks like, why one big table is a trap, and how to ask "Could this data actually answer that?" You can learn all three this week.

The everyday picture. A recipe box with one card per recipe is a table. Each card has the same fields in the same places: name, servings, time, ingredients. If one card says "serves 4" and another says "four people" and a third leaves it blank, you can't sort the box by servings. Cleaning data is making every card say it the same way. A database is several boxes that refer to each other: a recipe card says "see pantry card 12" instead of copying the pantry details onto every recipe.

The precise picture. The national high-school standards from the Computer Science Teachers Association describe two of this chapter's skills exactly. One asks students to "Create a data dictionary that describes the name, type, and allowable values for each attribute and the logical relationships between variables in a dataset."²⁵ Another asks them to "Create a data visualization of a multivariate dataset to answer a question or make a classification or prediction."²⁶ Notice that both standards put the description and the question first. The data dictionary comes before the data. The question comes before the chart.

Four ideas in plain words

- **A row** is one thing (one reading, one book, one loan). **A column** is one fact about every row (the date, the title). Every row has the same columns.
- **A data dictionary** is a short table about the table: for each column, its name, its **type** (date, whole number, text), and the values allowed. “Reading: whole number, never smaller than last month’s.”
- **Cleaning** is fixing the rows that break the dictionary: stray spaces, “n/a,” a number typed into the wrong column. Good cleaning is logged, so anyone can see what changed.
- **A key** is the column that links tables. In a library database, every person has a `person_id`, every book a `book_id`, and each loan row just says which person and which book. Nobody’s name is typed twice.

Cleaning in code

Here’s a small, made-up meter file, with three problems planted on purpose: a reading with spaces around it, a reading of “n/a,” and a blank line.

```
date,reading
2026-05-01,10230
2026-06-01,10810
2026-07-01, 11690
2026-08-01,n/a
2026-09-01,12460

2026-10-01,12950
```

And here’s a short Python program that cleans it and keeps a log:

```
import csv

clean, log = [], []
with open("readings.csv") as f:
    for row in csv.DictReader(f):
        date = (row["date"] or "").strip()
        text = (row["reading"] or "").strip()
        if not text.isdigit():
            log.append(f"dropped {date}: reading was {text!r}")
            continue
        clean.append((date, int(text)))

print(len(clean), "good rows")
for line in log:
    print(line)
```

When we ran it, it printed 5 good rows and dropped 2026-08-01: reading was 'n/a'. Three things happened, and one is a surprise worth teaching:

1. `.strip()` removed the spaces around 11690, so July was kept.

2. The “n/a” row failed the check and was logged, not silently thrown away.
3. The blank line never reached the loop at all. Python’s built-in csv reader skips completely empty lines on its own, so it doesn’t appear in the log.

That third one is the lesson: know what your tools do quietly. It’s fine for a blank line to vanish. It wouldn’t be fine for a real reading to vanish without a trace, which is why the log exists.

The missing month

Now the question. Usage is the change between readings: May to June used 580 units, June to July 880. But August is missing, so the next change, July to September, is 770 units for *two* months. A careless chart would show “Jul–Sep: 770” next to “Jun: 880” as if they were the same kind of bar, or worse, would fill the missing August with zero and show a one-month drop. A missing row is not a zero. The honest chart labels that bar as two months, and the caption says so.

Why two tables beat one

Picture a library-loans spreadsheet with one row per loan: borrower’s name, book title, pages, date. Every time Ana borrows a book, her name is typed again. Every time *The River Map* goes out, its title and page count are typed again. If someone misspells “Ana” once, she becomes two people. If the page count is corrected on one row and not the others, the data disagrees with itself.

The fix is three tables: **people** (an id and a first name), **books** (an id, a title, pages), and **loans** (an id, which person, which book, and the date). Each fact is written once. The loans table holds only keys and a date. SQL’s JOIN puts them back together when you ask a question.

What SQL looks like

SQL reads almost like English. Using a small made-up library database, this query asks for the longer books, longest first:

```
SELECT title, pages FROM books
WHERE pages > 200
ORDER BY pages DESC;
```

It returned three rows: *Night Garden* (540), *The River Map* (312), *Paper Boats* (210). SELECT names the columns you want, FROM names the table, WHERE filters the rows, and ORDER BY sorts them.

SQLite is a small database that lives in a single file on your own computer, with no server and no account. Python comes with a module for it, `sqlite3`, built in.²⁷ That’s enough for everything in this chapter.

Wrong answers you should be able to hear

- “**The chart proves it.**” Correlation, a cut axis, or missing data is doing the work. Say: “What would change your mind?” or “What else could explain it?”

- **“I averaged the categories.”** Averaging “favorite colors” or zip codes produces a number that means nothing. Say: “What would an average of those actually tell someone?”
- **“The missing ones are zero.”** A blank isn’t a zero. Say: “Did that month use no water, or did nobody write it down?”
- **“I’ll just keep it in one big table.”** Repeated names will drift. Say: “If Ana changes how she spells her name, how many rows do you have to fix?”
- **“I made a chart; now what’s the question?”** Tool first, question second. Say: “Put the question card on the table first.”

Your five-minute warm-up

Take any receipt or bill in the house. On a sheet of paper, draw it as a table: what are the columns? What type is each one? What values are allowed? Is anything repeated that could live in its own table? Then say this chapter’s opening question once, out loud: “What question are you asking — and could this data actually answer it?” That’s your preparation.

How to teach it this week

This chapter is two weeks for most families: one for questions, dictionaries, cleaning, and charts; one for databases and SQL. Run each Hour inside the six steps of **The Computing Hour**.

- **Specify.** The spec starts with the question card and the data dictionary. “Done” means the question is answered, or the student can say clearly why this data can’t answer it.
- **Predict.** “How many rows will survive cleaning?” “What will this query return?” Written down.
- **Build and Test.** Small steps: read the file, print the first three rows, then clean, then count. For SQL, one clause at a time: `SELECT`, then add `WHERE`, then `ORDER BY`.
- **Debug and Log.** Every cleaning decision goes in a change log; every surprise goes in the Debug Log with Expected/Actual.
- **Review.** Re-read the question card. Does the chart answer *that* question?
- **Ship and Explain.** “I specified... I predicted... It did... Next I’d change...” and “Help I used: ...”

A word on whose data. The student uses the family’s own non-personal data (meter readings, a reading log, weather they recorded, the household’s book collection) or public datasets from a library or government site. Data about siblings, friends, or strangers needs their permission first, and some of it shouldn’t be collected at all. That’s part of Family safety basics in **The Computing Hour**, and it’s a good conversation to have before the first file is opened.

Monday: the question.

In-home — DATA-01 Question First

- **Time:** 15 minutes
- **Materials:** an index card, a pencil

- **Safety and privacy:** pick a question about things, not people
- **The fun:** asking something nobody in the house knows the answer to yet
- **The skill:** writing a data question and the columns that would answer it
- **The crumb:** a question card

How it goes. On the front of the card, the student writes one question that has a checkable answer: “Which month do we use the most water?” “Do we finish shorter books faster?” “Is the backyard thermometer warmer than the forecast?” On the back, the columns that would answer it, and where the data would come from. Then they ask themselves: could this data actually answer the question, or only part of it? A question like “Are we a reading family?” can’t be answered by a page count. “Did we read more pages in October than September?” can.

15–16: you ask one question about the question: “How will you know when you’ve answered it?”

17–18: the student writes two questions, one they think the data can answer and one it can’t, and explains the difference.

Tuesday: the dictionary.

In-home — DATA-02 The Data Dictionary

- **Time:** 15 minutes
- **Materials:** paper, pencil, the question card
- **Safety and privacy:** none needed
- **The fun:** making rules before the mess arrives
- **The skill:** name, type, and allowed values for each column, before collecting
- **The crumb:** a data dictionary

How it goes. One row per column: name, type, allowed values, and an example. For the water question: date (text in YYYY-MM-DD form, first of each month); reading (whole number, never smaller than last month’s). Writing “never smaller than last month’s” is the kind of rule that catches a typo later. At 17–18, add a line about how columns relate, the “logical relationships” in the standard: “usage is this reading minus last month’s.”

Wednesday: cleaning.

In-home — DATA-03 Clean It in Code

- **Time:** 30 minutes
- **Materials:** the editor, a small CSV file the student made or downloaded
- **Safety and privacy:** never clean the only copy; work on a copy and keep the original
- **The fun:** watching the program catch the messes you planted
- **The skill:** reading a CSV, fixing spacing and case, dropping bad rows, logging what changed
- **The crumb:** a clean file plus a change log

How it goes. Start with the cleaning program from the parent section, or the student’s own version. Before running, predict how many rows will survive. Then run it, compare, and read the log. Plant one new mess (a word in place of a number, a reading in the wrong column) and predict again. At

the end, the program writes the clean rows to a new file, the way PY-04 Write a Report File taught in Chapter 2, so the original is never touched.

15–16: a file of 10–20 rows. **17–18:** a public dataset of a few hundred rows, and a change log that says how many rows were dropped for each reason.

Thursday: the chart.

In-home — DATA-04 Chart That Answers

- **Time:** 25 minutes
- **Materials:** the clean data; graph paper, a spreadsheet, or a charting library
- **Safety and privacy:** no names or faces in a chart that might be shared
- **The fun:** cutting the axis on purpose to see what it does
- **The skill:** one chart that answers the question, with an honest axis and a caption
- **The crumb:** a chart plus caption

How it goes. One chart. Its title is the answer (“June used the most water”). The axis starts at zero for bars. The caption says what the data can’t show. Then, as an experiment, the student makes a second version with the axis starting at 400 and compares the two side by side. Same numbers; very different story. Neither is a lie on its face, but only one lets a reader judge the size of the difference.

Charts can be drawn by hand on graph paper, made in a spreadsheet, or made in Python with a free charting library that’s installed separately. Here’s a short example using one widely used library, matplotlib:

```
import matplotlib.pyplot as plt

months = ["May", "Jun", "Jul+Aug", "Sep"]
units = [580, 880, 770, 490]

plt.bar(months, units)
plt.ylim(0, 1000)           # the axis starts at zero
plt.ylabel("Units used (meter reading change)")
plt.title("June used the most water")
plt.figtext(0.01, 0.01, "Jul+Aug is two months: the August reading is missing.")
plt.savefig("water.png")
print("saved water.png")
```

When we ran it, it printed saved water.png and made the image file. Notice where the honesty lives: the axis line, the label “Jul+Aug,” and the caption.²⁸

15–16: hand-drawn or spreadsheet is fine. **17–18:** the chart includes at least two variables (usage and month, plus a mark for a vacation or a garden month), which is what the standard means by “multivariate.”

Friday: who’s missing.

In-home — DATA-05 Who Is Missing?

- **Time:** 15 minutes
- **Materials:** the dataset and the question card
- **Safety and privacy:** none needed
- **The fun:** finding the hole in your own data
- **The skill:** sampling and coverage, meaning who or what the dataset leaves out
- **The crumb:** a two-sentence note

How it goes. The student writes two sentences: what or who this data leaves out, and how that could change the answer. For the water meter: “August is missing. The months we were on vacation used less, so a summer average would be too low.” For a public dataset: “This survey only asked people with landlines.” Chapter 8 comes back to this card, because it’s exactly the question to ask of an AI system’s training data.

The second week: databases. Start on paper again.

Practice that actually builds learning

In-home — SQL-01 Tables on Paper

- **Time:** 15 minutes
- **Materials:** a messy one-big-table sheet (you can write it in five minutes), paper, colored pencils
- **Safety and privacy:** first names only, made up
- **The fun:** untangling a mess with arrows
- **The skill:** rows, columns, keys, and why two tables beat one big one
- **The crumb:** a paper schema

How it goes. Write a table of library loans with a borrower’s name, book title, pages, and date on every row, with Ana’s name appearing three times and one book appearing twice. The student splits it into three tables (people, books, loans), gives each person and book an id, and draws an arrow from each id in the loans table to the table it points at. Then you ask: “Ana wants to be called Anna. How many rows change now?” One.

In-home — SQL-02 First Queries

- **Time:** 30 minutes
- **Materials:** a small SQLite database file the student built from yesterday’s paper schema, and either Python’s `sqlite3` module or the `sqlite3` command-line tool
- **Safety and privacy:** made-up data
- **The fun:** asking a question and getting the answer in one line
- **The skill:** `SELECT`, `WHERE`, `ORDER BY` on a small local database
- **The crumb:** a query file with results

How it goes. The student writes each query in a file, predicts the result (how many rows, which ones), then runs it. Start with `SELECT * FROM books;` to see everything, then narrow. The query

from the parent section, long books longest first, is a good third query. Then the student writes their own: books under 100 pages; loans in October (`WHERE out_date >= '2026-10-01'`); people whose names start with A. Every result gets compared with the prediction. A query that returns zero rows when you expected three is a bug, not a quiet success.

15–16: five queries from a list you agree on. **17–18:** the student writes the questions too.

In-home — SQL-03 Join Two Tables

- **Time:** 30 minutes
- **Materials:** the same database
- **Safety and privacy:** made-up data
- **The fun:** putting the pieces back together
- **The skill:** JOIN on a key; GROUP BY with COUNT
- **The crumb:** a query file

How it goes. A join answers “who borrowed what?” by following the arrows from the paper schema:

```
SELECT people.first_name, COUNT(*) AS loans
FROM loans
JOIN people ON loans.person_id = people.person_id
GROUP BY people.first_name
ORDER BY loans DESC;
```

On the small library database, it returned Ana 3 and Ben 2. Then the good question: the people table also has Cal. Where is Cal? He has no loans, so the join had nothing to match him to. That’s not a bug in SQL. It’s a “who is missing?” moment, and a perfect DATA-05 echo. (A different kind of join can include him with a count of zero; the student can look it up in the SQLite documentation as a stretch.)

Then the classic mistake, on purpose: leave off the JOIN ... ON and list both tables, `SELECT COUNT(*) FROM loans, people;`. On our database it returned 15: every one of the five loans paired with every one of the three people. When a join returns far too many rows, the key condition is missing or wrong.

17–18: joins across all three tables. **Ready 16:** a 16-year-old who drew the paper schema confidently can start here.

In-home — SQL-04 Python Talks to SQLite

- **Time:** 35 minutes
- **Materials:** the editor and the database file
- **Safety and privacy:** this card teaches the safe way first; Chapter 7 shows why, on the student’s own toy app
- **The fun:** a little program anyone in the family can use
- **The skill:** using Python’s built-in `sqlite3` with parameterized queries
- **The crumb:** the program file

How it goes. A short program asks for a first name and lists that person's loans:

```
import sqlite3

conn = sqlite3.connect("lib.db")
name = input("First name: ")
rows = conn.execute(
    "SELECT books.title FROM loans "
    "JOIN people ON loans.person_id = people.person_id "
    "JOIN books ON loans.book_id = books.book_id "
    "WHERE people.first_name = ?",
    (name,),
).fetchall()
for (title,) in rows:
    print(title)
print(len(rows), "loans")
conn.close()
```

Typing Ana printed *The River Map*, *Night Garden*, *Paper Boats*, and 3 loans. The ? is the important part. It's a **parameter**: a placeholder that sqlite3 fills in safely with whatever the user typed, as data, never as part of the SQL itself. When we typed the classic trick input Ana' OR '1'='1, the program printed 0 loans, because no one's first name is literally that string. Building the query by gluing the user's text into the SQL string would have let that input change the question. Parameterized queries are the default from the first day, and Chapter 7 shows the difference on the student's own toy login form, under the family security rule.²⁷

17–18 only.

In-home — DS-03 Nested Data on Paper

- **Time:** 15 minutes
- **Materials:** paper, pencil
- **Safety and privacy:** none needed
- **The fun:** finding a value three levels down
- **The skill:** drawing a list of dictionaries (records) and tracing access paths
- **The crumb:** a diagram

How it goes. You met this card in Chapter 2. This week, draw the loans as a Python list of dictionaries, one dictionary per row, and trace `loans[1]["book_id"]` with a pencil. Then compare with the table. A database table and a list of dictionaries are two drawings of the same idea, and a row from sqlite3 is easy to turn into either.

Out-of-home — OUT-02 Library Computer Hour

- **Time:** 50 minutes
- **Materials:** a library card, a USB drive or the student's own cloud folder
- **Safety and privacy:** log out of everything; save to your own drive; don't leave files on the public computer

- **The fun:** a librarian who knows where the data lives
- **The skill:** a Computing Hour on a public computer, logging out and saving to your own drive
- **The crumb:** a file saved in two places

How it goes. You met this card in **The Computing Hour**. This week, borrow it for data. Many public libraries offer access to local and government datasets, and librarians are often delighted to help find one. The student asks one question at the desk (“Where would I find data about rainfall in our county?”), downloads one small public dataset, writes its data dictionary on the spot, and saves it in two places. At home, the Question First card decides whether it’s worth cleaning.

Talk box (this chapter)

Opening: “What question are you asking — and could this data actually answer it?”

Follow-ups: 1. **Prediction:** “Before you run it, how many rows do you think will come back?” 2. **Evidence:** “Which rows make you say that? Show me in the table.” 3. **Location:** “Where in the data would the answer live, and which column?” 4. **Change:** “If the axis started at zero, what would the chart look like?” 5. **Revision:** “What else could explain it, and what data would tell us?”

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. Often that the question card was vague, so there’s no way to know if the data answers it. Go back to the card. Sometimes it means the student made the chart before the question and is now trying to find a question that fits. Sometimes it means they’ve just seen the hole in their data and don’t want to say so. Make it easy: “Missing data is a finding. Write it down.”

Don’t say / say instead. Don’t say “That’s a nice chart”; say “What’s the one sentence this chart proves, and what can’t it show?”

An incorrect example to diagnose

Hand the student this, labeled as a misleading chart: a bar chart of the four months where the vertical axis starts at 450. June’s bar stands more than ten times as tall as September’s. Ask: “Read me the actual numbers.” June used 880 units and September 490: under twice as much. Then: “Who might draw it this way on purpose? Who might do it by accident?” Both happen. Spotting it is the skill.

Mixed practice

Once each card has been done once, mix the stages. Give the student a small dataset with no instructions and see whether they reach for the question card first. Alternate a Python cleaning Hour with a SQL Hour on the same data. Keep the monthly AS-01 Cold Trace from Chapter 1,

and once a week ask, from memory, what WHERE and GROUP BY do. At the end of the second week, AS-03 Explain Your Code works well with a query: the student explains each clause of a join.

For the student

Data is everywhere, and most of what people say about it is wrong. Not because they're lying, usually, but because they started with a chart instead of a question. This chapter teaches you to start in the right place.

Question first. Before you open a single file, write one question on a card. Make it one you could actually answer: “Which month do we use the most water?” not “Are we wasteful?” On the back, write which columns would answer it.

Dictionary second. For each column: name, type, allowed values. “Reading: whole number, never smaller than last month's.” That one rule will catch typos for you later.

Then clean it, in code, with a log. Here's the heart of a cleaning program:

```
text = (row["reading"] or "").strip()
if not text.isdigit():
    log.append(f"dropped {date}: reading was {text!r}")
    continue
```

.strip() removes stray spaces. .isdigit() checks that what's left is a whole number. Anything that fails is logged, not silently deleted. When we ran this on a file with “n/a” in it, the log said dropped 2026-08-01: reading was 'n/a'. That log is how a stranger can trust your answer.

One honest chart. The title is the answer. The axis starts at zero (for bars). The caption says what the data can't show. Make one version with the axis cut off, once, just to see how much it changes the story. You'll never read a news chart the same way.

Databases. When data grows, it moves into tables that link by keys. A loans table doesn't repeat Ana's name; it just says person 1 borrowed book 3. A query puts it back together:

```
SELECT title, pages FROM books
WHERE pages > 200
ORDER BY pages DESC;
```

Read it out loud: select the title and pages from books, where pages are over 200, sorted by pages, biggest first. SQL is one of the most widely used languages in computing, and you can learn the core of it in a week.

Try this today. Write a question card about something in your house that isn't about people (books, weather, water, electricity, a garden). Write the data dictionary. Collect or find ten rows. Clean them in code with a log, even if they're already clean (plant one mess to make sure your code catches it).

Explain it back. Pick one cleaning decision you made and explain it to a parent: what was wrong, what you did, and why that was better than the alternatives. Was dropping the row better than guessing a value? Why? Then explain a join: which key connects which tables?

This part is hard for everyone. Messy data. Real data is always messier than you expect. Spaces, typos, blanks, a date written three ways, a number in the wrong column. Professionals spend a large share of their data time cleaning. It isn't the boring part before the real work; it is the real work, and the log is what makes it trustworthy.

About “proof.” Data rarely proves anything. It supports, or it doesn't. If two things go up together, ask what else could explain both. If someone shows you a chart, ask what's missing from it. And if your own data has a hole in it, say so in the caption. That's not weakness. That's what makes people believe the rest of your chart.

If you're 17 or older: connect Python to SQLite (SQL-04) and use parameterized queries, with the `?` placeholder, every single time a query includes anything a user typed. Never glue user input into a SQL string. Chapter 7 will show you why, on your own toy app. Also try a join across three tables, and explain what would happen if you got the key condition wrong. (Try it. On a five-loan, three-person database, the wrong version returns 15 rows.)

A worked cleaning step. Picture one row that reads 2026-07-01, 11690 with a space before and after the number. Without `.strip()`, the text is " 11690 ", and `.isdigit()` says no, because spaces aren't digits. Your program would drop a perfectly good reading and log it as bad. With `.strip()`, the text becomes "11690", passes the check, and becomes the number 11690. One method call is the difference between a missing July and a correct one. That's why you predict how many rows will survive before you run the cleaner: if your prediction is five and the program says four, you've found something.

A query to write today. Once you've built a small database, write this one yourself before looking anything up: “Which books have more than 200 pages, longest first?” Then change one clause at a time. Take off `ORDER BY` and predict the order. Change `>` to `<`. Add a second condition with `AND`. Each change is one prediction and one run, exactly the loop from Chapter 3. Save the queries in a file with the results pasted under each one; that file is your crumb, and it shows a stranger exactly what you asked and what came back.

Challenge. Build a small database of three tables from your house (books, people with first names only, and loans; or plants, beds, and waterings). Write a join that answers a real question, then find the row the join leaves out and explain why it's missing. Write it up in three sentences: the question, the answer, and what the data can't tell you.

What to save: your question card, data dictionary, original and cleaned files, change log, chart with caption, query file, and Debug Log pages. Save them in two places.

If it isn't clicking

The chart comes before the question. It's a tool-first habit, and the tools make it easy. The fix is physical: the question card goes on the table, face up, before the laptop opens. If the chart's title isn't an answer to the card, it's not done.

Joins return too many rows. The keys aren't clear yet. Back to SQL-01 on paper, with arrows. Ask the student to trace one loan row by hand, from its `person_id` to the right person and its

book_id to the right book. Then write the ON condition from the arrow.

“The data proves...” Cause and correlation have merged. Ask “What else could explain it?” and wait. Then ask what data would tell the two explanations apart. This is a habit, built one question at a time, not a lecture.

Cleaning feels pointless. The data looked fine. Plant three messes in a copy and let the student’s code find them. Then find a public dataset with real ones. When the student sees their first real “n/a,” the log stops feeling like busywork. When they ask for rescue, the order is the same as ever: ask → wait → hint → short demo on your own copy → hand it back. If data becomes the student’s passion, a statistics course, a co-op data project, or a dual-enrollment class (Chapter 9) can take it further, and “Data Science” can become its own transcript line.

Tools, including AI

Python’s csv and sqlite3 modules are built in. The sqlite3 command-line tool is a small free download if your student wants to type SQL directly. Spreadsheets are fine for looking at data, and for charts at 15–16, as long as the cleaning happens in code, where it’s logged and repeatable. A charting library such as matplotlib is a free extra, installed separately. SQLite’s own website has a full, readable reference for SQL.²⁷

The AI rules are the ones in the AI box in **The Computing Hour**. Here’s what differs. AI chat tools are fluent at writing SQL and at describing what a chart “shows,” and both are places where confident output can be quietly wrong. At 15–16, chat help may explain what a query clause does, after the student has written their own prediction. No personal data, family data, or anyone’s names go into a chatbot, ever; that includes pasting a dataset in to “clean it for you.” The editor coding assistant stays off through 16 and for every check. At 17, on the one named project, any suggested query is run against a small table where the student already knows the right answer before it’s kept. Check the current terms of any tool before your student uses it.

What “done enough” looks like

Move on when most of these are “seen,” a few are “growing,” and none of the first three is “not yet”:

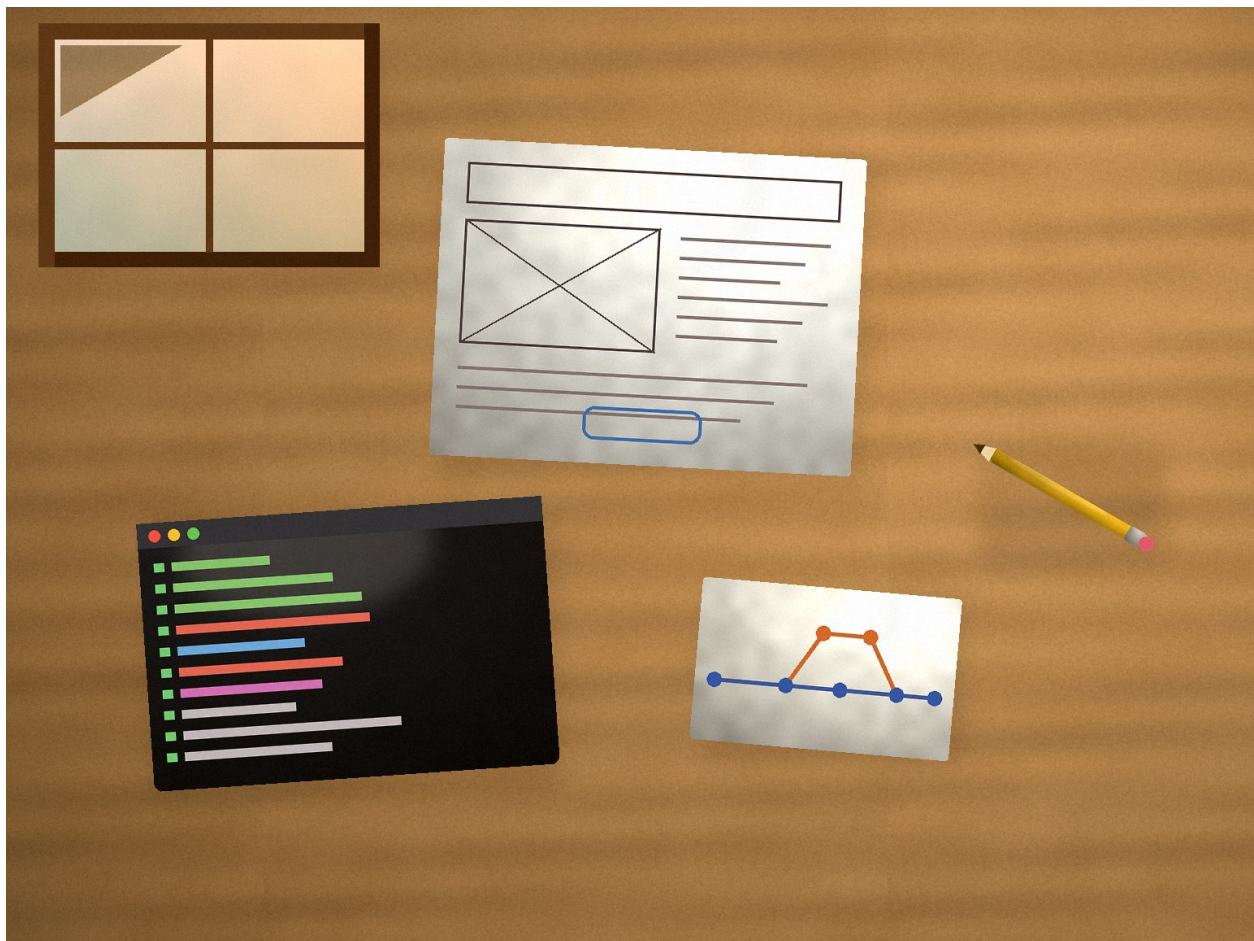
- ☐ Writes a question card with a checkable question before opening any data (DATA-01)
- ☐ Writes a data dictionary with name, type, and allowed values for each column (DATA-02)
- ☐ Cleans a file in code, with a change log, without touching the original (DATA-03)
- ☐ Makes one honest chart with a title that answers the question and a caption that names a limit (DATA-04)
- ☐ Writes two sentences about who or what the data leaves out (DATA-05)
- ☐ Splits a one-big-table mess into linked tables on paper, with keys (SQL-01)
- ☐ Writes SELECT, WHERE, and ORDER BY queries and predicts their results (SQL-02)
- ☐ **AS-03 Explain Your Code:** explains a cleaning program or a query, line by line or clause by clause. You write one line about how it went.

- **17–18 (ready 16):** writes a join with `GROUP BY` and explains the key (SQL-03); **17–18:** uses `sqlite3` from Python with parameterized queries (SQL-04).

Placement. Place by skill, not birthday. A 15-year-old who writes question cards and cleans with a log is ready for Chapter 6. A 17-year-old whose charts still come before questions needs another week here. If your student loves this chapter, let it run long; a data-heavy semester is a real course.

Chapter 6

Building for People: The Web, the Command Line, and Git



Why this matters

Here is an illustration, not a real family. Picture a fifteen-year-old building a one-page site for his grandmother's pancake recipe. He's proud of the design: a pale gray font on a cream background, a custom-drawn "button" that changes color when the mouse is over it. Then his grandmother tries it. She can't read the gray text without her glasses. She doesn't use a mouse; she uses the keyboard

on her tablet stand. She presses Tab to reach the button, and nothing happens, because the “button” isn’t really a button at all. It looks like one. It isn’t one.

He didn’t do anything careless. He built for the person he knew best, which was himself. This chapter is about building for someone else. Software is for people. The web is how most software reaches them, and every choice on a page (the size of the text, the order of the headings, whether a button is a real button) decides whether a real person can use it.

The chapter has three parts, and they belong together. **The web:** HTML for structure, CSS for appearance, JavaScript for behavior. **The terminal,** or command line: typing commands to the computer instead of clicking, which is where programmers do much of their daily work. And **Git:** a tool that keeps the history of every saved change. Each saved change is a **commit**, with a message saying what changed and why. Git is how programmers keep a project’s story, go back when something breaks, and work together without overwriting each other.

This is also the chapter where your student’s work starts to look like a professional’s. A small site, built for a named user, tested with the keyboard, kept in a repository with real commit messages and a README a stranger can follow, is a serious piece of work. It’s portfolio material. If the year leans heavily this way, it can even become its own transcript line: **Web Development**. Chapter 9 explains how to describe it.

One honest note: “every programmer uses X” is never quite true. Git, the terminal, and the web’s three languages are *widely used*, across most of the industry and in college courses. That’s reason enough to learn them, without pretending they’re universal.

For the parent: understand it yourself

You don’t need to build websites to teach this chapter. You need to be a demanding first user, know what a good commit message looks like, and be able to say “Try it with the keyboard only.” You’ll have all three by Friday.

The everyday picture. A house has a frame, paint, and wiring. HTML is the frame: walls, rooms, doors, which room is the kitchen. CSS is the paint and furniture: colors, sizes, where things sit. JavaScript is the wiring: press a switch and a light comes on. You can have a frame with no paint; you can’t paint a house that has no frame. And a door that’s only painted on the wall isn’t a door.

The precise picture. The national high-school standards from the Computer Science Teachers Association put the goal in one line: “Design a computing technology using human-centered design principles.”²⁹ *Human-centered* means you start with a person, not a feature list. Who is this for? What will they see, do, and need first? That’s this chapter’s talk-box opening, and it’s the first line of every web spec.

What a page looks like underneath

Here’s the whole recipe page, cut down to fifteen lines:

```
<!DOCTYPE html>
```

```

<html lang="en">
<head><meta charset="utf-8"><title>Nana's Pancakes</title>
  <link rel="stylesheet" href="style.css"></head>
<body>
  <main>
    <h1>Nana's Pancakes</h1>
    <ul><li>2 cups flour</li><li>2 eggs</li><li>1 1/2 cups milk</li></ul>
    <label for="batches">How many batches?</label>
    <input id="batches" type="text">
    <button id="scale" type="button">Scale it</button>
    <p id="result" aria-live="polite"></p>
  </main>
  <script src="script.js"></script>
</body></html>

```

Every piece in angle brackets is a **tag**, and most come in pairs that wrap content: `<h1>` starts the main heading and `</h1>` ends it. The words are chosen for meaning, not looks. `<main>` says “this is the main content.” `<ul>` and `<li>` say “this is a list.” `<label for="batches">` ties the words “How many batches?” to the box with `id="batches"`, so a screen reader (software that reads a page aloud for someone who can’t see it) announces them together. `<button>` is a real button. HTML written this way, where each tag says what the thing *is*, is called **semantic HTML**, and it’s most of what makes a page accessible.³⁰

A little style

CSS is a list of rules: which things, and how they should look.

```

body {
  font-family: sans-serif;
  font-size: 1.25rem;
  max-width: 40rem;
  margin: 0 auto;
  padding: 1rem;
  color: #1a1a1a;
  background: #ffffff;
}
button {
  font-size: 1.25rem;
  padding: 0.5rem 1rem;
}

```

This makes the text a bit larger than the default, keeps lines from stretching too wide to read, centers the page, and uses near-black text on white. That last choice is about **contrast**, the difference between text and background. MDN, the free web reference this chapter leans on, notes that text color has to be significantly different from the background, and gives 4.5 to 1 as the contrast ratio to aim for in ordinary text.³⁰ Pale gray on cream fails. Near-black on white passes easily.

A little behavior

JavaScript makes the page respond:

```
const button = document.querySelector("#scale");
const result = document.querySelector("#result");

button.addEventListener("click", () => {
  const text = document.querySelector("#batches").value.trim();
  const n = Number(text);
  if (text === "" || !Number.isInteger(n) || n < 1) {
    result.textContent = "Please type a whole number, like 2.";
    return;
  }
  result.textContent = `For ${n} batches: ${2 * n} cups flour, ${2 * n} eggs.`;
});
```

In plain words: find the button and the result paragraph; when the button is clicked, read what was typed, trim the spaces, and check that it's a whole number of at least 1. If not, say so politely. If so, scale the recipe. When we ran the page in a current desktop browser and typed 3, it showed “For 3 batches: 6 cups flour, 6 eggs.” Typing two, 1.5, 0, or nothing at all showed “Please type a whole number, like 2.” Typing 2 with spaces worked, because of `.trim()`.³¹

The page's structure, the **DOM** (Document Object Model), is what JavaScript reaches into. `document.querySelector("#scale")` means “find the thing whose id is scale.” That connects to Chapter 4: the page is made of objects, and JavaScript calls their methods.

The terminal in one paragraph

A **terminal** shows a prompt and waits for a typed command. Three commands cover most of a beginner's day: `pwd` (“print working directory,” meaning where am I?), `ls` (list what's here), and `cd` (change directory, meaning move into a folder). `mkdir` makes a new folder. A **path** says where a file lives: an *absolute* path starts from the top of the drive (`/home/sam/projects/recipes`), a *relative* path starts from wherever you are (`recipes`, or `..` for “one folder up”). Running a program from the terminal is just typing its command: `python3 hello.py`. The terminal isn't dangerous if you stay in your own project folders, though a few commands can delete files without asking, which is why the practice folder in TERM-02 below exists.

Git in one paragraph

A **repository** is a project folder whose history Git keeps. `git init` starts one. `git add` chooses which changes go into the next save point. `git commit -m "message"` saves them, with a message. `git log` shows the history. When we made a small recipe repository and ran `git log --oneline`, it showed:

```
9e7df23 Add large text style for easier reading
190c493 Add recipe page with ingredient list
```

Each line is one commit: a short code that identifies it (yours will be different; they're generated) and the message. Read top to bottom, newest first, it's the project's story. Git is **history, not backup**: it lives inside the project folder, so if the laptop dies, the history dies with it unless the folder is also saved somewhere else. The rule from **The Computing Hour** still applies: two places.³²

Wrong answers you should be able to hear

- **“It looks fine on my screen.”** Tested on one screen, with a mouse, by the person who built it. Say: “Try it with the keyboard only. Then try it on my phone.”
- **“My commit message is ‘stuff.’”** History isn't useful yet. Say: “If you came back in a month, what would ‘stuff’ tell you?”
- **“I put the password in the code so it works.”** Anything committed stays in the history, even after it's deleted from the file. Say: “Let's take that out before the next commit, and change that password today.”
- **“It works if you click.”** A page that only works with a mouse leaves people out. Say: “What happens when you press Tab?”
- **“Git is my backup.”** History confused with copies. Say: “Where's the second place?”

Your five-minute warm-up

Open any website you use often. Put the mouse aside. Press Tab, again and again, and watch where the focus goes (usually a highlighted outline). Can you reach every link and button? Can you see where you are? Press Enter on a button. Then say this chapter's opening question once, out loud: “Who is this for — and what will they see, do, and need first?” That's your preparation, and you've just done your first accessibility test.

How to teach it this week

This is a three-week chapter for most families: one week for the page, one for the terminal and Git, and one for a finished project with a README. Run each Hour inside the six steps of **The Computing Hour**.

- **Specify.** Every web spec starts with a named user (first name or role only) and what they need first. At 17–18 (ready 16), add a user story: “As a grandmother cooking at the stove, I want large text and one clear button, so that I can scale the recipe without my glasses.” The SPEC-04 User Story and Acceptance Test card from Chapter 1 gives the acceptance line: “Done when the recipe scales correctly using only the keyboard.”
- **Predict.** “What will the page look like after this change?” “What will `git log` show after this commit?”
- **Build and Test.** Small change, save, refresh, look. That rhythm is the web's version of Predict and Build and Test.
- **Debug and Log.** The browser's developer tools have a Console that reports JavaScript errors; it's the web's traceback.³³ Expected/Actual first, as always.

- **Review.** Keyboard-only test before every commit. Then commit with a message that says what changed and why.
- **Ship and Explain.** “I specified... I predicted... It did... Next I’d change...” and “Help I used: ...”

The first week: a page for a person. Begin on paper.

In-home — WEB-01 Paper Wireframe and Audience

- **Time:** 15 minutes
- **Materials:** paper, pencil, a named user (a family member who agrees to be it)
- **Safety and privacy:** first names only; no photos of the user
- **The fun:** designing for someone who will actually tell you what they think
- **The skill:** sketching a page for a named user and labeling its structure
- **The crumb:** a wireframe

How it goes. A **wireframe** is a box drawing of a page: no colors, no pictures, just where things go. At the top of the sheet the student writes the user’s first name and the one thing they need first (“Nana: the ingredient list, big”). Then boxes for each part of the page, each labeled with what it *is* (main heading, list, label and box, button, result). Show it to the user before building anything. Ask them what they’d look for first. Move boxes until the answer is “the top.”

15–16: one page, one user. **17–18:** two users with different needs (one who uses a keyboard, one on a phone) and a sentence about how the page serves both.

In-home — WEB-02 Semantic HTML and CSS

- **Time:** 40 minutes
- **Materials:** a plain editor, a browser, the wireframe
- **Safety and privacy:** local files only this week; nothing is published
- **The fun:** the drawing becomes a page
- **The skill:** headings, landmarks, lists, and forms; a stylesheet with a layout
- **The crumb:** `index.html` and `style.css`

How it goes. The student builds the page from the wireframe, one box at a time: type a few lines, save, refresh the browser, look. Each box on the wireframe becomes the tag that says what it is. Then a stylesheet: readable size, a comfortable line width, strong contrast. Predict before every refresh.

15–16: start from the shape in this chapter and change it to fit their wireframe. **17–18:** start from a blank file, using MDN as the reference.

In-home — WEB-03 JavaScript That Responds

- **Time:** 35 minutes
- **Materials:** the page, a browser with developer tools
- **Safety and privacy:** none needed

- **The fun:** the page talks back
- **The skill:** event listeners, the DOM, and input validation
- **The crumb:** `script.js`

How it goes. Write the test table first (Chapter 3 never stops): 3 should scale; two, 1.5, 0, and blank should get the polite message; 2 should work. Then write the script. Open the Console before the first run, so an error is seen rather than guessed at. The validation lines are the most important ones on the page: they’re where the page treats a confused user kindly.

15–16: one button, one message. **17–18:** add a second behavior (say, switching between cups and grams) and test it the same way.

In-home — WEB-04 Accessibility Pass

- **Time:** 20 minutes
- **Materials:** the page, the keyboard, a checklist
- **Safety and privacy:** none needed
- **The fun:** finding what you couldn’t see
- **The skill:** alt text, heading order, contrast, and a keyboard-only test
- **The crumb:** a checklist with fixes

How it goes. Four checks, each written down with a fix:

1. **Keyboard only.** Put the mouse away. Tab to everything. Press Enter on the button. Does it work? Can you see where the focus is?
2. **Headings in order.** One `<h1>`, then `<h2>`s, never skipping levels just to get a smaller size. Many screen-reader users jump from heading to heading to find their way.³⁰
3. **Alt text.** Every meaningful image has alt text that says what it shows; a purely decorative one gets `alt=""`.³⁰
4. **Contrast.** Text clearly darker (or lighter) than its background.

Here’s the story from the opening, finished. The custom “button” was a `<div>` styled to look like a button. It worked with a mouse and did nothing from the keyboard, because a `<div>` can’t receive keyboard focus on its own. The fix was one change: use a real `<button>`. The commit message read “Use semantic button for keyboard users.” MDN’s accessibility guide makes the same point: use real buttons and links for controls.³⁰

Both bands. At 17–18, add a phone-sized window and a zoomed-in browser (200 percent) to the pass.

The second week: the terminal and Git.

In-home — TERM-01 Command Line on Paper

- **Time:** 10 minutes
- **Materials:** paper, pencil
- **Safety and privacy:** none needed
- **The fun:** a treasure map made of folders

- **The skill:** a directory tree; `cd`, `ls`, `mkdir`; relative versus absolute paths
- **The crumb:** a tree sketch with paths

How it goes. Draw the folder tree: a top folder, a projects folder inside it, and recipes and notes inside that. You read out a list of commands (`cd projects`, `cd recipes`, `cd ..`, `mkdir images`), and the student follows with a pencil, predicting where they end up after each and what `ls` would show. Then write the absolute path to one file and a relative path to the same file from another folder.

In-home — TERM-02 First Terminal Session

- **Time:** 20 minutes
- **Materials:** the computer's own terminal, a practice folder that's safe to mess up
- **Safety and privacy:** work only inside the practice folder; never type a command copied from a stranger or a video without knowing what it does
- **The fun:** making the computer do things with no mouse at all
- **The skill:** navigating, making folders, running a script from the terminal
- **The crumb:** a command history excerpt

How it goes. The student repeats yesterday's paper map on the real computer, predicting each output first. In a practice folder containing projects/recipes, a session we ran looked like this (the \$ is the prompt; it isn't typed):

```
$ cd projects
$ ls
recipes
$ mkdir notes
$ ls
notes  recipes
$ cd recipes
$ python3 hello.py
Hello from the terminal
$ cd ..
```

`hello.py` was a one-line program that printed that greeting. (On Windows the Python command may be `python` or `py` instead of `python3`, and the listing command in older terminals is `dir`.) The student saves the last screen of commands as the crumb.

15–16: you sit nearby; the student reads each command aloud before pressing Enter. **17–18:** from here on, the student runs their Python programs and tests from the terminal every day.

In-home — GIT-01 Commits as Save Points (Paper)

- **Time:** 10 minutes
- **Materials:** a long strip of paper, sticky notes
- **Safety and privacy:** none needed
- **The fun:** a timeline you can hold
- **The skill:** snapshots, messages, and history drawn as a line

- **The crumb:** a paper history

How it goes. The strip is the timeline. Each sticky note is a commit: the student writes a message for each change they made to the recipe page last week, in order. Then you ask: “Which note would you go back to if the button broke?” If the messages say “stuff” and “more stuff,” nobody can answer. A branch is a second strip that starts from one note and rejoins later.

In-home — GIT-02 First Repository

- **Time:** 25 minutes
- **Materials:** Git installed on the student’s own computer, the recipe folder
- **Safety and privacy:** local only; no account needed for any of this
- **The fun:** a time machine for your project
- **The skill:** `init`, `add`, `commit`, and `log` on a local project
- **The crumb:** a `git log` excerpt

How it goes. In the terminal, inside the recipe folder: `git init` once; then, for each change, `git status` (what’s changed?), `git add` with the file name, and `git commit -m` with a real message. `git log --oneline` shows the story. Predict what the log will show before running it. One setup step comes first: Git asks for a name and email to put on each commit. A first name and a family email the parent chooses are fine; this information becomes public if the repository is ever published.³²

In-home — GIT-03 Good Commit Messages

- **Time:** 10 minutes
- **Materials:** a list of bad messages, a pencil
- **Safety and privacy:** none needed
- **The fun:** fixing other people’s terrible messages
- **The skill:** rewriting vague messages into what-and-why lines
- **The crumb:** a before/after list

How it goes. Write five bad messages: “stuff,” “fix,” “changes,” “asdf,” “final final.” The student rewrites each as a line that says what changed and, where useful, why: “Use semantic button for keyboard users.” “Add recipe list.” “Fix scaling for blank input.” A good habit: the message completes the sentence “This commit will...” Chapter 3’s Review step uses this card too: test and fix go in one commit, with a message that names the bug.

In-home — GIT-04 Branch, Change, Merge

- **Time:** 30 minutes
- **Materials:** the recipe repository
- **Safety and privacy:** none needed
- **The fun:** making a conflict on purpose and winning
- **The skill:** a feature branch, a merge, and one resolved conflict
- **The crumb:** a log graph excerpt

How it goes. A **branch** is a separate line of work. The student runs `git switch -c bigger-text` to start one, changes the text size, and commits. Then `git switch main`, changes the *same line* to a different size, and commits again. Then `git merge bigger-text`. Git can't choose between the two edits, so it stops and marks the conflict in the file:

```
<<<<<<< HEAD
    font-size: 1.4rem;
=====
    font-size: 1.5rem;
>>>>>>> bigger-text
```

The top half is the current branch's version; the bottom half is the branch being merged. The student deletes the markers, keeps the right line (or writes a new one), saves, runs `git add` on the file and commits. When we did exactly this, `git log --oneline --graph` drew the branch leaving the main line and rejoining it at the merge commit. A conflict isn't an error. It's Git asking a human to decide.

17–18: at least one real branch per project from here on. **Ready 16:** a 16-year-old who's comfortable with commits can try it now.

In-home — GIT-05 README a Stranger Can Use

- **Time:** 20 minutes
- **Materials:** the project, a plain text file named `README.md`
- **Safety and privacy:** no full names, school, town, or photos
- **The fun:** watching someone use your project without asking you anything
- **The skill:** what it does, how to run it, what helped, and known limits
- **The crumb:** the `README` file

How it goes. Four short sections: **What it does** (one or two sentences), **How to run it** (exact steps), **Help I used** (the same line as the Ship and Explain step, including “none”), and **Known limits** (what it doesn't do yet). Then the test: hand the folder to a family member who hasn't seen it and say nothing. Can they run it from the `README` alone? Every question they ask is a missing line.

In-home — PROJ-02 Version History and Changelog

- **Time:** 10 minutes
- **Materials:** the repository
- **Safety and privacy:** none needed
- **The fun:** a story a stranger can read
- **The skill:** commits plus a human-readable changelog
- **The crumb:** a `CHANGELOG` file

How it goes. Git's log is for programmers. A changelog is for users: a short dated list of what's new in plain words. “14 Oct: you can now scale the recipe from the keyboard.” At the end of each week, the student reads the log and writes the changelog lines. Chapter 9 collects these as evidence.

In-home — BUG-04 Bisect the Bug

- **Time:** 20 minutes
- **Materials:** a repository with at least eight commits and a bug that wasn't there at the start
- **Safety and privacy:** none needed
- **The fun:** letting Git do the halving
- **The skill:** halving the history to find where the bug starts
- **The crumb:** bisect notes

How it goes. Chapter 3 halved code by hand. Git can halve history. `git bisect start`, then `git bisect bad` (the current version has the bug), then `git bisect good` followed by the code of an older commit that didn't. Git jumps to a commit halfway between. The student tests it and says `git bisect good` or `git bisect bad`. Git halves again. In a few steps it names the first bad commit. `git bisect reset` returns to normal. Good commit messages make the answer instantly useful; “stuff” makes it a mystery. **17–18 (ready 16).**

Practice that actually builds learning

Practice in this chapter is mostly building, and the third week is a whole small project: a page for a named user, in a repository, with a README.

Out-of-home — OUT-03 Ask a Working Programmer

- **Time:** 30 minutes
- **Materials:** five questions written in advance, a notebook
- **Safety and privacy:** an adult the family knows (relative, co-op parent, neighbor); a public place or a video call with a parent present; first names only in notes
- **The fun:** a real person's real job
- **The skill:** a short interview with an adult who writes code at work, with one question about testing
- **The crumb:** interview notes (first name only)

How it goes. The student writes five questions before the meeting. Two are required: one about testing (“How does your team know a change didn't break something?”) and one about how the team decides what to build. Others might be: “What does a good commit message look like at your job?” “What's the last bug that took you a whole day?” “How do you work with people who use what you build?” Afterward, the student writes the answers in their own words and one thing that surprised them. Thank-you notes are a good habit. If your student did a short version in Chapter 4, this is the full one.

In-home — TEST-05 First User, Weekly

- **Time:** 15 minutes
- **Materials:** the page and you
- **Safety and privacy:** none needed

- **The fun:** you get to be difficult
- **The skill:** the parent uses the program like a stranger and reports, without fixing
- **The crumb:** a first-user note

How it goes. You know this card. On the web it has a few new moves: use only the keyboard; zoom the browser in; try it on a phone; type something odd into the box. Report in one sentence what happened. Every surprise becomes a test row and then a commit.

Talk box (this chapter)

Opening: “Who is this for — and what will they see, do, and need first?”

Follow-ups: 1. **Prediction:** “Before you refresh, what will change on the page?” 2. **Evidence:** “How do you know Nana can use it? What did you test?” 3. **Location:** “Which line in the HTML makes that button a real button?” 4. **Change:** “If she only used the keyboard, what would you change first?” 5. **Revision:** “What would your commit message say about this change, and why?”

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. Often that the student hasn’t actually met the user yet, so “what will they need first?” has no answer. Arrange the meeting. Sometimes it means the student has made something beautiful and is afraid you’re about to say it’s not usable. Keep it about the page: “Let’s see what the keyboard says.” And sometimes the terminal or a merge conflict has them frozen; say “It’s a practice folder. Nothing here can hurt anything that matters.”

Don’t say / say instead. Don’t say “That’s not how real websites look”; say “Let’s watch your user try it and see what they need.”

An incorrect example to diagnose

Show the student this commit history, labeled as a poor one:

```
a1b2c3d stuff
d4e5f6a more stuff
0a1b2c3 fixed it
3c4d5e6 final
7f8a9b0 final final
```

Ask: “The scaling broke somewhere in here. Which commit would you go back to?” No one can say. Then ask the student to rewrite each message as if they knew what it did. Then show them a second, imaginary history whose third commit reads “Hard-code password for test login.” Ask: “What’s wrong with this one, even if the next commit removes it?” The password is still in the history, readable by anyone who ever gets a copy. Anything secret goes in a file Git is told to ignore, and a password that ever got committed gets changed.

Mixed practice

By the third week, mix everything into one small project: a spec with a named user, a wireframe, a page with one behavior, an accessibility pass, a repository with real messages, a branch, and a README that a family member tests. Keep the monthly AS-01 Cold Trace from Chapter 1; this month, a few lines of JavaScript: “What will the page say if I type 0?”

For the student

You’ve been writing programs for yourself. This chapter is about writing for other people, which is what nearly all real software is.

Start with a person. Before you open an editor, write down who your page is for (first name or role) and the one thing they need first. Then draw a wireframe, a box sketch of the page, and show it to them. Ask what they’d look for first. Move boxes until the answer is “the top.”

Three languages, three jobs. HTML says what things *are*: a heading, a list, a button. CSS says how they look. JavaScript says what they do. Write the HTML first, with tags that mean something: `<main>`, `<h1>`, `<ul>`, `<label>`, `<button>`. Then style it. Then make it respond.

Here’s the heart of a page that responds:

```
button.addEventListener("click", () => {
  const text = document.querySelector("#batches").value.trim();
  const n = Number(text);
  if (text === "" || !Number.isInteger(n) || n < 1) {
    result.textContent = "Please type a whole number, like 2.";
    return;
  }
  result.textContent = `For ${n} batches: ${2 * n} cups flour, ${2 * n} eggs.`;
});
```

When someone clicks, it reads the box, checks it’s a whole number, and either scales the recipe or politely asks again. The check is the part that respects your user. Test it like Chapter 3: 3, two, 1.5, 0, blank.

Test with the keyboard. Put your mouse in a drawer. Tab through your page. Can you reach everything? Can you see where you are? Does Enter press the button? If you built a fake button out of a `<div>`, it won’t. Use a real `<button>`. This one test catches more problems than any other.

The terminal. It looks old-fashioned. It’s where programmers live. `pwd` tells you where you are; `ls` shows what’s here; `cd` moves you; `mkdir` makes a folder; `python3 hello.py` runs a program. Practice in a folder that’s safe to mess up, and never paste a command you don’t understand, no matter who posted it.

Git. Every time you finish a small change that works, commit it:

```
git add index.html
git commit -m "Use semantic button for keyboard users"
```

The message should finish the sentence “This commit will...” In a month, your log should read like the story of your project. “stuff” isn’t a story. Git is history, not backup: keep your project in two places as well.

Try this today. Pick a real person in your family and ask what small page would help them: a recipe, a chore list, a plant-watering schedule. Draw the wireframe and show them. Build the HTML. Make three commits with real messages. Run `git log --oneline` and read your story.

Explain it back. Show a parent your page and explain why each tag is the tag it is. Why `<button>` and not a `<div>`? Why one `<h1>`? Why a `<label>`? Then explain your three commits: what each one did and why.

This part is hard for everyone. Merge conflicts. The first time Git stops and fills your file with `<<<<<<<` and `>>>>>>>`, it feels like something broke. Nothing did. Git found two different edits to the same line and is asking you to choose. Read both versions, keep the right one, delete the markers, add the file, commit. Making a conflict on purpose in a practice repository (GIT-04) is the best way to stop fearing them.

If you’re 17 or older: use branches for every new feature. Write a README a stranger can use, and test it on a stranger (well, a family member who hasn’t seen the project). Learn `git bisect` for the bug that showed up sometime last week. And if you want to publish, that’s a family decision, made together (PROJ-04, below). Code-hosting sites require users to be at least 13 under their terms, which you are, but whether and where to publish, and what goes in it, is something you decide with your parent.

A worked commit history. Here’s what a good week can look like in `git log --oneline` (the codes at the front are generated, so yours will differ):

```
9e7df23 Add large text style for easier reading
190c493 Add recipe page with ingredient list
```

Two commits, and a stranger already knows what the project is and what changed. By Friday there might be six more: “Add batch box and scale button,” “Validate blank and non-number input,” “Use semantic button for keyboard users,” “Add README with how to run it.” Read it top to bottom and it’s the story of your week. If something breaks next month, you’ll know exactly which commit to go back to.

What a first user will teach you. When your parent or grandparent tries the page, don’t explain anything. Watch. Where do their eyes go first? What do they try to click that isn’t clickable? What do they type into the box? Write it down without arguing. Every surprise is a test row and a commit. That’s what “building for people” means in practice.

Challenge. Do a full keyboard-only test of your page, write down every problem, fix each one in its own commit with a message that says what you fixed for whom, and update your README’s “Known limits.” Then do the same test on a website you use every day. You’ll be surprised how many professional sites fail it.

What to save: your wireframe, your HTML, CSS, and JavaScript files, your accessibility checklist, your `git log`, your README and changelog, and your interview notes. Keep the whole repository in two places.

If it isn't clicking

The page works only with a mouse. Accessibility is still invisible. Make the keyboard-only test part of every commit for two weeks, out loud. Then have the student watch their user try it. One watched struggle teaches more than any checklist.

Commit messages are still “stuff.” History isn't useful yet, so it doesn't feel worth describing. Run GIT-03 again, then ask the student to find a specific change from last week using only their log. When they can't, the reason for good messages arrives on its own.

The terminal feels scary. It's unfamiliar and seems risky. Go back to paper (TERM-01), then make a practice folder with a silly name that's safe to mess up and spend ten minutes moving around in it with no goal. Read every command aloud before pressing Enter.

The project keeps growing. Every Hour adds a feature, and nothing is finished. Go back to the spec's done line. Finish, commit, write the README, and *then* write a new spec for version two. When you're asked for rescue, the order is the same as ever: ask → wait → hint → short demo on your own copy → hand it back. If the web becomes the student's main interest, a co-op web class, a dual-enrollment course, or a working web developer for one session (OUT-03) can take it further.

Tools, including AI

A plain code editor, a current browser, the computer's own terminal, and Git are all free and enough for this whole chapter. MDN Web Docs is the reference: free, thorough, and written for learners as well as professionals.³⁴ Everything here runs locally, with no account.

Accounts and publishing. A code-hosting account (GitHub is widely used) is optional. GitHub's terms require users to be at least 13.³⁵ Whether to make an account, and whether anything is published, is the parent's decision.

In-home — PROJ-04 Ship It (Optional Publish)

- **Time:** 30 minutes
- **Materials:** a finished project, the parent, a publishing choice the family has made together
- **Safety and privacy:** no names, faces, school, or town anywhere in the project, its history, or its README; check the commit author names and emails too
- **The fun:** a link you can show someone
- **The skill:** publishing a project with the parent's decision
- **The crumb:** a live link or a release note

How it goes. Before anything goes public, the student and parent read the whole project together: every file, the README, and `git log` (author names and emails included). Anything personal comes out, and if it was ever committed, the family decides whether to start a fresh repository instead. Then the parent decides whether and where to publish. Publishing is optional; a release note in the README (“Version 1.0, finished 30 Oct”) is a perfectly good crumb. **17–18 only.**

AI. The rules are the ones in the AI box in **The Computing Hour**. What differs here: AI tools write HTML, CSS, and JavaScript fluently, and pages built that way often *look* right while failing the keyboard test. At 15–16, chat help may explain a Console error or what a tag means, after the student’s own Expected/Actual. The editor coding assistant stays off through 16 and for every check. At 17, on the one named project, any suggested markup goes through the full accessibility pass before it’s kept, and the README’s “Help I used” says so. Never paste personal details, family photos, or anyone’s names into an AI tool to “build a page about” them. Check the current terms.

The transcript label. A semester or year where this chapter dominates (several finished sites for real users, a repository history, an accessibility checklist for each) can be listed as **Web Development**. Chapter 9 shows how to write the course description so a stranger can read it.³⁶

What “done enough” looks like

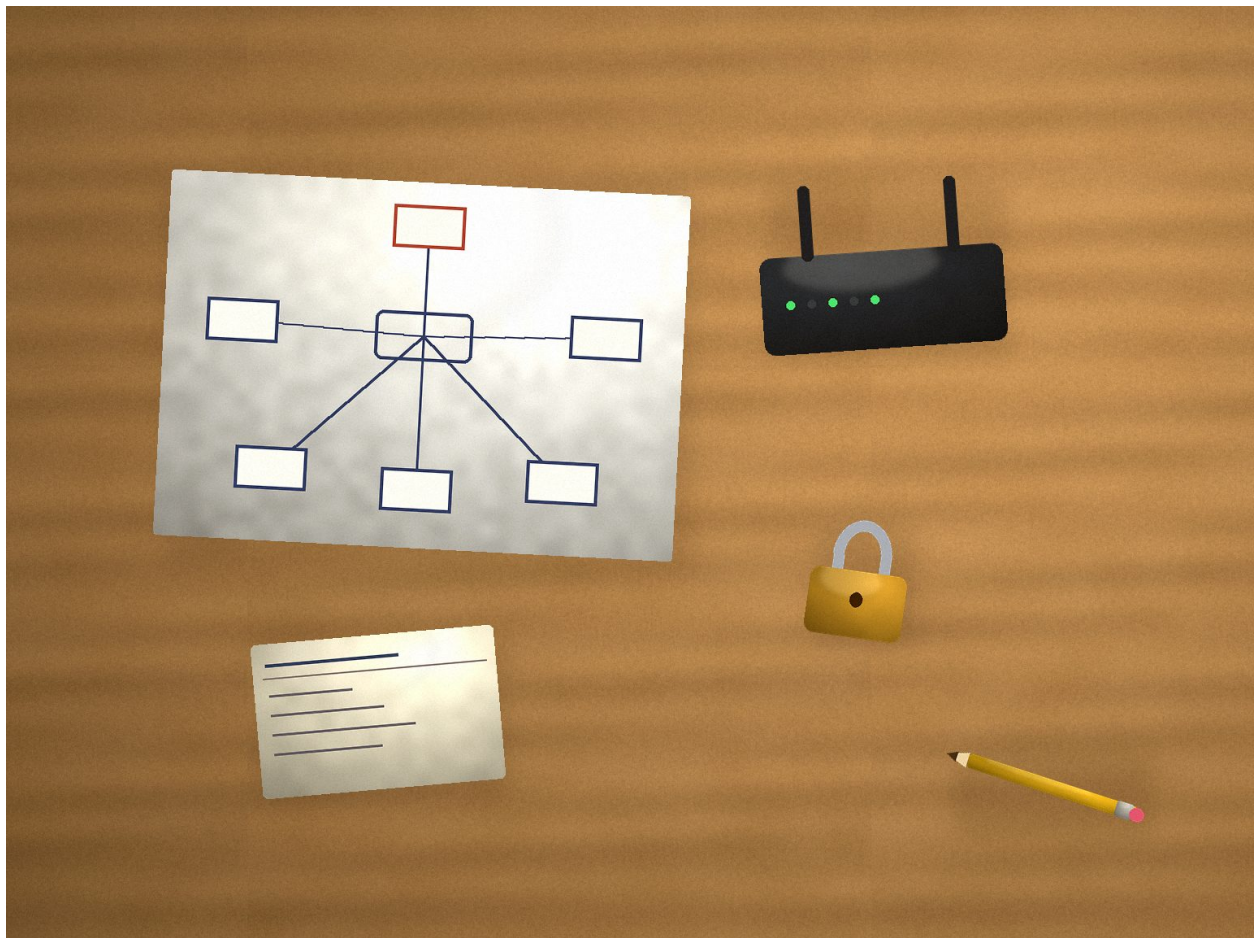
Move on when most of these are “seen,” a few are “growing,” and none of the first four is “not yet”:

- ☐ Draws a wireframe for a named user and says what they need first (WEB-01)
- ☐ Builds a page with semantic HTML and a readable stylesheet (WEB-02)
- ☐ Adds JavaScript that responds and validates input, tested with a table (WEB-03)
- ☐ Passes a keyboard-only test and fixes what fails (WEB-04)
- ☐ Moves around and runs a program from the terminal (TERM-01, TERM-02)
- ☐ Keeps a project in Git with commit messages that say what and why (GIT-01, GIT-02, GIT-03)
- ☐ Writes a README a family member can follow without help (GIT-05) and keeps a changelog (PROJ-02)
- ☐ Has interviewed a working programmer, including one question about testing (OUT-03)
- ☐ **AS-03 Explain Your Code:** explains the page tag by tag and the script line by line. You write one line about how it went.
- ☐ **17–18 (ready 16):** branches, merges, and resolves one conflict (GIT-04); bisects a bug in history (BUG-04); **17–18:** publishes only by family decision, with nothing personal (PROJ-04).

Placement. Place by skill, not birthday. A 15-year-old with an accessible page and an honest Git log is ready for Chapter 7. A 17-year-old whose pages only work with a mouse needs another week here. If this chapter catches fire, let it run a whole semester; that’s a real course with a real name.

Chapter 7

Systems, Networks, and Security: What You May and May Not Touch



Why this matters

By fifteen, your student has been using computers for most of their life. They have also been trusting them, which is a different thing. They type a web address and a page appears. They save a

file and expect it to be there tomorrow. They pick a password and hope it holds. This chapter opens the box. It shows what the operating system is doing behind the screen, how a message actually crosses a network, and why some protections work while others only feel like they do.

Knowing how things work brings a responsibility with it, and that's the second half of the chapter. A student who understands how a login form talks to a database also understands how someone could abuse one. At this age, that knowledge comes with a clear line: you practice on what you own, and nothing else. The line isn't there because we expect your student to do harm. It's there because curious, capable people sometimes cross it without meaning to, by "just checking" a friend's site or "just testing" the school's Wi-Fi. Testing someone else's system without permission can be a crime, even when the person doing it meant well.³⁷

So the chapter is built on a short written agreement, the **family security rule**, signed in the first week (SEC-01). Inside that rule there is a great deal of room. Your student can take apart an old computer, map the home network on paper, follow a web request across the internet with their own browser's tools, threat-model their own email account, and, at 17–18, build a small login app and break into it on purpose so they can fix it properly. That last one is the most satisfying hour in the book for many students. It's also the clearest lesson in why the rule exists.

There is a practical payoff, too. Nearly everything in this chapter protects your household. A teen who understands passphrases, two-step login, phishing, backups, and updates is, quite often, the most security-minded person in the house. Expect to learn something from them by the end of the month.

The Hour holds its shape. **Specify** means saying what you're protecting or what you expect a system to do. **Predict** means guessing where the packet goes, how long the passphrase takes to break, which message is the fake. **Build and Test, Debug and Log, Review, and Ship and Explain** work as always. The exit line still ends with "Help I used: ..."

For the parent: understand it yourself

The plain picture

A computer is a few parts doing a few jobs very quickly. The **processor** follows instructions. **Memory** holds what's being worked on right now and forgets it when the power goes off. **Storage**, a drive, keeps files when the power is off. The **operating system** is the manager that shares those parts among every program you run: it decides who gets the processor next, who gets which piece of memory, and who may open which file.

A **network** is computers passing messages. The message is chopped into small pieces called packets, each labeled with where it's going, and the pieces may travel by different routes before they're put back together at the other end. Your home **router** is the doorway between your devices and the internet. When you type a web address, your device first asks a directory service (DNS) for the number that address stands for, then sends its request to that number, and the reply comes back in packets.

Security is mostly about three questions: what are we protecting, from whom, and what's the cheap-

est strong fix? Security people call the answer a **threat model**: what you’re protecting, from whom, and the cheapest strong fix. For most families, most of the time, the answer is long passphrases, a password manager, two-step login, updates turned on, and a calm habit of checking before clicking.

The precise picture

Here is the slightly sharper version you’ll hear from your student by the end of the month.

- **Processes and permissions.** Each running program is a process with its own slice of memory. The operating system keeps processes from reading each other’s memory and checks permissions before a file is opened. Accounts on a computer exist so those permissions mean something.
- **Bits and encoding.** Everything stored is numbers, and the numbers are stored as bits (ones and zeros), eight to a byte. A letter is a number by agreement: capital A is 65. A color on a screen is usually three numbers, for red, green, and blue (HW-02).
- **Packets and protocols.** Protocols are the agreed rules of conversation. IP gets packets to the right machine; TCP makes sure they all arrive and in order; HTTP is how a browser asks for a page; HTTPS is the same conversation, encrypted so people along the route can’t read it or change it.
- **What HTTPS does not mean.** The lock icon tells you the connection is private. It says nothing about whether the site at the other end is honest. A scam site can have a lock.
- **Passwords.** Current federal guidance favors length over symbol tricks: at least 15 characters for a password used on its own, no forced periodic changes, and no rules that make people write “Password1!” on a sticky note.³⁸ A passphrase of four or more random words is easy to remember and hard to guess. A password manager remembers the rest.
- **Two-step login.** A password plus a second thing, such as a code from an app or a physical key. Security folks call this **multifactor authentication**, which means the same thing: proving who you are with more than one kind of evidence. It’s the single biggest upgrade most accounts can get.
- **Injection.** When a program builds a database command by pasting in whatever a user typed, a user can type something that changes the command itself. The fix is not clever filtering. It’s to send the user’s input separately from the command, so it’s always treated as data. Python’s `sqlite3` does this with `?` placeholders, which your student met in Chapter 5 (SQL-04).

The national standards for high-school computing put it this way: students should “Formulate a solution to a security flaw in a given system.”³⁹ Notice the verb. The point is the fix. They also ask students to “Diagram a network of computing systems, including hardware and software,” which is exactly what NET-02 asks of a home.⁴⁰

The family security rule and the law

Here is the one paragraph to read slowly. Federal law, and state law too, makes it a crime to access a computer system without authorization, and those laws apply to teenagers.³⁷ “I was only testing it” isn’t a defense. “My friend said I could” isn’t permission, because the friend usually doesn’t own the system, and permission that matters is written and comes from the owner. That’s why this

book draws the line plainly: **security practice happens only on the student's own systems and toy apps, under a signed family rule.**

The rule is short. You'll write it together in SEC-01, but here is what it usually says, in the family's own words:

1. I practice security only on computers, accounts, and apps that I own, or that my family owns and has said yes to, and on toy apps I built myself.
2. I don't test, scan, guess passwords on, or try to get into anyone else's system, including a friend's, a school's, a library's, a game's, or a website's, even to help, and even if someone says it's fine.
3. If I find a weakness in something the family owns, I tell my parent, and we fix it together.
4. If I notice a weakness in something we don't own, I don't poke at it. I tell my parent, and we decide together whether and how to report it to the owner.
5. I know that the law applies to me.

Signed by the student and the parent, dated, and kept with the records. You'll see it again in Chapter 9.

Four habits that do most of the work

A government guide for ordinary households puts the same ideas in four plain steps: use strong, unique passwords (or a password manager), turn on two-step login wherever it's offered, recognize phishing, and keep software updated.⁴¹ You don't need a fifth habit for most home accounts. Length, uniqueness, a second step, skepticism about urgent asks, and updates cover nearly everything this chapter teaches.

Wrong answers you should be able to hear

- **"It's only a test."** On whose system? A test on a system you don't own is the thing the law is about. Say: "Whose system is this, and where's the owner's permission in writing?"
- **"If it's online, it's public."** Reachable is not the same as permitted. A shop's back door can be unlocked and still not be yours to open.
- **"HTTPS means it's safe."** HTTPS means private, not honest. Say: "The lock tells you nobody's listening. Who's on the other end?"
- **"I'll change my password every month instead of making it long."** Frequent changes push people toward weak, patterned passwords. Say: "Long and unique beats short and fresh."
- **"Git is my backup."** History is not a second copy. If the drive dies, the history dies with it. Say: "Where's the second place?"
- **"My friend said I could."** Say: "Does your friend own it? Is it in writing? Does the rule cover it?" Usually the answer ends the conversation.

A five-minute parent warm-up

Before the first Hour, do three small things on your own. Look at your router: find the label, notice the cables, count the lights. Open your own email account's security settings and see whether two-

step login is on. Then pick a message in your inbox you're not sure about and ask the four questions from SEC-04: How urgent is it? Who is it really from? Where does the link go? What does it want me to do? You'll be ready to teach the chapter.

How to teach it this week

A worked Hour (illustration)

Here's how one Hour might go for a seventeen-year-old. Picture a student who has already signed the family security rule and finished Chapter 5.

Specify (10 min). The student writes: "A toy login for my own flashcard app. A user types a name and a PIN. It lets them in only if both match a row in my SQLite table. Done when the right PIN gets in, a wrong PIN doesn't, and an injection string doesn't." The parent reads it and asks, "What's an injection string you're going to try?" The student adds: `sam' OR '1'='1` as the third example.

Predict (3 min). "With the unsafe version, that string will get me in. With the parameterized version, it won't."

Build and Test (35 min). The student builds the unsafe login first, on purpose, on their own machine, and watches the injection get in. Then they rewrite it with ? placeholders, rerun the three cases, and watch the third one fail the way it should.

Debug and Log (10 min). Expected/Actual for each case goes in the Debug Log. The lesson line: "Pasting user text into SQL is the bug. Parameters fix it."

Review (5 min). Does the shipped code match the spec? Commit message: "Use parameterized queries for toy login."

Ship and Explain (5 min). "I specified a toy login with three cases. I predicted the injection would get in on the unsafe version. It did. Next I'd change it to lock out after five wrong tries. Help I used: none."

Exact wording you can say

- To open: "Whose system is this — and what are you allowed to do to it?"
- When a friend or a website comes up: "Do we own it? Is permission in writing?"
- When the lock icon appears: "Private connection. Who's on the other end?"
- To fade help: "What's the cheapest strong fix for this account?"
- To close: "What did you protect today, and how will you know the fix held?"

When to stop talking

When your student is reading a packet path, sorting a phishing sample, or walking a threat model, say nothing. Look at the paper or the screen, not at them. Naming the owner and the permission out loud is the skill. If they ask you, ask back: "Whose system?"

In-home — SEC-01 The Family Security Rule

- **Time:** 15 minutes
- **Materials:** one page, two pens, a quiet table
- **Safety and privacy:** first names only if it hangs where visitors can see it
- **The fun:** the student drafts it, and the adults sign it too
- **The skill:** a written line between curiosity and crime, in the family's own words
- **The crumb:** the signed rule, dated, kept with the year's records

How it goes. Read the five points under “The family security rule and the law” together. The student rewrites them in plain language that sounds like this household. Both of you sign and date it. Put a review date on it (the start of the next semester is fine).

15–16: The parent proposes the five points; the student rewords two of them and reads the finished page aloud before signing. **17–18:** The student writes the first draft; the parent edits only for clarity and for the “even to help” line, which teens often soften. **Ready 16:** A sixteen-year-old who already keeps a careful Debug Log can write the first draft.

In-home — SYS-01 The OS at Work

- **Time:** 15 minutes
- **Materials:** index cards labeled Process, Memory, File, Permission, and Scheduler
- **Safety and privacy:** none special
- **The fun:** acting out a computer that is trying to run three programs at once
- **The skill:** processes, memory, files, and permissions as jobs the operating system actually does
- **The crumb:** the role cards, with one sentence on the back of each

How it goes. One person is the Scheduler and hands the processor card to each Process in turn. Memory holds three sticky notes (one per process) and refuses to show one process another's note. File asks Permission before opening. When Permission says no, the process stops. Run it twice: once with every permission granted, once with a file denied.

15–16: Keep three processes and one shared file. **17–18:** Add a fourth process that tries to read another process's memory; the OS refuses.

In-home — NET-01 Packet Relay

- **Time:** 20 minutes
- **Materials:** a short written message cut into four or five slips, labeled 1 of 5, 2 of 5, and so on; two or three “routers” (people or chairs)
- **Safety and privacy:** the message is invented and contains no real names or addresses
- **The fun:** the message arrives out of order, and someone has to put it back
- **The skill:** packets taking different routes and reassembling at the end
- **The crumb:** the reassembled message taped in order

How it goes. The sender hands slips to different routers. Routers walk different paths to the receiver.

The receiver waits until every numbered slip is there, then rebuilds the message. Drop one slip on purpose once, and talk about what TCP would do (ask for it again).

15–16: Five slips, two routers, one deliberate drop. **17–18:** Add a “noisy” router that rearranges two slips; the receiver still rebuilds by number.

In-home — NET-02 Diagram the Home Network

- **Time:** 20 minutes
- **Materials:** paper, pencil, a walk around the house
- **Safety and privacy:** no addresses, passwords, SSIDs, or account names on the page
- **The fun:** discovering how many devices are actually on the network
- **The skill:** router, devices, internet link, and what each does, drawn so a stranger in the family could follow it
- **The crumb:** the diagram (no addresses or passwords)

How it goes. Draw a box for the router, a box for the outside link, and a box for each device that usually connects. Label wired or wireless with an arrow. Put a one-word job under each box (routes, stores photos, prints). Leave passwords and network names off the page entirely.

15–16: Living-room devices only. **17–18:** Add a guest network box if the router has one, and a note about what it’s for.

Band moves

15–16: Sign the rule first. Stay on paper and on the family’s own accounts for two weeks before any screen-heavy try-it. The parent is in the room for every Hour, initials Monday’s plan, and acts as first user when a toy app appears. Restore from the second copy (SYS-03) happens with both of you watching.

17–18: The parent reads the Debug Log the same day and is first user weekly. SEC-03 (break your own toy app) and NET-03 (trace a request) open once the rule is signed and Chapter 5’s parameterized queries are familiar. Ready-16 students may take SYS-02, NET-03, and HW-03 when the paper work is solid.

Practice that actually builds learning

Blocked practice first: three days of SEC-04 Phish or Fine? with new samples until your student names urgency, sender, link, and ask without prompting. Then three days of passphrase-and-two-step work (SEC-05) on family-approved accounts only. **Mixed practice** once both are known: shuffle a phishing sample, a threat-model question, a “whose system?” card, and a restore-check into one pile so each draw asks which habit applies.

Talk box (this chapter)

Opening: “Whose system is this — and what are you allowed to do to it?”

Follow-ups: 1. **Prediction:** “If you send this message in five packets, and two take a longer route, what arrives first — and how do you know when you have the whole thing?” 2. **Evidence:** “What on this page tells you who owns the system?” 3. **Location:** “Where in this login code is the user’s text being pasted into the command?” 4. **Change:** “What’s the cheapest strong fix for this account, and what would it cost us in time?” 5. **Revision:** “If a friend asked you to ‘just check’ their school’s portal, what does our signed rule say?”

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. The question was vague; no wait time; fear of looking wrong in front of a parent (keep feedback about the system, not about them); waiting for you to decide because you have before; no written rule or threat model to compare against; the screen is louder than the question (lower the lid for a moment); the student is ahead of you and bored (ask them to teach the packet relay).

Don’t say / say instead. Don’t say “Don’t ever hack anything”; say “Whose system is this, and where’s the owner’s permission in writing?”

In-home — SYS-02 Simulated Machine Limits

- **Time:** 20 minutes
- **Materials:** a free simulator the family already allows, or a small Python program that allocates a growing list until it slows; paper for the observation note
- **Safety and privacy:** only on the student’s own machine; stop before the computer becomes unusable
- **The fun:** watching a machine hit a wall on purpose
- **The skill:** showing a limit (memory, storage, or speed) and writing what you saw
- **The crumb:** observation note

How it goes. Pick one limit. Predict when the slowdown will show. Run the test. Write what you saw, what you measured, and what you’d change next time. Stop the program before the machine freezes.

17–18: Full try-it. **Ready 16:** Open once SYS-01 and a careful Debug Log are in place.

In-home — SYS-03 Files, Backups, Two Places

- **Time:** 15 minutes
- **Materials:** the student’s portfolio folder, a second drive or cloud the family already uses
- **Safety and privacy:** no portfolio on a shared bot computer; see Family safety basics in **The Computing Hour**
- **The fun:** deleting a practice file on purpose and bringing it back
- **The skill:** folder plan, naming, and a tested restore from the second copy
- **The crumb:** restore check line in the Debug Log

How it goes. Agree on folder names and a date style. Copy today’s work to the second place. Delete one practice file from the first place. Restore it from the second. Write “Restored [filename] from

[second place] on [date]” in the log. Git history is not the second place.

15–16: Parent watches the restore. **17–18:** Student runs the restore alone and shows you the log line the same day.

In-home — SEC-02 Threat Model Your Own Account

- **Time:** 20 minutes
- **Materials:** one family-approved account the student uses, paper, pencil
- **Safety and privacy:** no passwords written down; recovery codes go in the parent’s sealed place
- **The fun:** treating your own account like a small castle
- **The skill:** what you protect, from whom, and the cheapest strong fix
- **The crumb:** threat-model page

How it goes. Four headings: What’s valuable here? Who might want it? How would they try? What’s the cheapest strong fix? For most student accounts the fix is a long unique passphrase, two-step login, and a recovery plan the parent holds. Do the cheap fix before the Hour ends if you can.

15–16: Email or a school portal the family already uses. **17–18:** Add “what happens if my phone is lost?” as a fifth heading.

In-home — SEC-03 Break Your Own Toy App

- **Time:** 30 minutes
- **Materials:** Python, SQLite, the student’s own machine, the signed family security rule
- **Safety and privacy:** only the student’s toy app, only on their machine, under the signed rule; no scanning of anything else
- **The fun:** watching a classic injection get in, then watching it fail after the fix
- **The skill:** finding and fixing an injection flaw with parameterized queries
- **The crumb:** before/after code plus fix note

How it goes. Build a tiny unsafe login that pastes the name and PIN into a SQL string. Try a normal login. Then try a classic injection string as the PIN (for example, ' OR '1'='1). Watch it get in. Rewrite with ? placeholders and a tuple of values. Rerun both cases. Commit the fix with a real message.

A sketch of the unsafe idea (do not ship this pattern outside a toy):

```
# Unsafe on purpose – toy demo only, student's machine only
sql = "SELECT * FROM users WHERE name = '" + name + "' AND pin = '" + pin + "'"
```

And the fix your student already met in Chapter 5:

```
sql = "SELECT * FROM users WHERE name = ? AND pin = ?"
row = db.execute(sql, (name, pin)).fetchone()
```

17–18 only. Not before the rule is signed and SQL-04 is familiar.

In-home — SEC-04 Phish or Fine? (Grown-Up Edition)

- **Time:** 20 minutes
- **Materials:** eight to twelve printed sample messages you've made up (fake prize, fake account warning, "friend" asking for a login code, ordinary relative note, delivery notice, school reminder)
- **Safety and privacy:** paper only; no real brands, people, or live links
- **The fun:** catching a well-made fake before a parent does
- **The skill:** sorting by urgency, sender, link, and ask
- **The crumb:** sorted pile with reasons

How it goes. Sort into "phish," "fine," and "ask a parent." For each phish, name the strongest clue out loud. Then the student writes one new fake that hides three warning signs, and you try to catch it.

15–16: Eight messages, three obvious. **17–18:** Twelve messages, some very good; the student also names what the fake wants (money, a code, a click, a reply).

In-home — SEC-05 Passphrase and Two-Step Check

- **Time:** 20 minutes
- **Materials:** a password manager the family already chose, one family-approved account, paper for a checklist with no secrets on it
- **Safety and privacy:** practice phrases on paper; real passphrases typed only into the manager; recovery codes sealed with the parent
- **The fun:** replacing a short reused password with a long unique one in under five minutes
- **The skill:** long passphrases, a password manager, two-step login turned on
- **The crumb:** checklist (no secrets written)

How it goes. Build a practice passphrase of four or more random words (fifteen or more characters). Compare it with a short patterned password. For one real account, set a unique passphrase in the manager, turn on two-step login, and store the recovery method where the parent can find it. The checklist has boxes, not secrets: unique passphrase, manager holds it, two-step on, recovery stored.

15–16: Parent holds every real secret; student does the practice phrase and reads the checklist aloud.

17–18: Student may hold passphrases for family-approved tools, with the parent's sealed copy of recovery codes.

In-home — HW-01 Inside an Old Computer

- **Time:** 30 minutes
- **Materials:** an unplugged machine an adult is willing to open, a screwdriver, paper
- **Safety and privacy:** unplug first; no power; adult opens the case; no photos of serial numbers posted anywhere
- **The fun:** finding the parts you named on paper
- **The skill:** naming processor, memory, and storage in a real machine
- **The crumb:** labeled sketch

How it goes. On paper first: three boxes labeled processor, memory, storage. Then the adult opens the case. The student points and sketches. Close the case. No brands need naming; the jobs matter more than the stickers.

15–16 and 17–18: Same try-it; the adult always opens the case.

In-home — HW-02 Binary, Text, and Color

- **Time:** 15 minutes
- **Materials:** cards for bits, a short chart of letters and numbers, colored pencils
- **Safety and privacy:** none special
- **The fun:** encoding a short word as numbers, then as bits
- **The skill:** bits to bytes; characters and colors as numbers
- **The crumb:** card chart

How it goes. Show that capital A is 65, and 65 in eight bits is 01000001. Encode a three-letter word. Then pick a color and write it as three numbers (red, green, blue). No chip brands; just the idea that pictures and text are numbers underneath.

In-home — HW-03 What Runs AI (Generic)

- **Time:** 15 minutes
- **Materials:** paper, a short reading about data centers from a library book or a public agency page the parent chose
- **Safety and privacy:** no product pitches; no chip-company names
- **The fun:** estimating how many small sums a “many-at-once” processor does compared with an ordinary one (orders of magnitude, not a quiz)
- **The skill:** processors that do many small sums at once, the data centers they fill, and energy and water as honest open questions
- **The crumb:** one-paragraph note

How it goes. Write three sentences: what these processors are good at, why training big models uses buildings full of them, and one question about energy or water you still have. Leave the question open. Chapter 8 comes back to this.

17–18: Full try-it. **Ready 16:** Open when HW-01 and HW-02 are done.

Out-of-home — OUT-04 Network Spotting

- **Time:** 20 minutes
- **Materials:** a walk through a library, a store, or a downtown block; a small notebook
- **Safety and privacy:** no photographing equipment closets, no touching cables, no joining unknown Wi-Fi
- **The fun:** noticing how much of the city is quietly networked
- **The skill:** finding cables, cabinets, towers, and public Wi-Fi notices
- **The crumb:** spotting list

How it goes. List five things that move data: a cabinet, a dish, a cable run, a public Wi-Fi sign, a cell site. For one of them, write what you think it connects. Don't open panels. Don't join a network you don't trust.

15–16: Parent walks along. **17–18:** Student may go with a sibling or co-op partner and show you the list the same day.

One incorrect example to diagnose (illustration)

Picture a sixteen-year-old whose threat-model page says: “I’ll change my password every thirty days.” The passphrase is eight characters and reused on three sites. Two-step login is off. Ask: “What’s valuable on this account? Who might want it? Is monthly change the cheapest strong fix, or is length-plus-two-step stronger?” Rewrite the page before touching any settings.

For the student

This chapter has one sentence underneath everything else: **you may touch systems you own, and toy apps you built, under a rule you signed. You may not touch anyone else’s.**

That isn’t a lack of trust. It’s how grown professionals work. The interesting practice — breaking a login, tracing a request, threat-modeling an account — all fits inside that line. Crossing the line isn’t clever. It’s a legal problem, and “I was only looking” doesn’t fix it.

Here’s a worked threat model for a made-up email account. What’s valuable: the inbox, the password-reset links it can receive, the contacts. Who might want it: a stranger who sends phishing mail, someone who found an old password in a breach list. How they’d try: a fake “reset your password” message, guessing a short reused password, using a phone that was lost without a lock. Cheapest strong fix: a long unique passphrase in a password manager, two-step login, a recovery code the parent holds, and a habit of checking the sender before clicking. Do those four and you’ve done more than most adults.

If you’re 17 or older, try this fix on a toy app you wrote yourself. Build a tiny login that pastes the user’s text into a SQL string. Then attack it with a string like `' OR '1'='1` as the PIN. Watch it get in. Then change the code so the database gets the values separately:

```
sql = "SELECT * FROM users WHERE name = ? AND pin = ?"
row = db.execute(sql, (name, pin)).fetchone()
# Expected with a wrong PIN: row is None
# Expected with an injection string as PIN: still None
```

Explain the before and after out loud. If you can’t explain a line, it doesn’t stay.

Explain it back. In your own words: what is a packet, what does the operating system do for a running program, and why doesn’t the lock icon mean a site is honest?

Hard for everyone. Thinking like an attacker without becoming one. The skill is imagining how something breaks so you can fix your own things. The temptation is to try the same idea on a system that isn’t yours. The signed rule is how you keep the skill and leave the temptation.

Challenge. Restore one real file from your second copy today. Write the restore line in the Debug Log. If you can't restore it, the backup wasn't a backup.

What to save. The signed family security rule; the home network diagram (no addresses); the threat-model page; the phishing pile with reasons; the before/after note from SEC-03 if you did it; every Debug Log line with minutes.

If it isn't clicking

"It's only a test" about someone else's site

What you see. Interest in a friend's game server, a school's portal, or a public website "just to see."

What it usually means. Permission isn't understood yet, or the rule was signed too fast.

Next move. Re-read the signed rule together out loud. Ask: "Whose system? Where's the owner's permission in writing?" Put the rule back on the table before the next Hour. No SEC-03 until this is settled.

Passwords long but reused

What you see. A twenty-character passphrase used on three sites.

What it usually means. The manager isn't set up, or feels like extra work.

Next move. SEC-05 today, with you present. One account gets a unique passphrase in the manager before the Hour ends. Reuse is the bug; length alone doesn't fix it.

Networks feel like magic

What you see. Shrugging at "how did the page get here?"

What it usually means. No physical picture yet.

Next move. NET-01 Packet Relay on the living-room floor, then OUT-04 Network Spotting the same week. Paper before browser tools.

When to get outside help

A co-op security club, CyberPatriot next season (home-school teams compete in the Open Division — see Chapter 10)⁴², or one session with a working programmer who does security for a living (OUT-03 in Chapter 6). No shame; professionals learn the same way.

Rescue order when they're stuck on a toy app: ask → wait → hint → short demo on your own copy → hand it back.

Tools, including AI

The AI rules for this book live in the AI box in **The Computing Hour**. This chapter doesn't change them. Chat-style help may explain an error after Expected/Actual, with a parent nearby, under the terms. Editor coding assistants stay off through 16, and off for every check at every age.

What differs here: security practice tools are the student's own machine, their own toy apps, a password manager the family chose, and a browser's own developer tools for NET-03. No "hacking suites," no scanning services aimed at other people's systems, no tutorials that start with someone else's website. Check the current terms on any account before you turn on two-step login or a manager.

Hardware talk stays generic: processors, memory, storage, and processors that do huge numbers of small sums at once. No chip brands.

What "done enough" looks like

Mark each item **seen** / **growing** / **not yet**. Place by skill, not birthday.

- Signed and follows the **family security rule** (SEC-01).
- Diagrams the home network with no addresses or passwords (NET-02).
- Explains packets and reassembly in plain words (NET-01).
- Explains what DNS does in one or two sentences (NET-03 when ready).
- Threat-models one of their own accounts and applies a cheap strong fix (SEC-02, SEC-05).
- Sorts phishing samples by urgency, sender, link, and ask (SEC-04).
- Keeps files in **two places** and has tested a restore (SYS-03).
- Names processor, memory, and storage on a real or sketched machine (HW-01, HW-02).
- At 17–18: finds and fixes an injection flaw in their own toy app (SEC-03).
- AS-02 Debug on the Spot seen at least once this month on a planted bug in their own code.

Move on when most items are **seen**. **Growing** is fine. **Not yet** means another week, not a failure. CyberPatriot and other competitions wait for Chapter 10; they are never a transcript title.

Chapter 8

Working With AI: How It Learns, Where It's Wrong, and Whose Work It Is



Why this matters

Your student is old enough to use AI tools under the services' terms. That is not the same as being ready to hand those tools their homework, their voice, or their judgment. This chapter teaches how

models learn from examples, where fluent answers go wrong, how to check a claim before trusting it, and how to keep ownership honest when a tool helped. It also carries the book's one calm box about image-based blackmail, because real and generated images are now the sharpest online risk for this age.

The focus line still holds. **Specify it before you build it; predict before you run; debug before you rewrite; explain it before you call it yours.** With AI, “explain it before you call it yours” is the whole game. If your student can't walk a parent through a function line by line, it isn't their work yet — even if a tool drafted it in three seconds.

The national standards for high-school computing put two demands side by side: evaluate training data for source, quality, bias, and privacy, and evaluate AI-generated code for accuracy and fit with the requirements.⁴³ Computer science, on this view, is not “using generative AI tools”; it is what powers them.⁴⁴ Your student is here to understand the machine, not to outsource the thinking.

One more thing this chapter will not do: panic, or sell. No doom about jobs. No hype about magic in a box. A tool that predicts the next word is useful, limited, and made by people who chose its training data. Treat it that way.

For the parent: understand it yourself

The plain picture

An AI model in the sense this chapter means is a program that learned patterns from a large pile of examples — **training data** — and then makes predictions on new inputs. A spam filter learned from millions of labeled messages. A chatbot learned from huge amounts of text and predicts likely next words. A image generator learned from many pictures and captions. None of these systems “understand” the way a person does. They are very good at sounding sure.

Three kinds of tools get mixed up in conversation:

- **Rule-based code** — you wrote the steps; the same input gives the same output.
- **A trained model** — the program found patterns in labeled examples (this email is spam / not spam).
- **A generative model** — the program produces new text, code, or images that look like its training data.

Sometimes the right answer is **no AI at all**. Sorting a list of names, adding a receipt, or writing a closed-book check are usually rule-based or human jobs.

The precise picture

- **Training versus test data.** You train on one pile and check on a held-back pile the model has never seen. If you test on the training pile, you will think the model is better than it is (AI-01, AI-02).
- **Deterministic versus probabilistic.** Ordinary code is deterministic: same input, same output. Many generative tools are probabilistic: the same prompt can give different answers

(ALG-04, AI-03).⁴⁵

- **Fluent is not true.** Smooth sentences are not evidence. Claims need primary sources opened and read (AI-04).⁴³
- **Who is missing.** Training data reflects who showed up in it. If a group is thin in the examples, errors often fall on that group (DATA-05).⁴³
- **Hardware, generically.** Training big models uses processors that do huge numbers of small sums at the same time, packed into large data centers. Energy and water use are honest open questions, not settled talking points (HW-03). No chip brands belong in this conversation.
- **Whose work.** Anything a tool generates is **made by a tool**. The student’s work is what they can specify, predict, test, and explain. The **Whose Work Is It?** card (AI-06) and the exit line “Help I used: ...” go with every project.

When AI is a good tool, and when it isn’t

Use a generative tool when you need a draft to react to, a second explanation of an error you already diagnosed, or practice questions with answers hidden until you’ve tried. Use a trained model when the job is sorting or scoring on a kind of input you’ve actually measured. Use ordinary code when the steps are known and the same input should always give the same output. Use no AI when the point of the hour is to find out what the student can do alone — every AS check, every Cold Trace, every Build From a Blank.

Picture a sixteen-year-old who asks a chat tool for “a Python function that grades quizzes” before writing a spec. The tool returns something fluent. The student pastes it, runs one happy-path example, and calls the Hour done. What’s missing: a done line, an empty-answer case, a Whose Work card, and any sign the student could rewrite the function tomorrow with the screen dark. The fix is not a lecture about cheating. It’s SPEC-01 again, then AS-03 Explain Your Code before the next feature.

The editor-assistant rule, in one short paragraph

The full AI rules live in the AI box in **The Computing Hour**. Here is the piece this chapter uses. An AI coding assistant inside the editor stays **off through age 16**. At 17, it may be turned on for **one named project**, after the student passes a closed-book **AS-04 Build From a Blank**; the README and Whose Work card say so, and every kept suggestion is explained line by line (AI-05). The assistant stays **off for all checks** (AS-01 through AS-07) at every age. Chat-style help that explains an error, after Expected/Actual and a guess, is allowed at 15–16 with a parent nearby, under the terms.

Age terms, in one sentence

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 18.⁷ All students in this book are over the thirteen line; the eighteen line still matters for one major service. Check the current terms before any account is opened.

AP Computer Science Principles, briefly

If your student sits AP Computer Science Principles, generative AI may be used as a supplementary resource, but the student must understand, explain, and acknowledge AI-created code.⁴⁶ The course is being redesigned for 2027–28 with AI built more deeply into the framework — so this book lists **no unit outline**, which would go stale. Exact dates for exams and ordering live in the notes and in Chapter 9.

Wrong answers you should be able to hear

- **“It cited a source, so it’s true.”** Citations can be invented. Say: “Open the source. What does it actually say?”
- **“I changed some words, so it’s mine.”** Paraphrase without understanding is still the tool’s work. Say: “Walk me through it with the screen dark.”
- **“It’s just for ideas.”** Then the idea is disclosed on the Whose Work card. Say: “Write ‘Help I used: ...’ before we go on.”
- **“Everyone uses it on tests.”** Our checks are AI-free by design. Say: “This check is how we know what you can do alone.”
- **“The detector said it was human.”** Detectors are unreliable, and we don’t use them. Say: “Explain it to me” is the check.

A five-minute parent warm-up

Pick one short claim a chatbot makes about a topic you know (a recipe, a historical date, a local fact). Ask it for a source. Open the source. Notice whether the claim survives. That itch — open it before you trust it — is the whole chapter in miniature.

How to teach it this week

A worked Hour (illustration)

Here’s how one Hour might go for a seventeen-year-old who has already passed AS-04.

Specify (8 min). “Flashcard app, deck of twenty cards, quiz mode, score at the end. Done when a stranger can add a card and take a quiz without help. Editor assistant allowed on this named project only; README will say so.”

Predict (3 min). “If I accept the assistant’s regex for splitting cards, I’ll regret it unless I can explain every part.”

Build and Test (40 min). Two assistant suggestions stay after line-by-line explanation. One regex suggestion is deleted because the student can’t explain it. Tests cover empty deck and a one-card deck.

Debug and Log (10 min). Expected/Actual for the empty-deck crash; fix logged.

Review (5 min). Whose Work card lists the two kept suggestions and the deleted one. Commit: “Quiz mode with empty-deck guard.”

Ship and Explain (5 min). “I specified a flashcard quiz with two edge cases. I predicted the regex would be a trap. It was — I deleted it. Next I’d change the scorer to weight recent misses. Help I used: editor assistant on two functions, both explained in the Whose Work card.”

Exact wording you can say

- To open: “What did it learn from, where might it be wrong — and whose work is this?”
- When output sounds perfect: “Fluent isn’t true. What would you check?”
- When code appears from a tool: “Explain every line, or it doesn’t stay.”
- To fade help: “What’s on your Whose Work card so far?”
- To close: “What did you keep, what did you delete, and why?”

When to stop talking

When your student is checking a claim against a primary source, or walking a function line by line, say nothing. Look at the source or the code, not at them. Ownership is the skill. If they ask you, ask back: “What does this line do to the input?”

In-home — AI-01 How a Model Learns

- **Time:** 20 minutes
- **Materials:** index cards with simple drawings or words; a secret sorting rule only the parent knows
- **Safety and privacy:** no photos of real people; no personal data
- **The fun:** guessing a rule from examples, then failing on a held-back pile
- **The skill:** a human classifier; training versus test sets
- **The crumb:** rule guessed vs real rule

How it goes. Parent sorts a training pile by a secret rule (for example, “has a curve” versus “only straight lines”). Student guesses the rule. Then a test pile appears. Count errors. Talk about which kinds of cards were missing from training.

15–16: Twelve training cards, six test cards. **17–18:** Add a second round where the training pile is lopsided on purpose; watch which errors show up.

In-home — AI-02 Train and Test a Small Model

- **Time:** 30 minutes
- **Materials:** a simple classifier tool or a short Python sketch the family already allows; a dataset the student built from non-personal data (for example, sorting library book genres from titles the student typed)
- **Safety and privacy:** no names, faces, messages, or household private data; nothing uploaded that identifies a person

- **The fun:** watching accuracy drop when the test set is honestly held back
- **The skill:** training a simple classifier; measuring errors
- **The crumb:** accuracy note with errors listed

How it goes. Split data into train and test before touching the tool. Train. Test. List the mistakes. One sentence: who or what would be hurt if those mistakes were real decisions.

15–16: Parent sits beside every upload and every run. **17–18:** Student runs it; parent reads the error list the same day.

In-home — AI-03 Next-Word Table

- **Time:** 25 minutes
- **Materials:** paper; optional short Python program
- **Safety and privacy:** invented sentences only; label every generated line “made by a tool”
- **The fun:** a tiny table that “talks” in a clumsy way
- **The skill:** next-word prediction as probabilistic pattern-matching
- **The crumb:** table and output labeled made by a tool

How it goes. From a short paragraph, build a table of each word and the words that followed it. Start from a seed word and pick the next word from the table (dice if there are several). Read the result aloud. Optional Python version:

```
import random
text = "the cat sat on the mat and the cat ran"
words = text.split()
table = {}
for a, b in zip(words, words[1:]):
    table.setdefault(a, []).append(b)
word = "the"
for _ in range(6):
    word = random.choice(table.get(word, ["the"]))
    print(word, end=" ")
# Output varies; label it made by a tool
```

15–16: Paper first; Python optional. **17–18:** Paper and Python; compare two runs of the same seed.

In-home — AI-06 Whose Work Is It? Card

- **Time:** 10 minutes
- **Materials:** a half-page card (or a section in the README)
- **Safety and privacy:** no account passwords on the card
- **The fun:** honesty that takes less than ten minutes
- **The skill:** disclosing what a tool did and what the student did
- **The crumb:** the card

How it goes. Four lines: What I specified. What I wrote. What a tool drafted. What I can explain cold. File it with every project from this week on. “Help I used: none” is a complete answer when

it's true.

15–16 and 17–18: Same card; at 17, name the editor-assistant project explicitly when the rule applies.

Band moves

15–16: Chat-style help only after Expected/Actual, parent nearby, under the terms. Editor assistants off. Paper classifiers and claim-checking before any account. Stop, Save, Tell rehearsed this month (SAFE-02).

17–18: AS-04 Build From a Blank is the gate before any editor assistant. One named project only. AI-05 Explain Every Line on every kept suggestion. At 18, offer SAFE-05 The Family AI Agreement as adults in one household.

Ready 16: ALG-04 Coin Flip or Rule? and HW-03 when the paper work is solid; editor assistants still wait for 17 and AS-04.

Practice that actually builds learning

Blocked practice: three days of AI-04 Fluent Is Not True with new claims until opening a primary source is automatic. Then three Whose Work cards on ordinary Chapters 2–6 projects, even when no AI was used (“Help I used: none”). **Mixed practice:** shuffle a claim-check, a tool-choice card (AI-07), a Whose Work audit, and a Stop, Save, Tell rehearsal into one pile.

Talk box (this chapter)

Opening: “What did it learn from, where might it be wrong — and whose work is this?”

Follow-ups: 1. **Prediction:** “If we hold back these six cards from training, where will the model stumble?” 2. **Evidence:** “What primary source did you open, and what did it say in its own words?” 3. **Location:** “Which lines in this function did a tool draft — point to them?” 4. **Change:** “What would you delete if you had to explain this cold tomorrow?” 5. **Revision:** “If the Whose Work card is blank but the style suddenly shifted, what do we rewrite?”

How to wait. Count to five in your head. When the question is about code on the screen, let them read as long as they need. Look at the code, not at the student.

What a stuck silence usually means. The question was vague; no wait time; fear of looking wrong (keep feedback about the claim or the code); waiting for you to bless a pasted answer because you have before; no written prediction or Whose Work card to compare against; the screen is louder than the question (lower the lid); the student is ahead and bored (ask them to teach next-word prediction on paper).

Don't say / say instead. Don't say “AI is cheating”; say “If you can't explain it, it isn't yours yet — show me on the Whose Work card.”

In-home — AI-04 Fluent Is Not True

- **Time:** 15 minutes
- **Materials:** three short generated claims on paper; access to primary sources the parent approves
- **Safety and privacy:** no pasting private family facts into a chatbot to “check” them
- **The fun:** catching a confident falsehood
- **The skill:** checking generated claims against primaries
- **The crumb:** claim/check table

How it goes. For each claim: write it, name a primary source, open the source, mark true / false / unclear. Unclear is an honest win.

15–16: Parent chooses the three claims. **17–18:** Student gathers three claims from a tool under the terms, then checks them.

In-home — AI-05 Explain Every Line

- **Time:** 20 minutes
- **Materials:** one AI-suggested function, printed or on screen; pencil
- **Safety and privacy:** no private data in the prompt; under the terms
- **The fun:** sending a clever-looking line to the recycle bin
- **The skill:** explaining an AI-suggested function line by line before any of it is used
- **The crumb:** annotated code

How it goes. Cover the comments if the tool wrote those too. For each line, the student says what it does to the input. Any line they can’t explain is deleted. Re-run tests after deletions.

17–18 only (and only after AS-04 if this is the named editor-assistant project).

In-home — AI-07 Pick the Right Tool

- **Time:** 15 minutes
- **Materials:** task cards (“sort names,” “spot spam,” “write a birthday poem,” “add up a receipt,” “detect a face in a photo,” “decide a grade”)
- **Safety and privacy:** none special on paper
- **The fun:** arguing for “no AI” and winning
- **The skill:** sorting tasks into rule-based code, trained model, generative model, or no AI
- **The crumb:** sorted cards with reasons

How it goes. One sentence of reasoning per card. “Add up a receipt” is usually rule-based code. “Write a birthday poem” can be generative — and still needs a Whose Work card if any line is kept. “Decide a grade” is no AI in this house.

In-home — SAFE-01 Age Terms Map

- **Time:** 15 minutes

- **Materials:** the list of tools the family actually uses; paper
- **Safety and privacy:** parent opens the terms pages; student records the numbers
- **The fun:** finding the one service that says 18
- **The skill:** looking up current age terms
- **The crumb:** table with check dates

How it goes. Columns: tool, minimum age, parent permission needed?, date checked. Body rule of thumb stays the same; this table is how you keep it honest. Check the current terms.

In-home — SAFE-02 Stop, Save, Tell

- **Time:** 10 minutes
- **Materials:** a few paper “messages” you invent: mean, pushy, nosy, threatening. Keep them mild.
- **Safety and privacy:** pretend messages only
- **The fun:** swapping roles until it’s boring
- **The skill:** a rehearsed response that doesn’t depend on judging how bad a message is
- **The crumb:** script card

How it goes. Hand over a paper message. Student acts the three steps: hands off, save evidence (a pretend screenshot), tell a trusted adult. You say, “You are not in trouble.” Swap. Ordinary is the goal. Write two backup adults on the card.

In-home — SAFE-03 Footprint Audit

- **Time:** 20 minutes
- **Materials:** a search for the student’s own name, with a parent; a private list
- **Safety and privacy:** parent present; nothing new posted during the audit; fix settings rather than arguing with strangers
- **The fun:** finding an old account nobody remembered
- **The skill:** listing what is public; fixing settings
- **The crumb:** private list

How it goes. Search the name and one username. List what a stranger can see. Change settings or delete what shouldn’t be public. First names only in anything that stays.

In-home — SAFE-04 Privacy Settings Tour

- **Time:** 20 minutes
- **Materials:** one AI service the family already uses under the terms
- **Safety and privacy:** parent at the table; turn off training and memory where you can
- **The fun:** finding a switch nobody knew was on
- **The skill:** walking through memory, training, history, and image settings together
- **The crumb:** settings checklist

How it goes. Checklist in ordinary words: training off if offered; memory off if offered; history reviewed; image generation rules understood; quiet hours if offered. Product names matter less

than the switches.

In-home — SAFE-05 The Family AI Agreement

- **Time:** 20 minutes
- **Materials:** paper, everyone eighteen or older in the household who wants a say
- **Safety and privacy:** first names only if it hangs in a shared space
- **The fun:** writing rules as adults, not as a lecture
- **The skill:** drafting the family's AI agreement as an adult household member
- **The crumb:** signed agreement

How it goes. At 18 only. What we use AI for; what we never paste into it; how we label work a tool touched; how we handle images of people; how we revisit the agreement. The age-banded rules in the AI box give way to this household agreement.

In-home — SAFE-06 Deepfake and Image Safety

- **Time:** 15 minutes
- **Materials:** a calm help card you write together
- **Safety and privacy:** no examples of real harm; no graphic detail; no searching for harmful images
- **The fun:** none needed — clarity is enough
- **The skill:** what generated images can do; never upload others' photos; where to get help
- **The crumb:** help card (Take It Down, report steps)

How it goes. Write four lines on the card: never upload anyone else's photo to an AI image tool; never make an image of a real person without consent; if it happens to you, Stop, Save, Tell; use NCMEC's Take It Down for images of anyone under 18, and report. Read the sextortion box in this chapter out loud once as a family.

In-home — DATA-05 Who Is Missing?

- **Time:** 15 minutes
- **Materials:** a small dataset the student already has, or a public table with no personal rows
- **Safety and privacy:** no scraping of classmates or social accounts
- **The fun:** finding the hole in a table that looked complete
- **The skill:** sampling and coverage
- **The crumb:** two-sentence note

How it goes. Ask: who isn't in this data? What decisions would go wrong because of that? Tie the answer to AI-01's held-back cards.

In-home — ALG-04 Coin Flip or Rule?

- **Time:** 15 minutes

- **Materials:** cards describing procedures (a recipe, a random playlist, a dice game, a sorting rule)
- **Safety and privacy:** none special
- **The fun:** running the dice game three times to prove it changes
- **The skill:** deterministic versus probabilistic
- **The crumb:** two-column sort of examples

How it goes. Sort into “same input, same output” and “might change.” Generative chatbots go in the second column.

17–18: Full try-it. **Ready 16:** Open when AI-03 is done.

In-home — AS-04 Build From a Blank

- **Time:** 45 minutes
- **Materials:** a short spec the student hasn’t seen; empty file; no tutorial, no AI, no notes
- **Safety and privacy:** AI off; phone away if it has a chatbot
- **The fun:** shipping something small under real constraints
- **The skill:** a small program from an empty file; the gate for the editor-assistant project at 17
- **The crumb:** dated file

How it goes. Parent sets a timer. Spec on paper. Empty editor. Passing tests or an honest Debug Log both count as evidence. This check is AI-free at every age. At 17, a pass is what opens the one named editor-assistant project.

Out-of-home — OUT-05 Where Is AI Deciding?

- **Time:** 20 minutes
- **Materials:** a walk through a bank, store, or library; a small notebook
- **Safety and privacy:** one polite staff question max; no filming people; no arguing with workers
- **The fun:** noticing decisions that used to be human
- **The skill:** spotting automated decisions; asking one calm question
- **The crumb:** list with one answer

How it goes. List three places a system might be deciding something (fraud flag, recommendation shelf, chat bot on a library site). If a staff member has time, ask one question: “Is a person reviewing that, or is it automated?” Write the answer without a last name.

One incorrect example to diagnose (illustration)

Picture a sixteen-year-old whose project folder has a polished hundred-line module and a Whose Work card that says “Help I used: none.” The student can’t explain a regular expression on line 40 with the screen dark. Ask: “Which lines can you teach me cold?” Delete or rewrite the rest. Fill the Whose Work card honestly. The grade for the week is the honesty, not the polish.

Sextortion and image blackmail (read together)

One online danger is worth naming plainly at this age, because knowing about it ahead of time is most of the protection. It's sometimes called **sextortion**: someone online — often pretending to be another teen — builds a quick friendship, pressures a young person into sending a private picture, and then threatens to share it unless the young person pays or sends more. The same pattern now includes AI-generated images made to look like someone without their consent.⁴⁷

Here is the calm version of what to do. It is not your fault. Do not pay. **Stop** replying. **Save** the messages and any images as evidence (screenshots, export, or a second device). **Tell** a trusted adult right away — a parent, another adult in the family, or a counselor. Report the account through the platform, and report to NCMEC's CyberTipline. For images of anyone under 18, use NCMEC's Take It Down. Under federal law, most covered platforms must remove nonconsensual intimate images — real or AI-generated — within 48 hours of a valid request.⁴⁸ Sharing such images is itself a crime. You are not alone, and asking for help does not cost you your device or your parent's trust. Rehearse Stop, Save, Tell (SAFE-02) until it is boring. Boring is the goal.

For the student

You are old enough to use these tools. You are also old enough to lose yourself in them if you aren't careful — not in a dramatic way, just by letting a fluent paragraph stand in for a thought you never finished.

Here is how a model learns, in four sentences. It sees many examples. It adjusts millions of little knobs so its predictions match those examples better. It is checked on examples it didn't train on. It still makes mistakes, and the mistakes often fall on whoever was thinly represented in the training pile.

A worked Whose Work card for a made-up project might look like this:

- **What I specified:** a tip calculator with three examples and a done line.
- **What I wrote:** the function, the tests, the README.
- **What a tool drafted:** a rounding helper I asked a chat tool to explain, then retyped myself.
- **What I can explain cold:** every line of the function and the tests.
- **Help I used:** chat tool explained banker's rounding; I rewrote it in my own words.

If you're 17 or older and you've passed Build From a Blank (AS-04), you may turn an editor assistant on for **one named project**. Name it in the README. Explain every kept line. Delete what you can't explain. The assistant stays off for every check.

Try today. Take three confident claims from any generative tool you're allowed to use. Open a primary source for each. Fill a claim/check table. Unclear counts.

Explain it back. What is the difference between training data and test data? Why can the same prompt give two different answers? What does "made by a tool" mean on a Whose Work card?

Hard for everyone. Not reaching for a chatbot at the first red text. Write Expected/Actual and a guess first, every time. The tool is allowed to explain after that, under the terms, with a parent

nearby at 15–16. The order is the discipline.

One more hard part: liking the tool's voice more than your own. If every paragraph starts to sound like the same smooth assistant, stop and write a page with the network off. Your voice is part of what colleges, employers, and your future self are trying to hear.

Challenge. Build the next-word table in Python from a short paragraph you wrote. Run it twice. Label both outputs made by a tool. File them with a Whose Work card.

What to save. Every Whose Work card; the claim/check table; the age-terms map with dates; the Stop, Save, Tell card; the help card from SAFE-06; AS-04's dated file when you sit it; Debug Log minutes.

If it isn't clicking

Treats output as fact

What you see. Pasting a chatbot paragraph into a report without a source.

What it usually means. Fluency mistaken for truth.

Next move. AI-04 with three claims and primaries, today. No project work until one claim has been checked cold.

Can't explain pasted code

What you see. A sudden jump in style; shrugs when asked about a line.

What it usually means. Ownership skipped.

Next move. Whose Work card; explain or delete. Screen dark for the teach-back. AS-03 Explain Your Code before the next feature.

Reaches for AI at first red text

What you see. Chat open before Expected/Actual.

What it usually means. Habit, not judgment.

Next move. Box the chat app for one week of Hours. Expected/Actual and a guess first, every time. Rescue order: ask → wait → hint → short demo on your own copy → hand it back.

When to get outside help

A co-op discussion group, a dual-enrollment course that treats AI as an object of study, or OUT-03 with a working programmer who can talk about what their team allows. For safety scares, you are the first help; NCMEC and platform reporting are next. No shame in any of it.

Tools, including AI

Point back to the AI box in **The Computing Hour** for the full allow-and-ban list. This chapter is where those rules get practiced.

What differs here: classifiers on non-personal data; next-word tables; claim-checking; privacy-setting tours; the optional editor assistant at 17 on one named project after AS-04. Turn off training and memory where you can. No private messages, photos, names, or addresses into chatbots. No photos of family or friends into AI image tools. No companion or role-play apps. No AI “detectors” — “explain it to me” is the check. Check the current terms.

CodeAI (formerly Code.org) offers age-banded tools with their own rules; if you use them, the thirteen-plus lines still apply, and teacher or parent settings matter.⁷

What “done enough” looks like

Mark each item **seen** / **growing** / **not yet**. Place by skill, not birthday.

- Explains training versus test data in plain words (AI-01, AI-02).
- Checks three generated claims against primary sources (AI-04).
- Fills a Whose Work card on every project (AI-06); “Help I used: ...” on every exit.
- Sorts tasks into rule-based, trained, generative, or no AI (AI-07).
- Knows Stop, Save, Tell and has a help card for image abuse (SAFE-02, SAFE-06).
- Has an age-terms table with check dates for tools the family uses (SAFE-01).
- Has walked privacy settings and turned off training/memory where possible (SAFE-04).
- At 17: has passed **AS-04 Build From a Blank** before any editor assistant; explains every kept suggestion (AI-05).
- At 18: has been offered **SAFE-05 The Family AI Agreement**.
- AS-03 Explain Your Code seen on a project that touched a tool.

Move on when most items are **seen**. **Growing** is fine. **Not yet** means another week, not a failure.

Chapter 9

Credit and the Transcript: Records, Portfolio, AP, and Dual Enrollment



Why records matter now

Up to this point the book has been about learning to compute. This chapter is about making that learning legible to a stranger — a college admissions reader, a registrar, a scholarship committee,

or the adult your student will be in five years. Homeschool records are not a formality at 15–18. They are how a year of specs, Debug Logs, tests, and projects becomes a credit, a transcript line, and a portfolio a student can defend.

You do not need a school’s software to do this well. You need honest hours, titles a stranger can read, course descriptions in plain sentences, a portfolio with Whose Work cards, and a small set of AI-free checks that show what the student can do alone. You’re in good company: in the most recent federal survey that asked, from 2016, 17 percent of homeschooled students working at a high-school level had computer science among their subjects.⁴⁹ The law in your state decides what a diploma requires; this book does not. Read your state’s current law. This isn’t legal advice.

What the law usually asks

States differ. Some require a notice of intent, some require certain subjects, some require almost nothing on paper until a transcript is requested. What nearly every next step asks for is the same shape: a list of courses with titles, grades or narrative evaluations, credits, and a way to see the work underneath. Dual enrollment adds college transcripts the college controls. AP adds score reports the College Board controls. Your family records sit beside those, not inside them.

Homeschool family records are yours. They are not the same thing as a school’s FERPA education records. Keep copies of everything you care about.⁵⁰

Hours and credit

A common guideline for a full academic high-school credit is about **150 hours** of work; some families use **120 hours** as the floor for an elective credit; half credits are half those amounts (the same organization gives 60 hours for a half-credit elective).⁸ Those numbers come from a well-known homeschool organization’s advice, and they are a **guideline, not law**. Your state and your family decide.

For a year built on this book — programming, testing, data, web, systems, security, and AI — treat the year as a full academic course and aim near **150 hours for 1.0 credit**. If a semester was lighter and mostly exploratory, 120 hours for an elective credit can be honest. The Debug Log already carries date and minutes; PLAN-02 The Hour Audit totals a month; the year summary totals the months.

Count Computing Hours, named try-its, reading that feeds a project, Demo Day, and portfolio defense. Do not count idle browsing. Do not invent hours at the end of May.

Transcript titles a stranger can read

Use ordinary titles. Never a book name, a brand, a tool, or a competition.

Title	When it fits
Computer Science	Default for a full year built on this book
Computer Programming	A year or semester mostly Python and Java projects
Web Development	A web-heavy semester or year; the description may note it is comparable to the state category “Web Page Design” ³⁶
Cybersecurity	Only after a real security year under the family security rule
Data Science	Optional, if Chapter 5 dominated a semester

Put “AP” in a course title only after the course has passed the College Board’s AP Course Audit. Otherwise list an AP exam score under test scores, not in the course title. This book issues no certificates. Program certificates from other organizations are artifacts for the portfolio, not transcript titles.

Course descriptions

Three to five sentences, written for a stranger. What was studied, how it was assessed, about how many hours, and the major projects. No adjectives that try to impress. Here are labeled illustrations — not real records.

Illustration A. *Computer Science* — 1.0 credit — Grade 10. “A full-year course in programming and computer science: Python for files, functions, and data structures; testing and debugging; object-oriented programming in Python and Java; data analysis and SQL databases; web pages, the command line, and version control; operating systems, networks, and security practiced on the student’s own systems; and how AI systems learn and where they fail. Assessed by monthly unaided code traces, live debugging, build-from-blank checks, and a portfolio defense. About 155 logged hours.”

Illustration B. *Computer Programming* — 1.0 credit — Grade 11. “Object-oriented programming in Python and Java, including classes, arrays and lists, and two-dimensional arrays; written specifications and automated tests for every project. Student sat the AP Computer Science A exam (score listed under tests).”

Illustration C. *Web Development* — 0.5 credit — Grade 12. “HTML, CSS, and JavaScript; accessibility testing; Git version control; one published project chosen with parent approval. Comparable to the state course category ‘Web Page Design.’ About 80 logged hours.”

Illustration D. Dual enrollment line: “Introduction to Programming (college course, 3 credit hours), [college name], fall semester — grade reported by the college.” The family keeps the syllabus and the grade report.

The portfolio

The portfolio is a small set of projects a stranger could open, run, and understand, plus the paper that proves ownership.

Each project folder (or binder section) holds:

- the one-page spec (SPEC-01 and friends)
- the tests and a note about what they prove
- the Debug Log pages for that project
- the Git history or a changelog a human can read (PROJ-02, GIT-05)
- a README a stranger can use (GIT-05)
- the **Whose Work Is It? card** (AI-06), even when the answer is “Help I used: none”
- for shipped work, the parent’s publishing decision (PROJ-04)

Pick three to five projects for a year, not fifteen. Depth beats volume. PLAN-03 Scope It Down is how an oversized idea becomes a shippable one. SPEC-05 Spec Diff (17–18) is how you record what changed between the spec and what shipped.

Optional talk line when picking pieces

“Which piece shows what you can really do — and could you defend every choice in it?”

The check set

These live in every teaching chapter’s “done enough” list. Chapter 9 gathers them. All are **AI-free**. Mark each big idea **seen** / **growing** / **not yet**.

Code	Name	What it shows
AS-01	Cold Trace	Unaided: trace a program and predict its output (monthly)
AS-02	Debug on the Spot	Fix a planted bug while explaining aloud
AS-03	Explain Your Code	Teach-back of the student’s own program line by line
AS-04	Build From a Blank	A small spec from an empty file; no tutorial, no AI; the gate for the editor-assistant rule at 17
AS-05	Whose Work Audit	Checking Whose Work cards against the project and its history
AS-06	Spec-to-Test Check	Unaided: write tests for a stranger’s spec

Code	Name	What it shows
AS-07	Portfolio Defense	Presenting one project; answering three questions about choices and trade-offs (17–18, ready 16)

A year summary that says “all seen” on AS-01 through AS-04, with AS-07 growing, is a strong record. A year summary that only lists tools used is not.

Projects, demo days, and shipping

PROJ-01 Capstone Spec and Plan opens a larger project: problem, user, features, tests, dated schedule. **PROJ-03 Demo Day** shows the work to family or co-op users and collects feedback without fixing on the spot — the parent can still be first user. **PROJ-04 Ship It** (17–18) publishes only with the parent’s decision, with no names, faces, school, or town in the public materials.

OUT-03 Ask a Working Programmer and OUT-06 Visit a College Class or Open House belong here as artifacts too: interview notes (first names only), and questions asked before any dual-enrollment decision.

AP for homeschoolers

Homeschoolers do not order AP Exams as private individuals through a parent account in the usual school workflow. An authorized school’s **AP coordinator** adds the student and orders the exam.⁵¹ Find a school through the AP Course Ledger, contact the coordinator by **early fall**, and know that the **final ordering deadline falls in mid-November**. Exams are given in the **first two weeks of May**. Exact dates for the current season live in the notes.²⁴

AP Computer Science A (Java) fits a year that leaned on Chapter 4. The exam includes multiple-choice questions and four free-response question types: methods and control structures, class design, data analysis with ArrayList, and 2D arrays.²³

AP Computer Science Principles is being redesigned for 2027–28, with generative AI woven through the framework. This book lists **no unit outline**. Students who use generative AI as a supplement must still understand, explain, and acknowledge AI-created code (see Chapter 8).

Putting “AP” on a home course title requires the AP Course Audit. Sitting the exam without that audit still produces a score you can list under tests.

Dual enrollment and who sees the grades

Once a student attends a postsecondary institution — at any age — FERPA rights over the **college’s** records belong to the student. Parents keep rights over high-school records while the student is

under 18. The college **may** share grades with parents of a tax-dependent student, but it is not required to.⁵⁰

Before the first college class, agree as a family how grades will be shared. The student can sign the college's release form if everyone wants the parent to see the portal. The family keeps its own copies of every syllabus and grade report either way. Eligibility, cost, and age limits vary by state and college — ask the college's dual-enrollment office. OUT-06 is worth doing before anyone signs.

Illustration (dual enrollment). A line on the family transcript points to the college course and says the grade is reported by the college. The folder holds the syllabus, the grade report, and the signed release if one exists.

Competitions as artifacts

Competitions are evidence of work, not course titles. Check the current dates.

- **CyberPatriot** — next season; home-school teams compete in the Open Division; an adult coach is required; registration usually closes in early fall.
- **Bebras** — usually November; free in many countries; Juniors and Seniors bands cover 14–18.
- **ACSL, USACO, Congressional App Challenge, CyLab Security Academy** (formerly picoCTF) — useful practice and artifacts; never transcript titles.

File a score report or a project link in the portfolio. Do not rename a course “CyberPatriot” or “USACO.”

Work, internships, and hours

If a paid coding or tech job comes along, federal rules don't limit hours at 16 and 17, but your state's rules may, and some jobs are off-limits under 18.⁵² Keep the family security rule in force: work projects on employer systems follow the employer's written rules, which is a different permission than practicing on a friend's laptop.

Where records live

A family drive plus one backup — **two places** (SYS-03). Not inside a chatbot. Not only in Git on a single laptop. If you use Grok Bot, remember that all of your bots share one computer; keep the portfolio on a family drive.⁶ BUG-05 The Debug Log, PLAN-02 The Hour Audit, and GIT-05 README a Stranger Can Use are the spine. Print or export a year summary once, dated.

A year of records at a glance

Here is one honest shape for a school year. Adjust to your calendar.

When	What goes in the record
Every Hour	Debug Log line with date, minutes, Expected/Actual when something broke, “Help I used: ...”
Every Monday	PLAN-01 Monday Plan, Initialed (15–16) or reviewed (17–18)
Every project	Spec, tests, Whose Work card, README
Every month	PLAN-02 Hour Audit; one AS-01 Cold Trace
Every big idea	AS-02, AS-03 as they fit; AS-06 after Chapter 3
Mid-year	Draft transcript line and course description; count hours so far
Before any editor assistant at 17	AS-04 Build From a Blank, dated
Late spring	AS-05 Whose Work Audit; AS-07 Portfolio Defense; year summary; final transcript lines
Any time	SYS-03 restore check; two places verified

A thin month is information, not failure. The audit says what to protect next month. Inflating May to hit 150 is failure of a different kind.

Honest and dishonest records

Honest records are a little uneven. They show a month where Git messages were lazy, a project that shipped late, a Whose Work card that admits a tool drafted a helper. Dishonest records are smooth: round numbers with no Debug Log behind them, every project marked “original” with no history, an “AP Computer Science” title on a course that never passed the Course Audit, a certificate this book never issued.

If a relative asks for a transcript and you don’t have one yet, write it from the logs you do have. Do not invent a prettier year. Colleges and employers have seen thin records before; they have also seen inflated ones. Thin and honest is safer.

Special situations

Two students in the same Hours. Separate Debug Logs, separate Whose Work cards, separate portfolio folders. A shared project names both contributors and what each person can explain cold.

A mid-year move into a school. Hand the school the transcript lines, course descriptions, hour totals, and a portfolio sample. Ask what they need for placement; do not argue from this book’s authority.

A student who started the series late. Start at Chapter 1 of this book. Prior years from *Computing for Young Minds* or elsewhere can appear as separate transcript lines if the hours and evidence exist. Do not back-fill titles without the work.

Co-op classes. The co-op may issue its own description. Your family transcript can list the co-op course by ordinary title and note “taught in co-op; portfolio on file at home.”

Incomplete security year. Do not use the transcript title **Cybersecurity** unless the family security rule was signed and followed and the year was mostly security practice on the student’s own systems. A few phishing try-its inside a Computer Science year stay inside Computer Science.

How to teach the records week

Treat one Computing Hour as a records Hour. **Specify** the transcript line and the hour target. **Pre-dict** where the portfolio is thin. **Build and Test** by running one AS check and opening three project folders cold. **Debug and Log** anything that doesn’t run. **Review** the Whose Work cards against Git history (AS-05). **Ship and Explain** with the exit line about the portfolio pick.

Exact wording you can say:

- “Show me the minutes behind this credit.”
- “Which project would you defend tomorrow morning?”
- “Where is the second copy of this folder?”
- “Does this title match what we actually did?”

In-home — AS-05 Whose Work Audit

- **Time:** 10 minutes
- **Materials:** three project folders; their Whose Work cards; Git log or changelog
- **Safety and privacy:** no pasting code into a detector
- **The fun:** catching a blank card on a polished project
- **The skill:** matching disclosure to history
- **The crumb:** signed audit line

How it goes. For each project, read the card, skim the history, ask for a cold explanation of one function. Sign “audit [date]: matches / needs rewrite.”

In-home — PROJ-02 Version History and Changelog

- **Time:** 10 minutes
- **Materials:** an existing Git repo or dated folder copies
- **Safety and privacy:** no secrets in commit messages
- **The fun:** turning “stuff” commits into a story
- **The skill:** commits plus a human-readable changelog
- **The crumb:** CHANGELOG file

How it goes. Write five lines naming what changed and when. If Git messages were “stuff,” rewrite the changelog in plain English without rewriting published history unless you know how.

In-home — PROJ-03 Demo Day

- **Time:** 30 minutes
- **Materials:** one running project; two users who didn't build it
- **Safety and privacy:** first names only; no faces recorded without consent
- **The fun:** watching a sibling use it wrong in a useful way
- **The skill:** collecting feedback without fixing on the spot
- **The crumb:** feedback notes

How it goes. Parent or co-op partner is first user. Student takes notes, not the keyboard. Afterward: one commit or one plan note for the top request.

In-home — PLAN-03 Scope It Down

- **Time:** 15 minutes
- **Materials:** an oversized project idea
- **Safety and privacy:** none special
- **The fun:** crossing out features without guilt
- **The skill:** cutting to a version that ships in two weeks
- **The crumb:** before/after scope list

How it goes. Two columns: dream list, two-week list. Everything that moves to the dream list gets a “later” date, not a trash can.

In-home — SYS-03 Files, Backups, Two Places

- **Time:** 15 minutes
- **Materials:** portfolio folder; second drive or family cloud
- **Safety and privacy:** portfolio stays off shared bot computers
- **The fun:** restoring a “lost” practice file
- **The skill:** tested restore
- **The crumb:** restore check line

How it goes. Same as Chapter 7. Before transcript season, restore one real portfolio file end to end.

Grades, narratives, and what to put in the box

Families choose different evaluation styles. A letter grade, a percentage, or a short narrative all work if the evidence underneath is the same. What does not work is a glowing narrative with no artifacts, or an “A” with forty logged hours.

If you use letter grades, write one sentence about what an A required this year (for example: “specs with done lines, tests that can fail, Whose Work cards, and a portfolio defense”). If you use narratives, three to five sentences per course is enough — the same shape as the course description, plus

what the student still found hard. Put AP scores and dual-enrollment grades in their own boxes so a reader can see which marks came from which institution.

If your student moves to a school

Ask the receiving school what they need for credit and placement. Offer the transcript, course descriptions, hour totals, and a portfolio sample. Some schools grant credit as labeled; some place by their own test; some do both. Stay calm and factual. This book is a teaching manual, not a negotiation script.

Looking ahead

A student who finishes a year with honest hours, a defensible portfolio, and a transcript line a stranger can read is ready for whatever comes next: another year under this book, AP, dual enrollment, a job, or *Computing Refresh for Adults* if a parent wants a year of their own. Place by skill, not birthday. A path through high-school computing is the promise; a certificate from this book is not.

If your student is seventeen and still thin on AS-04, wait on the editor-assistant project and spend another month on Build From a Blank and Explain Your Code. If your student is fifteen and already defending portfolios like AS-07, you may mark ready-16 moves early — and you still keep AI coding assistants off until seventeen and a passed AS-04. Skill opens doors; the editor-assistant rule still stands.

One quiet use of this chapter: the day your student asks, “Did I actually do a real computer science year?” Open the Debug Logs, the specs, the Whose Work cards, and the hour total. The answer should be visible without a speech.

In-home — PLAN-02 The Hour Audit

- **Time:** 10 minutes
- **Materials:** a month of Debug Log pages
- **Safety and privacy:** first names only if shared with a co-op
- **The fun:** discovering where the hours actually went
- **The skill:** reviewing time on task, help used, and what moved
- **The crumb:** dated audit note

How it goes. Total the minutes. List the three projects that moved. List the help disclosed. One sentence: what to protect on next month’s Monday plan.

In-home — AS-07 Portfolio Defense

- **Time:** 20 minutes
- **Materials:** one project folder; a parent; optionally one outside adult from a co-op
- **Safety and privacy:** no last names of users; no addresses in demos

- **The fun:** answering hard questions about work you know cold
- **The skill:** defending choices and trade-offs
- **The crumb:** parent's notes

How it goes. Three questions: why this design, what you tested and why, what you'd change. Parent writes one line per answer. Ready 16 when AS-03 is solid.

In-home — PROJ-01 Capstone Spec and Plan

- **Time:** 30 minutes
- **Materials:** paper or a text file
- **Safety and privacy:** no real other people's data without consent
- **The fun:** choosing a problem that matters to someone in the house
- **The skill:** problem, user, features, tests, dated schedule
- **The crumb:** spec packet

How it goes. One user, three must-have features, two edge cases, a done line, and a two-week schedule. Scope It Down (PLAN-03) if the list doesn't fit.

Out-of-home — OUT-06 Visit a College Class or Open House

- **Time:** 60 minutes
- **Materials:** notebook; questions prepared in advance
- **Safety and privacy:** first names in notes; ask before posting anything
- **The fun:** seeing the real pace of a college CS class
- **The skill:** asking how grades are shared and what hours per week look like
- **The crumb:** notes and questions

How it goes. Before any dual-enrollment decision. Ask about weekly hours, grading, and how parents see grades (if at all). Write answers the same day.

In-home — SPEC-05 Spec Diff

- **Time:** 15 minutes
- **Materials:** original spec and the shipped project
- **Safety and privacy:** none special
- **The fun:** admitting the extra feature out loud
- **The skill:** comparing what the spec said with what shipped, and writing the change note
- **The crumb:** spec-vs-shipped note

How it goes. Two columns: promised, shipped. Extra features get added to the spec with a reason, or cut with a reason. Silence about drift is the failure mode.

17–18 only.

Chapter 10

Resources



How to read this list

This chapter is a cupboard, not a ranking. Take what fits your student, your devices, and your week. Library and no-cost options come first. No prices appear here. “Free” means verified free to use for the described purpose on the access date in the notes; check the current page before you depend on it. Fit matters more than fashion. A pencil and a Debug Log still beat a polished platform nobody

opens.

CodeAI (formerly Code.org) appears below by its current name. Other tools are named plainly once, then referred to by what they do.

Choosing a tool: five questions

1. Can we start on paper this week without it?
2. Does it need an account, and do we meet the age terms?
3. Will it run offline when the network is down?
4. Can a stranger (you) see what the student did without a teacher dashboard?
5. If it vanished tomorrow, would the Debug Log and portfolio still stand?

If the answer to the last question is no, keep looking.

Start at the library

Public libraries often lend laptops, offer quiet hours, and stock plain introductions to Python, Java, networks, and web standards. A library Computing Hour still follows Family safety basics in **The Computing Hour**: log out, take your files with you, save in two places. OUT-02 from earlier in the book is the pattern.

Ask a librarian for: a beginner's Python book, a Java reference, a practical networking overview, and whether the library carries any loaner hardware. No purchase required to begin.

Free and offline first

- **Paper, cards, a printer** — specs, test tables, packet relays, threat models, Whose Work cards.
- **Thonny** — a free Python editor that runs offline and needs no account; a solid default at 15–16 if text coding is still new.¹³
- **A free JDK and a simple editor** — enough for the Java strand in Chapter 4.
- **SQLite** — built into Python's standard library (sqlite3); no separate server for Chapter 5.
- **A browser and a text editor** — enough for HTML, CSS, and JavaScript in Chapter 6.
- **Git** — free version control for local history; an optional hosting account is separate and has its own age terms (commonly 13+).

Courses by fit (not ranked)

Need	Fit	Parent load	Notes
Structured year, AP CSP-aligned	CodeAI (formerly Code.org) Computer Science Principles — free, grades 9–12	low–medium	Can be taught as intro or AP; account rules apply; check the current terms ⁵³
Browser Python practice	CodeAI Python Lab (designed for students 13+)	low	Optional supplement, not a transcript title
Offline Python	Thonny + this book’s chapters 2–3	low	No account
Java / AP CSA practice	Free JDK + editor; College Board AP Classroom if a coordinator enrolls the student	medium	AP enrollment is through a school coordinator (Chapter 9)
Security practice	Student’s own toy apps and machines under the family security rule	medium	Never someone else’s system
Web reference	MDN Web Docs	low	Read, then build locally
Language references	Python docs; Java SE docs; SQLite docs; <i>Pro Git</i> (the official Git book)	low	Bookmark the pages you actually use

Self-paced video courses vary widely in accuracy and in how much they invite copying. If you use one, keep the Whose Work card honest and keep AS-04 Build From a Blank in the year.

References worth keeping open

- **MDN Web Docs** — HTML, CSS, JavaScript, accessibility.
- **Python documentation** — language reference and library docs (no version number required in your notes to the student; use the docs that match what is installed).
- **Java documentation** — language and API docs for the JDK you installed.
- **SQLite documentation** — SQL dialect used in Chapter 5.
- **Pro Git** — free official Git book for Chapter 6 habits.
- **CISA Secure Our World** — four household habits (passwords/manager, two-step login, phishing, updates).⁴¹
- **NIST password guidance in plain words** — length beats symbol tricks; no forced monthly changes (see Chapter 7).

Competitions and programs

Check the current dates before you plan a season. None of these are transcript titles.

- **CyberPatriot** — next season; home-school teams compete in the Open Division; an adult coach is required; registration usually closes in early fall.⁴²
- **Bebras Challenge** — usually November; free in the USA Bebras window; age bands include Juniors (14–16) and Seniors (16–18); home-educated students are welcome in many national programs — check your country’s page.⁵⁴
- **ACSL** (American Computer Science League) — contest rounds across the school year; registration timing varies; check the current dates.⁵⁵
- **USACO** — programming contests; check the current dates and divisions.⁵⁶
- **Congressional App Challenge** — a project artifact with a civic frame, for middle and high school students; check the current dates; the family’s publishing rules still apply.⁵⁷
- **CyLab Security Academy** (formerly picoCTF) — free cybersecurity practice and events from Carnegie Mellon University’s CyLab; always-free challenges and learning paths.⁵⁸ Practice stays inside the family security rule when work moves off the platform onto your own machines.

For the parent: references worth knowing

CSTA 2026 High School Foundational Standards (sidebar)

The Computer Science Teachers Association’s 2026 High School Foundational Standards cover **Grades 9–12, Ages 14–18**, with 46 standards across algorithms, programming, data, systems, and impacts.⁵⁹ Specialty areas (for later study) include AI, cybersecurity, data science, game development, physical computing, and software development. This book maps activities to those codes in the notes; you do not need to teach from the PDF. One useful line from the overview: computer science is not “using generative AI tools”; “CS is what powers AI.”

When you read a code such as HS-SYS-SE-33, the middle letters name the concept (systems, security) and the number is the sequence in the list. Quote exact text when you need it; the chapter notes already carry the lines this book relies on.

AP sidebars

AP Computer Science A. Java. Multiple-choice plus four free-response types (methods and control structures; class design; ArrayList; 2D arrays). Exam in the first two weeks of May. Home-schoolers test through an authorized school’s AP coordinator; final ordering falls in mid-November. “AP” on a home course title needs the Course Audit.^{24,23}

AP Computer Science Principles. Broad introductory CS. Generative AI may be used as a supplement when the student understands, explains, and acknowledges AI-created code. The course is redesigned for 2027–28 — **no unit list here**, because it would go stale. Same ordering and May exam window pattern; check the current dates.^{24,23}

Credit, privacy, and safety pages

- HSLDA high-school credit and transcript help pages — useful guidelines; not law (Chapter 9).⁸
- FERPA dual-enrollment guidance from the Student Privacy Policy Office — plan grade-sharing before the first college class.⁵⁰
- FTC TAKE IT DOWN Act pages — platform duties for nonconsensual intimate images, real or AI-generated.⁴⁸
- NCMEC Take It Down and CyberTipline — removal help and reporting for images of anyone under 18.⁴⁸
- CISA Secure Our World; NIST SP 800-63B-4 (password guidance) — household security habits (Chapter 7).

Not recommended

Trinket — the service shut down on 31 August 2026. If an older guide points there, use Thonny offline or another family-approved editor instead.⁶⁰

Free, unplugged (a longer shelf)

These need no screen and almost no money:

- Index cards for objects, packet slips, phishing sorts, and human classifiers
- A kitchen timer for Cold Trace and Build From a Blank
- Graph paper for network diagrams and class diagrams
- A binder with tabs: Specs, Tests, Debug Log, Whose Work, Transcript
- Sticky notes for Git-commit timelines on a wall
- An old unplugged computer an adult may open (Chapter 7)
- A walk outdoors for Network Spotting and Where Is AI Deciding?

If a try-it in chapters 1–8 lists a screen mode and you have no machine that day, ask: what is the skill on paper? Specs, traces, threat models, and teach-backs almost always have an answer.

Free, on a screen

- Thonny (Python, offline)
- System terminal or command prompt already on the computer (Chapter 6)
- Browser developer tools that ship with the browser (NET-03)
- SQLite through Python (no install beyond Python itself)
- Git (install once; use locally even if you never create a hosting account)
- MDN, language docs, and CISA pages in the browser
- CodeAI CS Principles and Python Lab under their terms
- CyLab Security Academy practice challenges (free; stay inside the family security rule for anything you reproduce at home)

Avoid collecting five accounts in one weekend. One new tool per month is plenty.

Fit at a glance

If your student...	Reach for...	Wait on...
Is new to text coding	Thonny; chapters 1–3 paper-heavy	Java, public repos
Writes Python comfortably	Chapters 4–6; Git locally	Publishing (parent decision)
Wants AP CSA	Java strand; coordinator by early fall	Calling a non-audited course “AP”
Loves security puzzles	Family rule; own toy apps; CyLab Security Academy; CyberPatriot next season	Scanning anything you don’t own
Wants a college class	OUT-06; dual-enrollment office; FERPA talk first	Signing without a grade-sharing plan
Lights up at AI tools	Chapter 8 order: paper → claim-check → Whose Work → AS-04 before any editor assistant	Detectors, companion apps, private photo uploads

Optional hardware

Hardware is optional. A family’s existing laptop is enough. Useful when available, never required:

- A second-hand machine for HW-01 tear-down practice (unplugged, adult opens it)
- A spare USB drive for the “second place”
- A simple external keyboard for accessibility testing in Chapter 6

No chip brands. No shopping list with prices. Borrow before you buy.

A preview of version control

If Chapter 6 is still ahead and you want a one-sentence preview: Git is a history tool, not a backup by itself. Commits are save points with messages. The second place is still a second drive or family cloud (SYS-03). Hosting sites are optional and have age terms.

Programs that look official

Treat company “certificates of completion” as portfolio artifacts at most, never as transcript titles and never as something this book endorses as a credential. The same goes for competition placements. Chapter 9’s titles — Computer Science, Computer Programming, Web Development, Cybersecurity, Data Science — are the ones a stranger can read.

Low-device and no-device homes

You can run a serious month on paper: specs, algorithms with cards, trace tables, Parsons strips, schemas, wireframes, network diagrams, threat models, Whose Work cards, and oral defenses. Borrow screen time at a library for the hours that truly need a compiler or a browser. Two places still applies to paper photographs of whiteboards if those are your artifacts — one binder at home, one envelope at a relative's, or a scanned copy on a family drive.

A no-device week might look like this: Monday SPEC-01 and a talk box; Tuesday ALG card searches; Wednesday a phishing sort and Stop, Save, Tell; Thursday a portfolio defense of a paper prototype; Friday an Hour Audit. When a machine returns, catch up on one Build From a Blank rather than five half-finished tutorials.

Co-ops, siblings, and pairing

Co-ops are natural homes for Demo Day, Portfolio Defense, and packet-relay try-its. Keep separate Debug Logs per student. Pairing (two students, one keyboard) can help on stubborn bugs; switch driver and navigator on a timer, and still collect individual AS checks so ownership stays clear. Siblings in different bands can share OUT-04 Network Spotting and OUT-05 Where Is AI Deciding? with roles matched to age.

If you host a co-op section from this book, agree in writing on the family security rule before any security try-it, and agree that publishing anyone's project needs that family's parent decision. Shared Wi-Fi at a church or library is not permission to test that network.

A starter kit by band

15–16. Thonny or a simple editor; paper spec pads; a password manager the parent holds; signed family security rule; library card; this book.

17–18. The above, plus a free JDK, Git on the student's machine, a family drive with verified restore, and — only after AS-04 — a decision about one named editor-assistant project under the AI box rules.

Ready 16. Skill, not birthday: open ready-16 try-its when Cold Trace and Explain Your Code are solid, even if the calendar says sixteen.

Resources by chapter (quick pointers)

Chapter	Reach for
1 Specs and algorithms	Paper, cards, stopwatch
2 Python as a tool	Thonny; Python docs
3 Testing and debugging	Printer for test tables; Debug Log binder

Chapter	Reach for
4 Objects and Java	JDK; Java docs; index cards
5 Data and SQL	SQLite via Python; spreadsheet for messy originals
6 Web, terminal, Git	Browser; text editor; MDN; Git; <i>Pro Git</i>
7 Systems and security	Old unplugged machine; password manager; family rule page
8 Working with AI	Paper classifiers; primary sources; Take It Down help card
9 Credit and transcript	Hour audits; portfolio drive; AP Course Ledger

What “done enough” looks like for a resource year

You do not need every link in this chapter. You need:

- One offline way to write Python (and Java, if you take that strand)
- One place for files in two copies
- One password manager habit and a signed family security rule
- One honest path to records (Chapter 9)
- Bookmark access to MDN or language docs when web or AP work begins
- The safety cards from Chapter 8 (Stop, Save, Tell; Take It Down) printed or saved offline

If those six are in place, the rest of this cupboard is optional seasoning.

When a resource fights the book

Sometimes a flashy course asks students to paste generated code they can’t explain, publish personal data, or “try this exploit on a live site.” Put the resource down. Our rules win: specs first, explain-back, own systems only, Whose Work cards, parent decision on publishing. A free course that breaks those rules is expensive in the ways that matter.

Accessibility and assistive tools

Large text, keyboard-only navigation, screen readers, and captions are part of building for people (Chapter 6) and part of using tools as a learner. Prefer platforms that offer text-to-speech, captions, or offline reading when you have a choice. The student who needs assistive tech is not doing a lesser Computing Hour; they are doing the same Hour with the interface that fits.

A note on NCWIT and similar organizations

Some programs aimed at broadening participation in computing are excellent and some have eligibility rules that change. This book does not claim homeschool eligibility for any particular NCWIT

program. If a program interests you, read its current eligibility page and ask them directly. Fit, not prestige.

Keeping the cupboard small

Write a one-page “tools we actually use” list once a semester: editor, language docs, password manager, second-place backup, and any course login. Cross off anything unused for sixty days. A short list is easier to keep safe, easier to restore, and easier to explain on a Whose Work card.

When friends recommend a new app, run the five questions at the top of this chapter before anyone creates an account. Curiosity is good. Account sprawl is how footprints grow without anyone meaning them to.

Where to go next in this series

- *Computing for Little Thinkers* (ages 5–10) — younger siblings.
- *Computing for Young Minds* (ages 11–14) — the step before this book.
- *Computing for Emerging Adults* (ages 15–18) — this book.
- *Computing Refresh for Adults* — for a parent who wants a year of their own.

A student new to the series can start at Chapter 1 of this book without the earlier titles. This book never depends on a different computing title outside the WisdomForge Computing ladder.

If your student finishes early and wants more depth, prefer a deeper portfolio project, an AP path through a coordinator, dual enrollment, or a competition season — not a scramble through unrelated certificates. The focus line still closes the year: specify it before you build it; predict before you run; debug before you rewrite; explain it before you call it yours.

Keep the cupboard honest, small, current, and backed up in two places. The best resource in the list is still the one your student can explain without opening.

A Note on Sources

Standards, studies, government pages, program pages 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 12 October 2026. A few pages refused automated reading that day, and the notes say how their wording was checked. The pediatrics policy statement was checked against its PubMed record and a search engine's copy. Meta's AI terms, the FBI's sextortion pages, the FTC's TAKE IT DOWN Act announcement and the College Board's AI guidance for AP Computer Science Principles were read through a search engine's copy of the official pages. The physical-activity guidelines are summarized in plain words rather than quoted. 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.

Some things rest on the best available source rather than a perfect one, and the book says so instead of filling the gap. The share of homeschooled high-school students taught computer science comes from a 2016 federal survey, the most recent found that asked. The CSTA standards are written for schools; this book uses them as a map, not a requirement. 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 Python, Java, SQL, web and Git samples were run to check their output, and an error message may be worded slightly differently on your screen. The six-step Hour, the count-to-five pause, the dose suggestions, 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 families in the examples are illustrations, not real people. HSLDA is an advocacy organization; its credit-hour figures are guidelines, not law.

Competition and exam dates in the notes belong to the 2026–27 season and are labeled that way; the chapters give the usual pattern instead, because dates change every year. Program pages, platform terms, age rules, state rules and dates change; the date in each note says when it was read. Nothing here is legal advice. Check your own state's homeschool rules with the state itself. Access date for web sources: 12 October 2026.

Notes

1. National Center for Education Statistics, *Parent and Family Involvement in Education: 2023* (NCES 2024-113), U.S. Department of Education, 2024, <https://nces.ed.gov/pubs2024/2024113.pdf>, accessed 12 October 2026. Among students aged 5 through 17 in kindergarten through grade 12, “some 3.4 percent were reported as being homeschooled” in 2022–23, about 1,765,000 students.
2. American Academy of Pediatrics, Council on Communications and Media (Tiffany Munzer et al.), “Digital Ecosystems, Children, and Adolescents: Policy Statement,” *Pediatrics* 157, no. 2 (February 2026): e2025075320, published online 20 January 2026, <https://publications.aap.org/pediatrics/article/157/2/e2025075320/206129/>, accessed 12 October 2026 (wording checked against the PubMed record, <https://pubmed.ncbi.nlm.nih.gov/41556917/>, and a search engine’s copy, because the journal page refused automated reading). The statement says the amount of screen media “might vary based on each family,” that “the most important considerations are high-quality content and prioritizing healthy activities,” and advises families to “establish phone-free zones during mealtimes, in bedrooms, an hour before bed, and during homework.”
3. World Health Organization, *WHO Guidelines on Physical Activity and Sedentary Behaviour*, WHO, 25 November 2020, <https://www.who.int/publications/i/item/9789240015128>, accessed 12 October 2026. For ages 5–17 the guidelines recommend an average of 60 minutes a day of moderate- to vigorous-intensity physical activity across the week and limiting sedentary time, particularly recreational screen time; they give no numeric screen limit.
4. Shalini Paruthi et al., “Recommended Amount of Sleep for Pediatric Populations: A Consensus Statement of the American Academy of Sleep Medicine,” *Journal of Clinical Sleep Medicine* 12, no. 6 (2016): 785–786, <https://aasm.org/resources/pdf/pediatricsleepdurationconsensus.pdf>, accessed 12 October 2026: “Teenagers 13 to 18 years of age should sleep 8 to 10 hours per 24 hours on a regular basis to promote optimal health.”
5. Hamsa Bastani, Osbert Bastani, Alp Sungu, Haosen Ge, Özge Kabakcı, and Rei Mariman, “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 12 October 2026. A field experiment with about 1,000 high-school math students in Turkey: access to an unrestricted chatbot improved performance on practice problems but lowered performance on a later exam taken without it; a tutor version with guardrails, which gave hints rather than answers, largely avoided the harm. High-school mathematics, not programming, and not homeschooled students.
6. SpaceXSI, “Frequently asked questions,” Grok Bot documentation, SpaceXSI Docs, last updated

29 September 2026, <https://docs.x.ai/grok-bot/faq>, accessed 12 October 2026: “Every Bot on your account uses one persistent cloud computer. They share its files, browser sessions, and logins so they can hand work off. The computer is assigned per user, not per Bot. Do not use separate Bots as a security boundary.”

7. Minimum ages as stated on each service’s terms or help pages, accessed 12 October 2026: OpenAI, Terms of Use (United States), effective 1 January 2026, <https://openai.com/policies/terms-of-use/>: “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”; Google, “Guide your child’s Gemini Apps experience,” Gemini Apps Help, <https://support.google.com/gemini/answer/16109150> (a parent-supervised path for children under 13, or the applicable age in their country); Microsoft, Copilot Terms of Use, <https://www.microsoft.com/en-us/microsoft-copilot/for-individuals/terms-of-use> (“at least 13, but sometimes older, depending on your country’s laws”); GitHub, Terms of Service, <https://docs.github.com/en/site-policy/github-terms/github-terms-of-service> (“A User must be at least 13 years of age”); Anthropic, “Minimum age requirement,” <https://support.claude.com/en/articles/13117299-minimum-age-requirement-access-restriction> (users must be “at least 18 years old to create and use a Claude account”); SpaceXSI, Terms of Service – Consumer, last updated 12 October 2026, <https://x.ai/legal/terms-of-service> (at least 13, with a parent’s or legal guardian’s permission from 13 to 17); Meta, Meta AIs Terms of Service, <https://www.facebook.com/legal/ai-terms> (no access under 13, or a higher age where a country requires it; read through a search engine’s copy of the page because it does not open for automated readers). Terms change; check the current terms before your student signs up.

8. 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 12 October 2026. For a core course, “log at least 150 hours for 1.0 credit (roughly five hours a week for 30 weeks)”; for a lab science, a minimum of 180 hours; for an elective, “log 120+ hours for 1.0 credit and 60+ hours for 0.5 credit.” HSLDA is a homeschool advocacy organization, and these figures are guidelines, not law.

9. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-ALG-PS-03: “Evaluate algorithms for efficiency, correctness, and clarity, using metrics or test cases.” The document lists 46 foundational high-school standards (HS-ALG-PS-01 through HS-SOC-CE-46). The standards describe what schools aim for; they are a map for this book, not a requirement for homeschoolers.

10. 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): 764–792, <https://doi.org/10.1037/edu0000314>, accessed 12 October 2026 (title and publication details checked against the DOI record). The meta-analysis reports a moderate overall transfer effect, larger for skills close to programming than for skills further away, with smaller effects against active control groups and the smallest for school achievement and literacy. Studies vary widely in age and design, so this book treats transfer as possible, not promised.

11. 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): 1–13, <https://doi.org/10.1145/2996201>, accessed 12 October 2026 (title and abstract checked against the DOI record). The meta-analysis pooled 18 manuscripts with 28 independent effect sizes and 3,308 students; its findings “suggest positive results in favor of pair programming in three of four domains” (programming assignments, exams, and passing rates), “with exception to affective measures.” Most studies were in college courses.

12. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-PRO-PD-13: “Use documentation, libraries, application programming interfaces (APIs), and other tools in program development.”

13. Thonny, “Thonny, Python IDE for beginners,” <https://thonny.org/>, accessed 12 October 2026. Official downloads for Windows, macOS, and Linux, with installers that include Python; no account is needed. The page describes the debugger: “press Ctrl+F5 instead of F5 and you can run your programs step-by-step,” with F6 for a big step and F7 for a small one, and View → Variables to watch values.

14. CodeAI (formerly Code.org), “Python Lab,” <https://code.org/en-US/tools/python-lab>, accessed 12 October 2026: a browser-based environment “Designed for students aged 13+,” listed for ages 13 to 18.

15. Python Software Foundation, *The Python Standard Library* and *The Python Tutorial*, <https://docs.python.org/3/>, accessed 12 October 2026. On `int()`: <https://docs.python.org/3/library/functions.html#int>; leading and trailing whitespace around the digits is accepted. Every Python sample in this chapter was run on 12 October 2026 and its stated output confirmed, including `int("7 ")` returning 7, `int("7.5")` and `int("")` raising `ValueError`, the reading-log program’s `ValueError: not enough values to unpack (expected 2, got 1)` on a blank line, and `[4, 7, 2]` from the list-removal example. Error message wording can differ slightly between Python versions.

16. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, 2026, standard HS-PRO-TR-19, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Full text: “Evaluate a computing technology’s alignment with design specifications and responsible design values, including its correctness, effectiveness, and user experience.” Activity codes in this chapter map to TEST-01, TEST-04, AS-06 (HS-ALG-PS-03); TEST-02, TEST-03, BUG-01, BUG-03, AS-02 (HS-PRO-TR-19); BUG-02 (HS-PRO-RD-17); TEST-05, BUG-05 (HS-PRO-TR-20); BUG-04 (HS-ALG-PS-03).

17. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-ALG-PS-03: “Evaluate algorithms for efficiency, correctness, and clarity, using metrics or test cases.”

18. Python Software Foundation, “8. Errors and Exceptions,” *The Python Tutorial*, <https://docs>

.python.org/3/tutorial/errors.html, accessed 12 October 2026. The tutorial says “The last line of the error message indicates what happened” and, of the place a syntax error is reported, “this is not always the place that needs to be fixed.” Every code sample and error message in this chapter was run on 12 October 2026 and matches the text; tracebacks are shown without the caret-and-tilde markers recent Python versions add, and with the folder path shortened to the file name. Error wording can differ slightly between Python versions.

19. Python Software Foundation, “unittest — Unit testing framework,” *The Python Standard Library*, <https://docs.python.org/3/library/unittest.html>, accessed 12 October 2026. Part of the standard library that comes with Python; named once as optional, while the chapter itself uses plain `assert`.

20. Andrew Hunt and David Thomas, *The Pragmatic Programmer: From Journeyman to Master* (Addison-Wesley, 1999), section “Debugging,” excerpt at <https://courses.cs.duke.edu/fall21/compsci307d/readings/pragmatic-programmer-debugging.pdf>, accessed 12 October 2026. If you “changed only one thing” and the system stopped working, “that one thing was likely to be responsible”; explaining what the code is supposed to do “often causes the problem to leap off the screen and announce itself.” Practitioner advice, not a study.

21. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, 2026, standard HS-PRO-PD-12, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Activity codes in this chapter: OOP-01, OOP-02 (HS-PRO-PD-12); OOP-03, PY-06 (HS-PRO-RD-17); OOP-04 (HS-ALG-PS-01); OOP-05 (HS-PRO-VD-16); OUT-03 (HS-SOC-CE-45).

22. Java is free to download and use through OpenJDK, the open-source reference implementation, <https://openjdk.org/>, and the language’s official learning site is <https://dev.java/>, both accessed 12 October 2026. Every Python and Java sample in this chapter was compiled and run on 12 October 2026 with a current free JDK, and the outputs and error messages quoted match. Compiler wording can differ slightly between Java versions.

23. College Board, “AP Computer Science A Exam,” AP Central, <https://apcentral.collegeboard.org/courses/ap-computer-science-a/exam>, accessed 12 October 2026. Section I: 42 multiple-choice questions, 55 percent of the score; Section II: four free-response questions, 45 percent (methods and control structures; class design; data analysis with `ArrayList`; 2D array). The current course framework took effect in fall 2025 (College Board, “Course Changes Overview,” <https://apcentral.collegeboard.org/courses/how-ap-develops-courses-and-exams/course-changes-overview>, accessed 12 October 2026).

24. College Board, “Ordering AP Exams,” AP Central, <https://apcentral.collegeboard.org/exam-administration-ordering-scores/ordering-fees/how-to-ap-registration-ordering/order-ap-exams>, and “2027 AP Exam Dates,” <https://apcentral.collegeboard.org/exam-administration-ordering-scores/exam-dates>, both accessed 12 October 2026. For 2026–27: preferred ordering deadline 2 October 2026; final ordering deadline 13 November 2026, 11:59 p.m. ET; spring-course orders and fall order changes by 12 March 2027. 2027 exams run 3–7 and 10–14 May; AP Computer Science A is Wednesday, 12 May 2027 (afternoon session); AP Computer Science Principles is Friday, 14 May 2027 (morning session). Homeschooled students test through a school’s AP coordinator: College Board, “I’m Homeschooled... How Can I Take an AP Exam?,” AP Students, <https://apst>

udents.collegeboard.org/help-center/im-homeschooled-how-can-i-take-ap-exam, accessed 12 October 2026. Check the current dates.

25. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, 2026, standard HS-DAT-DC-22, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Activity codes in this chapter: DATA-01, DATA-04 (HS-DAT-DI-25); DATA-02, SQL-01 (HS-DAT-DC-22); DATA-03 (HS-DAT-DC-23, “Use a computational tool to clean and organize text-based data.”); DATA-05 (HS-ALG-ML-07); SQL-02 (HS-PRO-PD-13); SQL-03 (HS-PRO-VD-16); SQL-04, OUT-02 (HS-SYS-SE-31); DS-03 (HS-PRO-RD-17).

26. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-DAT-DI-25: “Create a data visualization of a multivariate dataset to answer a question or make a classification or prediction.”

27. SQLite is a small, file-based database that Python includes as the `sqlite3` module: Python Software Foundation, “`sqlite3` — DB-API 2.0 interface for SQLite databases,” <https://docs.python.org/3/library/sqlite3.html>, and SQLite, “SQL As Understood By SQLite,” <https://www.sqlite.org/lang.html>, both accessed 12 October 2026. The `sqlite3` documentation says: “Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks.” Every Python and SQL sample in this chapter was run on 12 October 2026 against the made-up files shown, and the outputs quoted match, including the 0 loans result for the quoted trick input and the 15-row result of the join with no key condition.

28. The chart sample uses Matplotlib, a charting library for Python that is installed separately, <https://matplotlib.org/>, accessed 12 October 2026; its license page says its license “is based on the PSF license” and that it “only uses BSD compatible code,” <https://matplotlib.org/stable/project/license.html>. The sample was run on 12 October 2026 and saved the image as described. Named as one option among several, not a recommendation.

29. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, 2026, standard HS-ALG-IM-09, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Activity codes in this chapter: WEB-01, SPEC-04 (HS-ALG-IM-09); WEB-02 (HS-PRO-PD-12); WEB-03, GIT-03 (HS-PRO-RD-17); WEB-04, TEST-05 (HS-PRO-TR-20); TERM-01, TERM-02 (HS-SYS-HW-29); GIT-01, GIT-02, GIT-04, PROJ-02 (HS-PRO-PD-15); GIT-05 (HS-PRO-PD-14); BUG-04 (HS-ALG-PS-03); OUT-03 (HS-SOC-CE-45); PROJ-04 (HS-SOC-HU-43).

30. MDN Web Docs (Mozilla), “HTML: A good basis for accessibility,” https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Accessibility/HTML, accessed 12 October 2026. The guide contrasts a control marked up as `<div>Play video</div>` with a real `<button>`, which has “built-in keyboard accessibility — users can navigate between buttons using the Tab key and activate their selection using Space, Return or Enter”; says alt text “should always provide a direct representation of the image and what it conveys visually”; and says link text color “has to be significantly different from the background color (a 4.5:1 contrast).”

31. MDN Web Docs (Mozilla), “EventTarget: addEventListener() method,” <https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>, and “Document: querySelector() method,” <https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector>, both accessed 12 October 2026. The page, stylesheet, and script in this chapter were loaded in a current desktop browser on 12 October 2026 and given the inputs 3, blank, 2, two, 1.5, and 0; the messages quoted match.
32. Git project, “Documentation,” <https://git-scm.com/doc>, and “git-config,” <https://git-scm.com/docs/git-config> (for the `user.name` and `user.email` setup), both accessed 12 October 2026. Every terminal and Git sequence in this chapter (the `cd/ls/mkdir` session, `init/add/commit/log`, the branch and merge conflict, and `git bisect`) was run on 12 October 2026 in practice folders; the outputs quoted match, except that commit codes are generated and will differ on every computer.
33. MDN Web Docs (Mozilla), “What are browser developer tools?,” https://developer.mozilla.org/en-US/docs/Learn_web_development/Howto/Tools_and_setup/What_are_browser_developer_tools, accessed 12 October 2026. The Console reports the errors the browser meets as it tries to run your code.
34. MDN Web Docs (Mozilla), “Learn web development,” https://developer.mozilla.org/en-US/docs/Learn_web_development, accessed 12 October 2026.
35. GitHub, “GitHub Terms of Service,” <https://docs.github.com/en/site-policy/github-terms/github-terms-of-service>, accessed 12 October 2026: “A User must be at least 13 years of age.” Check the current terms.
36. Federal SCED (School Courses for the Exchange of Data) course titles as given in state guidance opened 12 October 2026: Washington Office of Superintendent of Public Instruction, *Guidance on Computer Science State Course Codes* (November 2025), https://ospi.k12.wa.us/sites/default/files/2025-11/computer_science_state_course_code_guidance_ada.pdf; Connecticut State Department of Education, *Computer Science Course Reporting Guidance (SCED)*, <https://portal.ct.gov/-/media/SDE/Computer-Science/Launch-of-SCED-CS-Guide.pdf>, which describes 10201 as covering HTML, CSS, JavaScript, and the DOM, and its 2024–25 computer science course list, https://portal.ct.gov/-/media/sde/performance/data-collection/help-sites/tcs/tcs_cs_courselist_2024-25_pdf.pdf; Wyoming Department of Education, *SCED Codes for WY Computer Science Courses SY2021*, <https://edu.wyoming.gov/wp-content/uploads/2021/10/SCED-Codes-for-WY-Computer-Science-Courses-SY2021.pdf>. Relevant codes include 10152 Computer Programming, 10155 Java Programming, 10157 AP Computer Science A, 10020 Cybersecurity, and 10201 Web Page Design. The national framework is explained in the National Forum on Education Statistics, *Forum Guide to Understanding the School Courses for the Exchange of Data (SCED) Classification System* (NFES 2023-087), <https://nces.ed.gov/pubs2023/NFES2023087.pdf>. This book uses “Web Development” as the clearer transcript title; a course description may say it is comparable to the state category “Web Page Design.”
37. In the United States, unauthorized access to computer systems can be a federal crime under the Computer Fraud and Abuse Act, 18 U.S.C. § 1030 (“Fraud and related activity in connection with computers”), <https://www.law.cornell.edu/uscode/text/18/1030>, accessed 12 October 2026, and state laws often cover similar conduct. The book keeps practice on the student’s own systems and toy apps for that reason. This is not legal advice; read your state’s current law.

38. National Institute of Standards and Technology, *Digital Identity Guidelines: Authentication and Authenticator Management*, NIST Special Publication 800-63B-4, <https://pages.nist.gov/800-63-4/sp800-63b.html>, accessed 12 October 2026: passwords used on their own “SHALL” be “a minimum of 15 characters in length”; services “SHALL NOT impose other composition rules (e.g., requiring mixtures of different character types)” and “SHALL NOT require subscribers to change passwords periodically,” though a change is forced if a password is compromised. Plain-language household habits appear in CISA’s “Secure Our World,” <https://www.cisa.gov/secure-our-world>.
39. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-SYS-SE-33: “Formulate a solution to a security flaw in a given system.”
40. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-SYS-NT-34: “Diagram a network of computing systems, including hardware and software.”
41. Cybersecurity and Infrastructure Security Agency, “Secure Our World,” <https://www.cisa.gov/secure-our-world>, accessed 12 October 2026. “Four Easy Ways to Stay Safe Online”: recognize and report phishing, use strong passwords, turn on multifactor authentication, and update software.
42. CyberPatriot, Air Force Association, “Competition Overview,” <https://www.uscyberpatriot.org/competition/competition-overview>, and “Join the Competition,” <https://www.uscyberpatriot.org/join-the-competition/>, both accessed 12 October 2026. The Open Division is for high school students “from schools, scouting units, Boys and Girls Clubs, home school programs, STEM programs, etc.”; teams register through an adult coach, and team assistants must be at least 18 and pass a background check. In the 2026–27 season the practice round ran 7–20 October 2026 and rosters were finalized 21 October, so a family starting now is looking at next season. Check the current dates.
43. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standards HS-ALG-ML-07: “Evaluate training data by examining its source, quality, representativeness, potential biases, and privacy implications”; HS-PRO-RD-18: “Evaluate AI-generated code for accuracy, reliability, and alignment with program requirements”; HS-ALG-PS-05: “Evaluate AI-generated output to assess bias, accuracy, and potential harms.”
44. Computer Science Teachers Association, *2026 CSTA PK–12 Computer Science Standards*, overview page, <https://csteachers.org/pk12standards/>, accessed 12 October 2026. Under “Computer Science Is Not,” the page lists “using technology tools like word processors, slide decks, or generative AI tools. These appear in CS and in every other subject too,” and “being replaced by AI. CS is what powers AI.”
45. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Standard HS-ALG-

PS-04: “Describe the differences between deterministic and probabilistic algorithms.”

46. College Board, “What is the acceptable usage of generative AI?,” AP Computer Science Principles FAQ, AP Central, <https://apcentral.collegeboard.org/courses/ap-computer-science-principles/course/faq/generative-ai-usage-guidelines>, accessed 12 October 2026 (read through a search engine’s copy): students “can use Generative AI tools as supplementary resources for understanding coding principles, assisting in code development, and debugging,” must “review and understand any code co-written with AI tools,” must “be prepared to explain their code in detail,” and “should include appropriate acknowledgment” for any code created by generative AI tools. The course is being redesigned for 2027–28 with generative AI across the framework and a new AI Create Project: College Board Newsroom, 23 September 2026, <https://newsroom.collegeboard.org/new-college-board-research-ap-teachers-push-guardrails-and-support-genai-reshapes-classroom>, accessed 12 October 2026.

47. Federal Bureau of Investigation, “The Financially Motivated Sextortion Threat,” <https://www.fbi.gov/news/stories/the-financially-motivated-sextortion-threat>, and FBI Atlanta, “Sextortion: A Growing Threat Targeting Minors,” <https://www.fbi.gov/contact-us/field-offices/atlanta/news/sextortion-a-growing-threat-targeting-minors>, both accessed 12 October 2026 (read through a search engine’s copy because the FBI site refused automated reading). Offenders “typically pretend to be alluring young girls to befriend teen boys—often on social media and gaming platforms”; victims are “typically males between the ages of 14 to 17, but any child can become a victim.” Paying rarely ends it; the FBI advises cutting off contact, keeping the evidence, telling a trusted adult, and reporting at tips.fbi.gov or 1-800-CALL-FBI.

48. TAKE IT DOWN Act, Public Law 119-12, signed 19 May 2025, text at <https://www.congress.gov/bill/119th-congress/senate-bill/146/text/enr>; 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 12 October 2026 (read through a search engine’s copy because the FTC site refused automated reading). Covered platforms must give people a way to request removal of intimate images shared without consent, real or AI-generated, and must remove them and known identical copies within 48 hours of a valid request; complaints about platforms that fail to act go to <https://TakeItDown.ftc.gov>. For images of anyone under 18, the National Center for Missing & Exploited Children runs Take It Down, <https://takeitdown.ncmec.org>, and the CyberTipline, <https://report.cybertip.org>, both accessed 12 October 2026.

49. Meng Cui and Rachel Hanson, *Homeschooling in the United States: Results from the 2012 and 2016 Parent and Family Involvement Survey (PFI-NHES: 2012 and 2016)*, NCES 2020-001, National Center for Education Statistics, U.S. Department of Education, December 2019, Table 9, <https://nces.ed.gov/pubs2020/2020001.pdf>, accessed 12 October 2026. Among homeschooled students with a grade equivalent of 9th to 12th grade, 17 percent had “computer science (e.g., computer programming)” among the subject areas taught during home instruction in 2016. It is the most recent federal figure found.

50. U.S. Department of Education, Student Privacy Policy Office, “If a student under 18 is enrolled in both high school and a local college, do parents have the right to inspect and review his or her education records?,” <https://studentprivacy.ed.gov/faq/if-student-under-18-enrolled-both->

high-school-and-local-college-do-parents-have-right-inspect, accessed 12 October 2026: “If a student is attending a postsecondary institution - at any age - the rights under FERPA have transferred to the student.” If the student is under 18, “the parents still retain the rights under FERPA at the high school,” and the college “may disclose personally identifiable information from the student’s education records to the parents, without the consent of the eligible student, if the student is a dependent for tax purposes.” The office’s “Dual Enrollment” infographic page, <https://studentprivacy.ed.gov/resources/dual-enrollment>, accessed 12 October 2026, is marked “Last Updated: August 2026.” Rights at 18 are explained in *An Eligible Student Guide to FERPA*. This is not legal advice.

51. College Board, “I’m Homeschooled... How Can I Take an AP Exam?,” AP Students, <https://apstudents.collegeboard.org/help-center/im-homeschooled-how-can-i-take-ap-exam>, and “AP Course Audit for Homeschool Providers of AP Courses,” AP Central, <https://apcentral.collegeboard.org/courses/ap-course-audit/explore-by-role/homeschool-providers-ap-courses>, both accessed 12 October 2026. Homeschooled students contact a school’s AP coordinator, “as early in the school year as possible,” to arrange to test there.

52. U.S. Department of Labor, Wage and Hour Division, “Child Labor Rules Advisor,” <https://webapps.dol.gov/elaws/elg/childlbr.htm>, accessed 12 October 2026: in nonfarm work, “Minors age 16 and 17 may perform any job not declared hazardous by the Secretary, and are not subject to restrictions on hours”; 14- and 15-year-olds may work only outside school hours, within federal hour limits. States may be stricter. See also Fact Sheet #43, <https://www.dol.gov/agencies/whd/fact-sheets/43-child-labor-non-agriculture>.

53. CodeAI (formerly Code.org), “Computer Science Principles,” <https://code.org/en-US/curriculum/computer-science-principles>, accessed 12 October 2026: a course for grades 9–12 whose “curriculum and platform are available at no cost for anyone, anywhere, to teach.” Its Python Lab is “Designed for students aged 13+,” <https://code.org/en-US/tools/python-lab>, accessed 12 October 2026. Accounts carry their own terms; check the current terms.

54. USA Bebras Computing Challenge, <https://www.bebрасchallenge.org/>, accessed 12 October 2026: “The Bebras Challenge USA 2026 is scheduled for November 9–20”; participation “is free”; age groups include Junior (14–16) and Senior (16–18), <https://bebraschallenge.org/examples/>. UK Bebras, <https://www.bebрас.uk/>, accessed 12 October 2026, also runs 9–20 November 2026 and says “Home educated students are very welcome.” Other countries vary; check the current dates on your national page.

55. American Computer Science League, <https://www.acsl.org/>, accessed 12 October 2026: “Registration for 2026-27 is now open! Contest #1 will be available on Oct 19th.” Check the current dates.

56. USA Computing Olympiad, <https://usaco.org/>, accessed 12 October 2026. The 2025–26 season ran three online contests in early 2026, followed by a proctored contest; the 2026–27 schedule had not been posted on the access date. Check the current dates.

57. Congressional App Challenge, Internet Education Foundation, <https://www.congressionalappchallenge.us/>, accessed 12 October 2026: registration for the 2026 challenge was open; to be eligible “you must be a middle or high school student at the time of app submission” (rules page,

<https://www.congressionalappchallenge.us/students/rules/>). Participation depends on your congressional district taking part. Check the current deadline.

58. CyLab Security Academy, Carnegie Mellon University, <https://www.cylabacademy.org/>, accessed 12 October 2026. The site announces that “picoctf.org is now CyLab Security Academy”; picoCTF logins created before 8 May 2026 still work.

59. Computer Science Teachers Association, *2026 CSTA High School Standards at a Glance (Foundational, Grades 9–12, Ages 14–18)*, CSTA, 2026, <https://csteachers.org/wp-content/uploads/2026/07/High-School-Foundational-Standards.pdf>, accessed 12 October 2026. Overview: <https://csteachers.org/pk12standards/>, accessed 12 October 2026.

60. Trinket, “Trinket has shut down,” <https://trinket.io/>, accessed 12 October 2026: “Trinket shut down on August 31, 2026.”