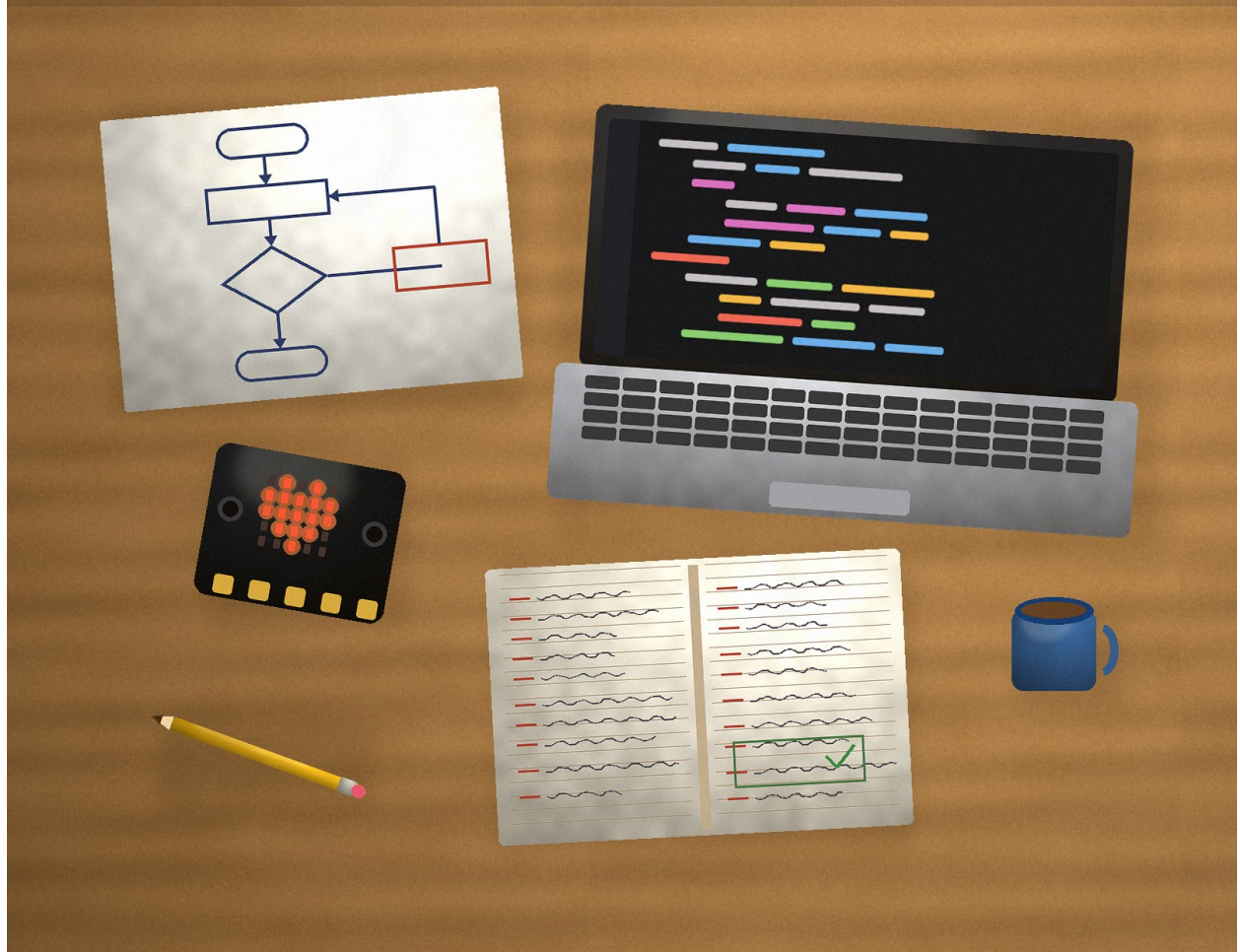


COMPUTING FOR YOUNG MINDS

Michael Gannotti



Computing for Young Minds

Michael Gannotti

A homeschool guide to computing for ages 11–14: plans, Python, the web, how computers work, safety and AI. Book 2 of the SMF WisdomForge Computing series. Illustrated edition, 6 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 11–14: Plans, Algorithms, and Thinking It Through
- Chapter 2 — From Blocks to Text: First Programs in Python
- Chapter 3 — Debugging as a Discipline: Errors, Tests, and the Debug Log
- Chapter 4 — Functions, Lists, and Data: Bigger Programs and Spreadsheets That Answer Questions
- Chapter 5 — Building for the Web: HTML, CSS, and a First Taste of JavaScript
- Chapter 6 — How Computers and Networks Really Work — and Hardware You Can Program If You Like
- Chapter 7 — Safe, Smart, and Kind Online: Passwords, Scams, Social Media, and Your Footprint
- Chapter 8 — AI Up Close: Training Data, Bias, Prediction, and Whose Work It Is
- Chapter 9 — Records, Portfolio, and Transcript
- Chapter 10 — Resources
- A Note on Sources
- Notes

Welcome

You don't need to be a programmer to teach an 11- to 14-year-old real computing.

You do need to ask "What do you predict will happen?" before anyone presses run, and to leave the program in your student's hands when it breaks.

That is what this book is for.

The book in one minute

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

Nora is twelve. She has built dozens of Scratch games: a maze, a quiz, a cat that dodges falling pizza. Today she opens a Python editor for the first time and types three lines. The third one is supposed to greet the player by name. She presses run.

Red text fills the bottom of the screen.

Her mother is sitting beside her. She can see the problem, more or less. She doesn't reach for the keyboard. "What does it say?"

Nora reads it aloud, a little offended. "Something was never closed. Line 3."

"So where's line 3?"

Nora finds it. She counts the brackets with her finger: one open, none closed. She adds the missing one and runs it again. The greeting appears. She grins, then frowns. "It could have just said that."

"It did," says her mother. "It said line 3."

That is the book in one minute. The computer did exactly what it was told. The error message was information, not judgment. The parent asked a question instead of fixing it. And the program that works now is Nora's, because Nora fixed it.

Who this book is for

This book has two readers.

The first is you: the parent, grandparent, or co-teacher at the table. You may never have written a line of code. That's fine. Each chapter teaches you the idea first, in plain words, so you can hear where your student's thinking goes off track and ask one good question.

The second is your student. At 11 to 14, young people can hold a plan in their heads, write it down, test it, and argue with you about it. They want more say in their own work, and they're ready for it. Every chapter has a section written to them directly, as "you," longer and more grown-up than the student pages in a book for younger children.

What your student will learn

Over a year, your student will plan and write real programs in Python, a language you type rather than snap together, readable enough for beginners and used by professionals. They'll learn to find and fix their own bugs calmly. They'll build a small website. They'll find out how computers and networks actually move information around. They'll learn how to stay safe and kind online, how AI tools work, and whose work it is when one of them helps.

The pace is about two or three Computing Hours a week. Each Hour is forty to sixty minutes long, and about half of the book's activities need no screen at all: paper, index cards, a deck of cards, masking tape on the floor.

The line the whole book runs on

Predict before you run; debug before you rewrite; explain it before you call it yours.

Predict before you run means writing down what you think a program will do before you start it, so there's something to check against.

Debug before you rewrite means that when something breaks, you find the one wrong thing and fix it, instead of deleting everything and starting over.

Explain it before you call it yours means that your work is what you planned, wrote, tested, and can explain, and that you say plainly what help you used.

Hold those three habits, and the rest of this manual is practice.

You are not alone

About 3.4 percent of school-age children were homeschooled in 2022–23, by the federal count.¹ Many of those families have taught computing for years with very little help, piecing together a website here and a free course there. This book is the help: one clear way to run a week, and a menu of activities you can choose from.

What this book will not do

It won't hand you a 180-day stack of worksheets. You'll get a weekly shape and a menu of named activities, and you choose.

It won't sell you a curriculum, a gadget, or a subscription. The home Python editor this book recommends is free and needs no account.

It won't promise a test score. It won't tell you that coding makes young people better at everything, either. Coding teaches coding. Other good things may come along, and that's a welcome possibility, not a promise.

It won't give anyone under 13 an account or a chatbot. At 11 and 12, you create and hold every login, and any AI tool runs on your account with you at the keyboard. At 13 and 14, your student can use some tools under their terms, with you nearby.

It won't title your student's year with a book name or hand out a certificate. If you keep records, use a plain title a stranger can read, such as **Computer Science**.

It isn't a reprint of any earlier computing book. It was written fresh for this age.

Where this book sits

Computing for Young Minds is the second title in the SMF WisdomForge Computing series. *Computing for Little Thinkers* (ages 5–10) comes before it, and *Computing for Emerging Adults* (15–18) comes after. *Computing Refresh for Adults* is for grown-ups who want their own turn.

Little Thinkers is not required. A student who never used it starts at Chapter 1 and is fine. A student who did will recognize the shape of the week and the habit of predicting first, and will find almost everything else new.

How the book is built

The front matter gives you a plan for this week, five things to remember, and **The Computing Hour**: the five-step session shape, Family safety basics, a short note on screen time, and the one box about AI. Each is said carefully, once, and the chapters point back to it.

Eight teaching chapters follow. They cover plans and algorithms; first programs in Python; debugging; functions, lists, and data; building for the web; how computers and networks work; staying safe online; and how AI works. Then come two chapters for you: records and a portfolio, and resources.

Every teaching chapter uses the same eight headings, a talk box with an exact opening question, and named activities for inside the home and outside it. Each chapter gives moves for two bands, **11–12** and **13–14**. Place your student by skill, not birthday.

Where to begin

This week, you can learn one question, “What do you predict will happen?”, and practice waiting after you ask it.

Your student can write instructions for something ordinary, like making a sandwich, and watch you follow them exactly as written. The results are usually funny, and very instructive.

You don’t need to know Python yet. You need a kitchen table, a notebook, and a question you’re willing to wait for.

Bring a pencil. The computer can wait five minutes.

— Michael Gannotti

How to Use This Book This Week

You can start this week. You don't need to read the whole book first, buy anything, or know any Python. Here is a plan for your first week, and then a short tour of how the rest of the book works.

This week

Monday (you, about 40 minutes). Read **The Computing Hour**, the session shape every chapter uses. Then read the first two sections of Chapter 1, “Why this matters” and “For the parent: understand it yourself.” That's all the preparation you need.

First Computing Hour. Unplug and Plan with PLAN-01 The Literal Sandwich Spec: your student writes instructions for making a sandwich, and you follow them exactly as written, like a very literal computer. Then Predict with PY-02 Predict the Print: a four-line program on paper, a written guess, and a check. No screen is needed today. Start the Debug Log with the sandwich bugs. The Debug Log is a dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes. Photograph the marked-up spec. Close with the exit lines: “I predicted... It did... Next I'd change...” and “Help I used: ...” (today: none).

Second Computing Hour. Play ALG-01 Guess My Number, where your student finds your secret number by halving the range each time. Then install Thonny together (PY-01 Hello, Editor). Thonny is a free Python editor for beginners. It comes with Python built in, needs no account, and works offline once it's installed.² Your student types a first program, runs it, and saves it as `2026-10-12-hello.py` in two places: the family drive (or a USB stick) and one backup.

Optional, this week or next. Write SAFE-08 Family Device Agreement together: where devices live, who holds the passwords, and what to do if something online feels wrong. It belongs to Chapter 7, but many families like to set the rules before the computer comes out. If a phone is arriving soon, do it now.

Other days. Ten minutes is plenty: a Paper Parsons (PLAN-07, cut-up lines of a program to put in order) or a short typing warm-up.

What to save from week one

- the marked-up sandwich spec (a photo is fine);
- `2026-10-12-hello.py`, saved in two places;

- one dated page of the Debug Log;
- the family agreement, if you wrote it.

That is the beginning of a portfolio. Chapter 9 shows you how to keep it.

How each chapter is built

Every teaching chapter uses the same eight headings, in the same order:

1. **Why this matters:** the idea in a page.
2. **For the parent: understand it yourself:** what you need to know to hear a wrong turn.
3. **How to teach it this week:** the first activities, and the chapter's **talk box**, which gives an exact opening question, follow-ups, and how to wait.
4. **Practice that actually builds learning:** more activities, each with a bank code.
5. **For the student:** written to your student directly.
6. **If it isn't clicking:** what to try when it's not working.
7. **Tools, including AI:** what to use, and what not to, at each age.
8. **What "done enough" looks like:** how to know it's time to move on.

Activities are marked **In-home** or **Out-of-home**, and each lists time, materials, safety and privacy, the fun, the skill, and the crumb (what to save). Moves for the two bands are marked **11–12** and **13–14**. "Ready 12" marks a 13–14 move that some 12-year-olds can take. Place by skill, not birthday.

What order to go in

Do Chapters 1–3 in order: plans and algorithms, then first Python, then debugging. After that, Chapters 4–8 can go in any order that suits your family. If a phone or a first account is arriving, move Chapter 7 earlier. Chapters 9 and 10 are for you; dip into them whenever you need them.

The bank

The book holds 100 named activities, each with a code such as PLAN-01 or PY-04. Forty-nine of them need no screen at all. You won't do them all, and you shouldn't try. Pick what fits the week and skip the rest without guilt.

A year in outline (optional)

Weeks	Focus	A project to finish
1–4	Chapter 1, plans and algorithms (agreement in week 1)	A flowchart and pseudocode for a family routine

Weeks	Focus	A project to finish
5–10	Chapter 2, Python basics	A guess-the-number game
11–14	Chapter 3, debugging	A planted-bug swap
15–20	Chapter 4, functions, lists, and data	A quiz program; a small data investigation
21–25	Chapter 5, the web	A three-page site and a demo day
26–30	Chapter 6, computers and networks	A pixel picture; a secret-code challenge
31–33	Chapter 7, safety (going deeper)	A settings tour
34–36	Chapter 8, AI	A next-word guesser; a Whose Work audit

A Cold Trace (AS-01) once a month and a demo day once a semester run all year. Treat this as a picture of one year, not a schedule to keep.

Siblings and co-ops

Siblings can work as a pair: one types (the driver), one reads and checks (the navigator), and they switch every ten minutes. Two computers side by side, with lots of talk, works just as well. In one study of students entering sixth grade, pairs and students on their own computers learned about the same, and the solo students finished a little faster.³ Pairing is a good way to work together, not a promise of better scores.

A younger sibling, say nine, can join the unplugged cards and play the literal computer. For their own Hour, *Computing for Little Thinkers* fits. A co-op can hold one demo day a semester.

Special situations

A student far ahead of you. Your job doesn't change: ask for the prediction, be the first user, and ask for the explanation. Let your student teach it back to you.

A student new to all of this at 13. Start at Chapter 1. The 13–14 moves go quickly, and the Scratch side-by-side work is optional.

No computer at home. Use the 49 unplugged activities, plus Computing Hours at the public library (OUT-02).

Learning differences. Use shorter screen blocks, larger fonts, code read aloud, and cut-up programs to put in order instead of a blank file. The Hour's five steps stay the same.

A student who has already used chatbots at 11. No shame and no scolding. Move to parent-operated use, and talk through the terms and why they exist.

A week with no screens

Sick laptop, travel, or a week at grandma's: run the Hour on paper. Trace a program by hand, sort cards, play Guess My Number in the car. Paper weeks are real computing weeks.

When life happens

You will miss days. Pick up where you left off. Two short Hours in a busy week still count. The habits matter more than the calendar.

If You Only Remember Five Things

Some weeks you won't open the book at all. On those weeks, these five are enough.

1. Predict before you run.

Before anything runs, your student writes one line: *I predict it will...* A prediction turns a program into an experiment. When the guess is wrong, you've learned exactly where your student's picture of the code differs from the computer's, and that's the most useful thing that can happen in an Hour. No run without a prediction line.

2. Debug before you rewrite — read the error, change one thing, log it.

When the program surprises you, nobody deletes anything. First write what you expected and what actually happened. Then read the error message, which usually names a line. Change one thing, run it again, and write one line in the Debug Log: the bug, the clue, the fix. "I'll just start over" becomes "Save this as a new version, then tell me the one thing that's wrong."

3. Explain it before you call it yours. "Help I used:" every time, even "none."

Your student's work is what they planned, wrote, tested, and can explain. If they can't explain a line, it isn't theirs yet. Every Computing Hour ends the same way:

"I predicted... It did... Next I'd change..."

"Help I used: ..."

A tutorial, a sibling, you, an AI tool: all of it counts, and none of it is shameful. Anything a tool generated is made by a tool, not your student's own work.

4. Paper first, then screen; save in two places; screens out of the bedroom.

Every Hour starts with the screen dark and a plan on paper. About half of this book's activities never need a screen at all. When there is a file, it lives in two places: the family drive (or a USB stick) and one backup, with a date-first name a stranger could read. The computer lives in a shared room, and phones and laptops sleep outside bedrooms.

5. You hold the passwords at 11–12 and share them at 13–14; AI is parent-operated under 13 and supervised after.

At 11 and 12, you create and hold every login, and any AI tool runs on your account with you at the keyboard. At 13, many services let a young person sign up. A big reason is a federal privacy law that requires a parent's consent only for children under 13.⁴ Thirteen is a rule for companies, not a readiness test. At 13 and 14, your student may hold passwords for tools the family has approved, with two-step login turned on and a copy kept by you, and may use AI tools whose terms allow their age, with you nearby. Check the current terms before you start. The details live in **The Computing Hour**, under Family safety basics and the AI box.

If you remember only one sentence, make it this one:

Predict before you run; debug before you rewrite; explain it before you call it yours.

The Computing Hour



Forty to sixty minutes. Five steps. Paper first, screen second, and a clear stop.

“Hour” is the family word. At 11–12 a session runs about 40–50 minutes, roughly 20–30 of them on a screen. At 13–14 it runs about 50–60, and a project day can stretch to about 75 with a movement break. Later chapters say “run the Computing Hour” and point back here, so you learn it once.

Every Hour runs on one line:

Predict before you run; debug before you rewrite; explain it before you call it yours.

Parent warm-up (about three minutes)

Read today's talk-box opening aloud once. Run the day's program yourself, on your own copy, so nothing surprises you except the interesting things. Check where today's file will be saved. Put your own phone face down.

The five steps

Step	11–12	13–14	What happens
1. Unplug and Plan	5–10 min	5–10 min	Paper first: pseudocode, a flowchart, a test table, or one unplugged card.
2. Predict	2–5 min	2–5 min	A written prediction line in the Debug Log before anything runs.
3. Build and Test	15–25 min	20–35 min	Small steps, run often, test against the plan.
4. Debug and Log	5–10 min	5–15 min	Expected/Actual, read the error, one change at a time, one Debug Log line.
5. Save and Explain	3–5 min	3–5 min	Save a new version in two places. Explain it, and say what helped.

1. Unplug and Plan

The screen stays dark. Your student puts a plan on paper: a few lines of pseudocode (a plan for a program written in plain words), a flowchart, or a test table that lists inputs and the outputs they expect. Some days the whole step is one unplugged card from the chapter. Ask the talk-box opening here. A plan doesn't need to be right. It needs to exist, so there is something to test against.

2. Predict

Before anything runs, your student writes one line in the **Debug Log**, a dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes. No run without a prediction line. The model for this step is PY-02 Predict the Print: a short program on paper, a written guess, then the run.

```
score = 3
score = score + 2
```

```
print("Score:", score)
print(score * 2)
```

This prints Score: 5 on one line and 10 on the next. A student who predicted Score: 3 has just met the most useful idea in Chapter 2: a variable is a labeled box whose contents can change, and line 2 changed it. A wrong prediction is not a failure; it is a window into your student's thinking.

3. Build and Test

Your student types, runs, and tests in small steps, running after every few lines and testing against the plan. Most weeks follow the same arc: use a working program, change one part, then make something new.

Your hands stay off the keyboard. Your job is **first user**: when the program is ready, you try it the way a stranger would and report what happened, without fixing anything.

4. Debug and Log

A bug is a mistake in the instructions; debugging is finding and fixing it. When the program surprises them, your student writes an Expected/Actual line before touching the code (BUG-01): what I expected, what actually happened. Then they read the error message, if there is one, and change one thing at a time, running after each change. One short line goes in the Debug Log (BUG-08): the bug, the clue, the fix.

A Debug Log entry is evidence of good work, not a confession. Chapter 3 is all about debugging.

5. Save and Explain

Save a new version with a name a stranger could read, date first, such as 2026-10-12-guess-game-v2.py, in two places. Screens off. Then your student says (at 11–12) or writes (at 13–14) the exit lines:

“I predicted... It did... Next I'd change...”

“Help I used: ...”

The second line 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. Said every Hour, honesty becomes ordinary.

Who runs it

11–12. You run Unplug and Plan for the first month: you bring the plan or the card and ask the opening. Your student runs Predict through Save and Explain from week one. By midyear your student writes the plan, and you check it.

13–14. Your student is the session lead. They write the plan, run the clock, keep the Debug Log, and call the save. You are the first user, the question-asker, and the keeper of the safety rules. Once a week, take five minutes to read the Debug Log together.

Place by skill, not birthday. A 14-year-old who is new to all of this may want a month with you running the plan. In the chapters, “ready 12” marks a 13–14 move some 12-year-olds can take.

The rescue order

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

Ask a real question. Wait. Offer a hint that points without solving: “Look at line 4.” If a demo is truly needed, show it on your own paper or file, then close it and hand the turn back. Your hands stay off your student’s keyboard. A program they fixed is theirs.

The talk-box shape

Every teaching chapter has a talk box. The shape never changes.

The opening question is real. It asks about your student’s own plan, prediction, evidence, or judgment, and it has no answer to recite. Each chapter gives its exact opening.

Three to five follow-ups, drawn from these:

- prediction: “What will it do?”
- evidence: “What did you see that makes you say that?”
- location: “Which line is the last one that did what you expected?”
- change: “What’s the one thing you’d change?”
- revision: “You can change your prediction. Run it again.”

How to wait. After you ask, count to five in your head. The habit comes from classroom studies of “wait time,” used here as a pause, not a stopwatch.⁵ 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. The question was vague. There was no wait. Your student fears being judged, which runs strong at this age, so keep every comment about the code. They’re waiting for you to fix it, because you have before. There was no prediction to compare. Or the screen is louder than the question; lower the lid for a moment.

One “don’t say / say instead” line in every chapter. For example: don’t say “That’s wrong”; say “Show me the line where it stops doing what you expected.”

The try-it shapes

Every named try-it gives you six things: **time**, **materials**, **safety and privacy**, **the fun**, **the skill**, and **the crumb** (what to save). Each heading carries a bank code, such as *In-home — PLAN-01 The*

Literal Sandwich Spec or *Out-of-home* — *OUT-02 Library Computer Hour*. Each teaching chapter has at least one of each.

OUT-02 is worth knowing now. A whole Computing Hour can run on a public library computer, with you beside your student, the file saved to your own USB stick, and both of you logged out before you leave. A home with no computer can do about half of this book's activities on paper and the rest at the library.

When the Hour goes wrong

If the Hour looks like...	Try this
The computer won't start, or the internet is down	Thonny, the book's home Python editor, runs offline. If the machine itself won't start, make it a paper day: trace the program by hand.
An error nobody understands	Expected/Actual first. Read the error's last line aloud, then find the line number. Log it.
"I'm deleting it and starting over"	Tomorrow is allowed.
Racing ahead and skipping Predict	"Save this as a new version first. Then tell me the one thing that's wrong."
Tears, or a slammed laptop	No run without a written prediction line. Hands on the table until it's there.
A sibling interrupts or takes over	Stand up, get water, come back in five minutes.
You're itching to type	Stop while it's still fixable.
	Driver and navigator, swapping every ten minutes, or an unplugged card for the sibling.
	Hands in your lap. Point with words: "Look near line 6." Demo on your own copy if you must.

Family safety basics

These are ordinary and non-negotiable. Chapters point back here; Chapter 7 teaches them in depth.

- **You hold the keys.** At 11–12 you create and hold every login, and your student never signs up for anything alone. At 13–14 your student may hold passwords for tools the family has approved. You keep a sealed copy or a family password-manager entry, and two-step login (a password plus a second thing, such as a code) is turned on.
- **The 13 shift.** A federal law called COPPA requires sites and apps that collect personal information from children under 13 to get a parent's consent first, and photos, videos, and voice recordings count as personal information.⁴ That protection ends at 13, a big reason so many services set 13 as their minimum age. Thirteen is a rule for companies, not a readiness test. Your family's rules sit on top.

- **Shared room, phone-free bedrooms.** The computer lives in a shared space. Phones and laptops stay out of bedrooms overnight, and screens go off an hour before bed.
- **Stop, Save, Tell.** If something online is scary, mean, or threatening: stop replying, save it instead of deleting it, and tell a parent. Nobody is in trouble for telling.
- **First names only.** No school, town, or face in anything shared beyond the family. Screenshots are cropped to the work.
- **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.
- **Updates on, unknown downloads off.** At 11–12 you approve every new app. Publishing anything online is optional, and it's your call.
- **Old hardware.** Adults open old machines, unplugged. Nobody opens batteries, power supplies, or old tube screens.

A note on screen time

The American Academy of Pediatrics replaced its older media guidance in 2026. Its policy statement puts the weight on content and on what screens push out: sleep, meals, movement, and homework. It suggests phone-free meals, bedrooms, the hour before bed, and homework; parent-set controls for privacy, downloads, content, purchases, and time; and a family-tailored range of roughly one to two hours or more a day of entertainment media for school-age children and teens, with school media counted separately.⁶

The World Health Organization asks families to limit recreational screen time and to keep children and teens moving at least an hour a day. It gives no number for screens.⁷ On sleep, the CDC points to the sleep-medicine recommendations: 9 to 12 hours in every 24 for children 6 to 12, and 8 to 10 hours for teens 13 to 18.⁸ A late-night bug goes in the Debug Log and waits for morning.

These are guidelines for families, not a stopwatch rule, and they change as the research does. The Computing Hour is purposeful school media. About half its work is unplugged, and it ends with screens off.

The AI box

What AI is here for. AI can explain, suggest, and check, sometimes well and sometimes confidently wrong. Your student does the planning, predicting, writing, testing, and debugging. Anything an AI produces is made by a tool, never your student's own work. Chapter 8 explains how it works.

Ages 11–12: parent-operated. Most consumer AI services set 13 as their minimum age, and at least one major one sets 18.^{9,10} So at 11–12, AI runs on your account, on your device, with you at the keyboard or reading every exchange. Your student can dictate a question, read the answer beside you, and check it against a source. No student accounts on chatbots, AI tutors, or coding assistants, even inside a course that offers one. CodeAI (formerly Code.org), for example, labels its AI Tutor for students 13 and up, so in this book it stays off at 11–12.^{11,12}

Ages 13–14: supervised, under the terms, with you. Use only a service whose terms allow your student’s age, and a teen or linked account where the service offers one. Sit nearby with the screen visible. Turn off training on your student’s conversations, and turn off memory, where you can. Leave image generation off unless a lesson needs it. No companion, “friend,” or role-play apps. Teen settings help; they are not a substitute for you in the room. Check the current terms before you start.

You may, within the rule for your student’s age:

- have it explain an idea or an error message *after* your student has written Expected/Actual and a guess;
- ask for practice problems or a hint ladder, held back until your student has tried;
- at 13–14, get a code suggestion, then have your student explain every line before deciding whether to use any of it, with a “Help I used” line;
- check a claim it made against a real source.

Please keep these off the table:

- having it write your student’s program, page, or project;
- using it during the AI-free checks: Cold Trace, Debug on the Spot, and Build From a Blank;
- filing generated code, pages, pictures, or text as your student’s work, or as a portfolio piece;
- pasting private messages, photos, names, or your address into a chatbot;
- uploading photos of your children or family to any AI tool;
- companion, “friend,” or role-play apps;
- an AI “detector” to judge your student’s work, because “explain it to me” is the check;
- trusting a source it cites before you’ve opened it yourself.

Whose work is it? Your student’s work is what they planned, wrote, tested, and can explain. If they can’t explain a line, it isn’t theirs yet. Every project carries a Whose Work Is It? card (AI-06): who or what helped, what was asked, what came back, what was kept, what was changed, and how it was tested.

One caution from research, with a label. In a study of high-school math students, those given an unrestricted chatbot did better on practice problems and worse on a later test taken without it; a version with guardrails avoided most of the harm.¹³ That is high-school math, not 11–14 or code. The rule it supports is the one this book already uses: your student plans, predicts, and debugs first.

The Grok Bot caution. 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.¹⁴

Dose honesty

There is no national table of how many minutes a middle-schooler should code. The standards many schools use imagine the equivalent of a yearlong course across grades 6–8, a school design rather than a home rule.¹⁵ Two or three Computing Hours a week across a school year is a full course of experience. On other days, ten minutes is plenty: a Paper Parsons, a Cold Trace, a typing warm-up. Less is still real. Short and frequent beats long and rare.

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, without help, dated. Finish a project every four to six weeks. Hold a demo day once a semester for family or a co-op. When the habit leaks out of the Hour, notice it and say so in a sentence.

Five things to say

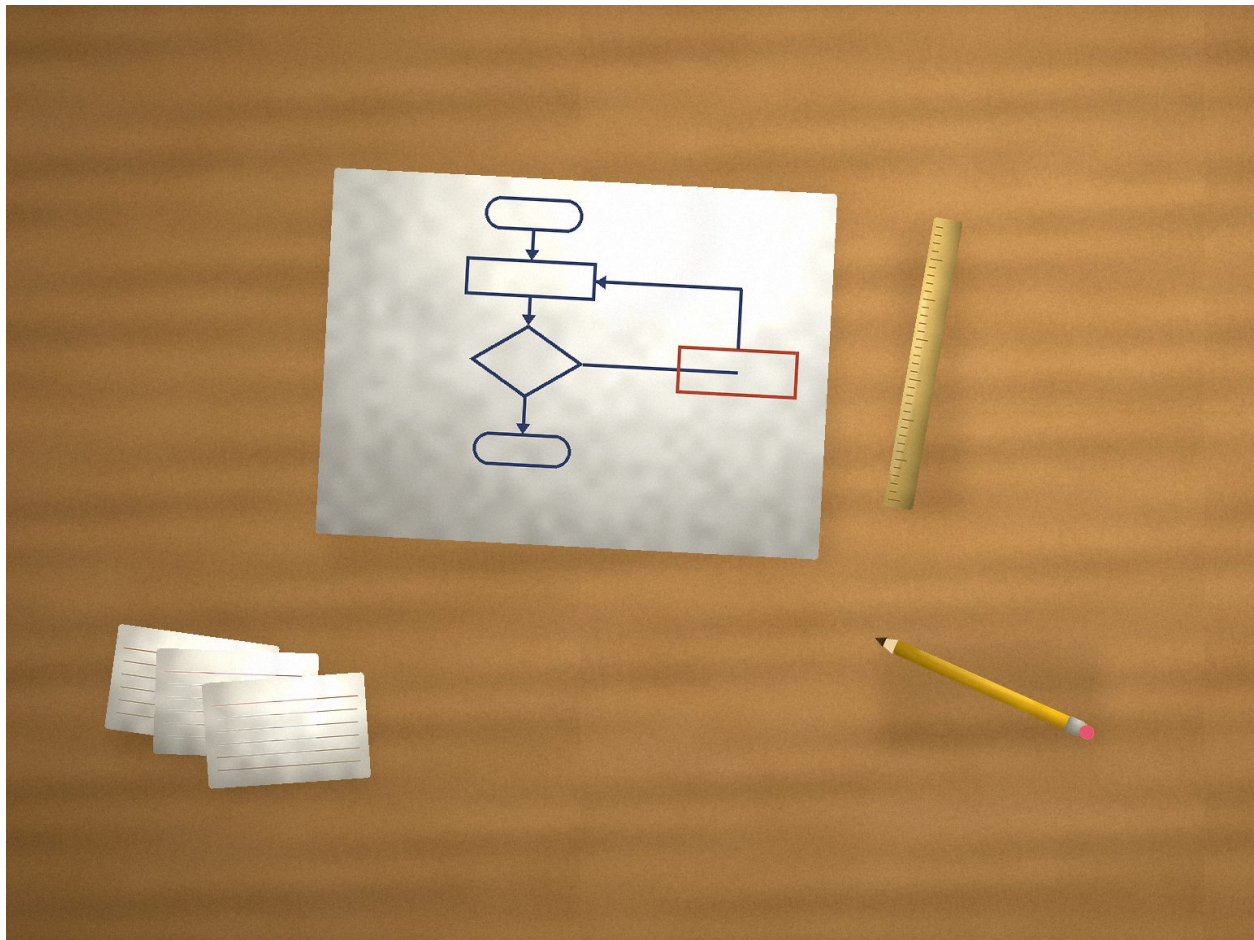
1. **Opening.** “Show me the plan. What happens first?” Then count to five in your head.
2. **Predict.** “Write the line: I predict it will... No run without it.”
3. **First user.** “I’m going to try it like a stranger would.” Then report: “It asked my name, I typed a number, and it crashed. Over to you.”
4. **The bug moment.** “What did you expect, what happened instead, and what’s your best clue?”
5. **Save and Explain.** “Save it as a new version, two places. Then tell me: ‘I predicted... It did... Next I’d change...’ and ‘Help I used: ...’ ”

Ready to begin

You have the five steps, the rescue order, the talk-box and try-it shapes, Family safety basics, the screen-time note, and the AI box. Chapter 1 puts the focus line to work. Find bread, a jar, and a pencil.

Chapter 1

What Computing Looks Like at 11–14: Plans, Algorithms, and Thinking It Through



Why this matters

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

Eli is thirteen. His father has asked him to write instructions for making a peanut butter sandwich,

and has promised to follow them exactly as written. Eli finds this beneath him. He writes four lines in about a minute: “Get the bread. Put peanut butter on it. Put the other slice on top. Done.”

His father picks up the whole loaf, still in its bag, and sets it on the counter. Then he picks up the jar of peanut butter and places it, lid on, on top of the loaf. He looks up. “Done.”

Eli laughs, and then he starts to argue. “You know what I meant.” His father nods. “I do. A computer won’t.”

Twenty minutes later Eli has a page of instructions that open the bag, take out exactly two slices, twist the lid counterclockwise, and use the knife. His father asks one more question: “What if the jar is already open?” Eli adds a line that starts with IF.

That is computing at this age. Computing is telling a machine exactly what to do, and then finding out, calmly, why it did something else. A computer never guesses what you meant. It does what you said, in the order you said it, every time.

Eleven- to fourteen-year-olds are ready for this in a new way. They can hold a plan in their heads, write it down, test it, and defend it. They can imagine what might go wrong before it does. They are also ready to make the jump from *using* computers to *making* them do things. That jump is what this book is about.

It helps to know what computing is not. The organization that writes the computer science standards many American schools use says plainly that computer science is not “using technology tools like word processors, slide decks, or generative AI tools.” It adds, about AI, “CS is what powers AI.”¹⁶ A student who can type a fast essay or get an answer from a chatbot is a skilled user. A student who can write the steps, predict what they’ll do, and fix them when they go wrong is starting to do computing.

This chapter lays the foundation, and almost none of it needs a screen. Your student will write exact instructions, break big jobs into small ones, draw flowcharts, trace programs by hand, and compare two ways of doing a job. Every later chapter stands on these habits, and on the line the whole book runs on:

Predict before you run; debug before you rewrite; explain it before you call it yours.

For the parent: understand it yourself

You don’t need to know a programming language to teach this chapter. You need a handful of ideas in plain words, and the patience to play a very literal computer.

The big ideas, in plain words

People sometimes call this whole set of habits computational thinking. In this book we’ll just say what it is: breaking a job into parts, spotting patterns, writing exact steps, and checking them.

- **An algorithm** is a set of exact steps for doing a job. A recipe is an algorithm. So is the route you drive to the store. In computing, the steps have to be exact enough that something with

no judgment at all could follow them.

- **Decomposition** means breaking a big job into smaller jobs. “Make dinner” is too big to program. “Boil water,” “cook pasta,” and “heat sauce” are getting closer.
- **Pseudocode** is a plan for a program written in plain words, with a few capitalized keywords such as SET, IF, and REPEAT. It isn’t any real programming language, and that’s the point: your student can think about the steps without worrying about brackets.
- **A flowchart** is the same plan as a picture. Ovals mark the start and end, rectangles are actions, and diamonds are decisions with a “yes” arrow and a “no” arrow.
- **A decision** (IF) lets a program do different things in different situations. **A loop** (REPEAT) does the same steps more than once.
- **A variable** is a labeled box that holds a value, such as a score, and the value can change as the program runs.
- **Tracing** means following a program by hand, line by line, and writing down what’s in each box after every step.

The middle-school standards ask students to model an algorithm “with a flowchart or pseudocode” and to “verify the accuracy of an algorithm for given inputs.”¹⁷ In ordinary words: write the plan, then check that it works on real examples. That’s this chapter.

A program you can trace together

Here is a short program in pseudocode:

```
SET total TO 0
SET n TO 1
REPEAT 4 TIMES
  ADD n TO total
  ADD 1 TO n
SAY total
```

Trace it with your student on paper, in a table with one column for `total` and one for `n`. After each trip through the loop, write the new values. The first trip makes `total` 1 and `n` 2. Then 3 and 3, then 6 and 4, then 10 and 5. The program says **10**.

Plenty of adults guess wrong the first time. That’s normal, and it’s exactly why tracing matters. Your student will never trace a program perfectly by looking at it. Writing down each box after each step is how people who program for a living check their own thinking.

Why paper first

Paper is not a lesser kind of computing. With the screen dark, your student has to hold the whole plan in mind instead of trying random changes to see what sticks. Paper is also where planning mistakes are cheapest. A wrong step in a sandwich spec costs a laugh. The same wrong step in a two-hundred-line program costs an evening.

And on paper, you can see your student’s thinking. On a screen, you mostly see the result.

The student you have

Young teens are building more capacity for complex thought and a stronger sense of right and wrong. They make more of their own choices and become more independent, although parents are still very important.¹⁸ That's the practical case for this book's shape: your student writes more of the plan and runs more of the Hour, and you become the first user, the question-asker, and the keeper of the safety rules.

Young teens are also more sensitive to being judged by others than younger children or adults.¹⁹ You'll see it when a bug feels like a verdict on the person. Keep every comment about the code: "Line 3 never runs when the jar is open" lands very differently from "You forgot again." Let your student choose what gets shown to anyone outside the family.

One more thing about the student you have. Children pick up the idea that computing is "for" some kids and not others as early as first grade.²⁰ Say plainly, and more than once, that computing is for anyone who likes making things work and figuring out why they don't.

What the evidence says, honestly

Coding teaches coding. That much is solid. Whether it also helps with math, reading, or general problem-solving is a fair question, and the honest answer is "sometimes, a little, and less than people claim." A large review of studies found that learning to program helps most with skills close to programming, helps less with skills further away, and shows smaller effects when students are compared with others doing a different active task.²¹ So treat any spillover as a welcome possibility, not a promise.

The good news about this age is solid too. In one carefully designed middle-school course, seventh and eighth graders learned real computer science concepts and carried some of them over to a text programming language, though loops and variables stayed hard for many.²² This is a useful study, not a promise that every home will see the same result. It does match what you'll see at your table: this age can do real work, and some of it will take longer than you expect.

Fast thumbs are not the same thing as exact steps. Most 11- to 14-year-olds have never been asked to write a plan and test it, and that's the skill this book teaches.

Why the third clause matters

The focus line ends with "explain it before you call it yours." That isn't a homeschool invention. The guidance for one widely taken high-school computer science course says students who use AI tools must be ready "to explain their code in detail."²³ At 11–14 this starts small and spoken: "Help I used: none." Chapter 8 goes further.

Wrong answers you should be able to hear

What you hear	What it usually means	What you can say
“You know what I meant.”	The steps leave something to guessing.	“I do. The computer won’t. Which line needs to be more exact?”
“It’s obvious.”	A hidden step nobody wrote down.	“Then it’ll be quick to write. Show me where it goes.”
“I’ll just start over.”	Overwhelm.	“Keep this page. Tell me the one thing that’s wrong.”
“It works, so it’s right.”	One example tested, no others.	“What’s the strangest input you can think of? Let’s try it.”
“It says 4.” (for the program above)	Guessing instead of tracing.	“Let’s write the boxes after each trip around the loop.”

How to teach it this week

This week’s Hours run on the five steps from **The Computing Hour**. Here is how they fit the first activity:

1. **Unplug and Plan.** Your student writes the sandwich spec on paper. Ask the talk-box opening.
2. **Predict.** Your student writes one line: “I predict the sandwich will...”
3. **Build and Test.** You follow the spec exactly, one line at a time.
4. **Debug and Log.** Your student writes an Expected/Actual line (BUG-01), fixes one step, and you run the spec again from the top. Start the Debug Log (BUG-08) with the sandwich bugs: the bug, the clue, the fix.
5. **Save and Explain.** Photograph the marked-up spec. “I predicted... It did... Next I’d change...” and “Help I used: none.”

Exact words you can say

- Opening: “What are the steps — and where could they go wrong?”
- When a step is vague: “Computer does not understand. Which bread? How many slices?”
- After a funny result: “What did you expect? What happened instead?”
- At the fix: “Change one line. Then we start again from the top.”
- An edge case: “What if the jar is already open?”
- At the end: “Save it. Then tell me: ‘I predicted... It did... Next I’d change...’ and ‘Help I used: ...’”

In-home — PLAN-01 The Literal Sandwich Spec

- **Time:** about 20 minutes.
- **Materials:** bread, a spread (or a pretend version: a box for the loaf, a jar of anything), a butter knife, paper, and two colors of pen.

- **Safety and privacy:** use a butter knife, and check for food allergies in the room. Any photo is of the paper only.
- **The fun:** a parent who follows instructions with total, deadpan literalness.
- **The skill:** writing precise instructions, and thinking of the cases that could break them.
- **The crumb:** the marked-up spec, with the fixes in a second color.

How it goes. Your student writes the spec first, without you watching. Then you follow it line by line, exactly as written, and nothing more. “Put peanut butter on the bread” with no mention of the lid or the knife gets an unopened jar resting on the loaf. Keep a straight face. When it goes wrong, stop and let your student find the first line that failed, fix it in the second color, and run the whole spec again from the beginning.

The 11–14 twist is the edge case. Once the spec works, ask: “What if the jar is already open? What if there’s only one slice left?” A good spec handles the ordinary case. A better one says what to do when things aren’t ordinary.

11–12: your student can say the steps aloud first while you write them, then copy them out. You play the computer.

13–14: your student writes the spec alone, then plays the computer for a younger sibling’s spec, or for yours. Try to write a spec they can’t break. They will break it.

In-home — PLAN-03 Pseudocode Recipe Card

- **Time:** about 15 minutes.
- **Materials:** an index card, a pencil, and a simple family recipe.
- **Safety and privacy:** paper only; no cooking needed.
- **The fun:** turning grandma’s pancakes into something a robot could follow.
- **The skill:** plain-language pseudocode with IF and REPEAT.
- **The crumb:** the card.

How it goes. Pick a recipe and rewrite it in pseudocode, with at least one decision and one loop:

```
SET cups_of_flour TO 2
REPEAT cups_of_flour TIMES
  ADD 1 cup of flour TO bowl
ADD milk TO bowl
STIR
IF batter is lumpy THEN STIR again
```

Then swap: you read the card aloud as a literal computer, and your student listens for gaps. Where does the butter go? How long is “stir”?

11–12: one decision and one loop is plenty. **13–14:** add a decision with two outcomes (IF ... ELSE), or a loop that repeats until something happens.

In-home — PLAN-04 Flowchart the Morning

- **Time:** about 20 minutes.

- **Materials:** a large sheet of paper, a pencil, and an eraser.
- **Safety and privacy:** first names only if it's posted on the fridge.
- **The fun:** discovering how many decisions happen before breakfast.
- **The skill:** a flowchart with decision diamonds and a loop.
- **The crumb:** the flowchart.

How it goes. Your student draws the morning routine as a flowchart, from “alarm rings” to “out the door.” Every decision is a diamond with exactly one question and two arrows, yes and no: “Is it a school day?” “Is the bathroom free?” At least one arrow should loop back: “Snooze pressed? Yes → back to alarm rings.” Then you follow the flowchart with your finger, out loud, and your student checks whether it really describes their morning.

The usual trouble is a diamond with three questions in it. One question per diamond. If a diamond needs “and,” it's two diamonds.

In-home — ALG-01 Guess My Number

- **Time:** about 10 minutes.
- **Materials:** paper and a pencil for keeping count.
- **Safety and privacy:** none needed. It works in the car.
- **The fun:** beating a parent at their own game, in very few guesses.
- **The skill:** searching by halving, and counting the steps.
- **The crumb:** the guess counts for 1–100 and for 1–1,000.

How it goes. You pick a secret number from 1 to 100. Your student guesses, and you say only “higher,” “lower,” or “yes.” Count the guesses. Play a few rounds, then ask: “What’s the best first guess? Why?”

The best plan is to guess the middle every time, which throws away half of the remaining numbers with each guess. Played that way, any number from 1 to 100 takes at most 7 guesses. Here’s the surprise: 1 to 1,000 takes at most 10. Ten times as many numbers, only three more guesses. Let your student discover that by playing before you explain it. Chapter 2 turns this game into a real program.

13–14: ask your student to explain why each guess halves the problem, and to predict the most guesses needed for 1 to 2,000 before testing it.

Talk box (this chapter)

Opening: “What are the steps — and where could they go wrong?”

Follow-ups:

1. Prediction: “What will happen at this step, exactly as it’s written?”
2. Evidence: “What did you see that makes you say it worked?”
3. Location: “Which step is the last one that did what you expected?”
4. Change: “What’s the one line you’d change?”
5. Edge case: “What’s the strangest situation this plan might meet?”

How to wait. After you ask, count to five in your head. When the question is about code on the screen, let them read as long as they need. This chapter mostly lives on paper, so look at the page, not at your student. Your eyes on the plan say “we’re examining the steps,” not “I’m examining you.”

What a stuck silence usually means. The question was too big (“Why didn’t it work?”), so shrink it: “Was it this step or that one?” Your student may be bracing for a verdict, so name a bug of your own first. They may be waiting for you to fix it, so put the pencil down on your side of the table. Or there was no prediction, so there’s nothing to compare. Go back and make one.

Don’t say / say instead. Don’t say “That’s not how you make a sandwich.” Say “I did exactly what line 2 says. Is that what you wanted?”

How to fade help

Use the rescue order from **The Computing Hour**: ask → wait → hint → short demo on your own copy → hand it back.

- **First Hour:** you may point to the step that broke: “Look at line 2.”
- **Second Hour:** ask only where it first went wrong.
- **By week’s end:** ask the opening question and let your student run the fix.

When you notice good work, name it exactly. “You added a line for when the jar is already open, before I even asked” teaches more than “Great job.”

This week at a glance

Day	Focus	Try-it
Hour 1	Exact steps and edge cases	PLAN-01 The Literal Sandwich Spec; start the Debug Log
Hour 2	Searching by halving	ALG-01 Guess My Number
Hour 3	Plans on paper	PLAN-03 Pseudocode Recipe Card or PLAN-04 Flowchart the Morning
Other days	Ten minutes	PLAN-07 Paper Parsons; a trace table
Errand	Algorithms around you	OUT-01 Algorithms in the Wild
End of week	A baseline	AS-01 Cold Trace (see “done enough”)

Practice that actually builds learning

Pick one or two of these per Hour over the next few weeks. Each is short, and most need nothing but paper.

In-home — PLAN-02 Break It Down Tree

- **Time:** about 15 minutes.
- **Materials:** a large sheet of paper and a pencil.
- **Safety and privacy:** none needed.
- **The fun:** finding out how many tiny jobs hide inside one big one.
- **The skill:** decomposition: breaking a real job into parts a program could do.
- **The crumb:** the tree, dated.

How it goes. Write a big job at the top: “Host a family game night” or “Make tacos for six.” Your student breaks it into three or four smaller jobs, then breaks each of those again, until every bottom box is something a person, or a program, could just do. “Choose a game” becomes “list the games we own,” “count the players,” and “cross off games for fewer players.” Stop when the boxes are small enough to act on.

13–14: circle the boxes a computer could actually handle (counting, sorting, checking a list), and the ones that need a person.

In-home — PLAN-05 Trace Table by Hand

- **Time:** about 15 minutes.
- **Materials:** paper ruled into columns, and a short program in pseudocode.
- **Safety and privacy:** none needed.
- **The fun:** being the computer, and catching your own wrong guess.
- **The skill:** tracing variable values line by line.
- **The crumb:** the trace table.

How it goes. Use the total and n program from “For the parent.” Your student writes a prediction first, then traces it with one row per trip around the loop, and compares. Then change one line, such as REPEAT 5 TIMES, and predict again before tracing. The habit to build is predicting first, so a wrong guess becomes a clue instead of an embarrassment.

In-home — PLAN-06 Two Algorithms, One Job

For 13–14; ready 12.

- **Time:** about 20 minutes.
- **Materials:** paper, a pencil, and a calculator for checking.
- **Safety and privacy:** none needed.
- **The fun:** finding a shortcut that does a hundred steps’ work in one.
- **The skill:** comparing two algorithms by counting their steps.
- **The crumb:** the two step counts, side by side.

How it goes. The job: add up every whole number from 1 to 100. Algorithm A adds them one at a time: $1 + 2 = 3$, $3 + 3 = 6$, and so on. Count the additions: 99. Algorithm B pairs the numbers from the ends: 1 and 100 make 101, 2 and 99 make 101, all the way to 50 and 51. That's 50 pairs of 101, so the answer is $50 \times 101 = 5,050$, in one multiplication. Both get the same answer. One takes far fewer steps. That's what programmers mean by *efficient*.

Ask your student to predict how the step counts change for 1 to 1,000.

In-home — PLAN-07 Paper Parsons

- **Time:** about 10 minutes.
- **Materials:** a short program printed or written out, cut into strips, one line per strip; tape.
- **Safety and privacy:** use scissors with care.
- **The fun:** a puzzle where every piece is a line of code.
- **The skill:** reading a program's order and structure.
- **The crumb:** the taped strip, in the right order.

How it goes. Cut up a short program, such as the pancake card or this one, mix the strips, and have your student put them in order:

```
SET coins TO 0
REPEAT 5 TIMES
  ADD 3 TO coins
IF coins > 10 THEN SAY "Enough for a snack"
```

Indented strips stay indented, so your student can see which lines belong inside the loop. Putting lines in order is a gentler start than writing from a blank page. One study, with university students, found that this kind of puzzle took less time than writing or fixing the same code, with similar learning.²⁴ It is not a study of 11- to 14-year-olds, but it explains why Parsons puzzles turn up again in later chapters.

In-home — ALG-02 Card Sort Race

- **Time:** about 20 minutes.
- **Materials:** a deck of playing cards; paper for keeping a tally.
- **Safety and privacy:** none needed.
- **The fun:** a race where the slower method sometimes wins.
- **The skill:** two sorting methods, and counting comparisons.
- **The crumb:** the comparison tally.

How it goes. Deal eight cards face up in a row. Method one (selection sort): scan the whole row for the smallest card, move it to the front, then scan the rest for the next smallest, and so on. Method two (insertion sort): take cards one at a time from the left and slide each one back to its correct place among the cards already sorted. One player counts aloud every time two cards are compared.

Selection sort always takes 28 comparisons for eight cards, however they start. Insertion sort depends on the deal. Give it eight cards already in order, and it needs only 7. Ask your student to predict which method wins on an already-sorted row, then on a reversed one, and test both.

In-home — ALG-03 Sorting Network on the Floor

- **Time:** about 20 minutes.
- **Materials:** masking tape, four index cards with numbers on them, and four people (or four stuffed animals moved by hand).
- **Safety and privacy:** clear the floor first. If you take a photo, take it of the tape, with no faces.
- **The fun:** a sorting machine you walk through.
- **The skill:** a sorting network: comparisons that happen side by side.
- **The crumb:** a photo of the floor network, with no faces.

How it goes. Tape four parallel lanes on the floor, with three rows of meeting squares. In row one, lane 1 meets lane 2, and lane 3 meets lane 4. In row two, lane 1 meets lane 3, and lane 2 meets lane 4. In row three, lane 2 meets lane 3. Each walker holds a number card and walks forward. At a meeting square, the two walkers compare cards: the smaller number takes the left lane out, the larger takes the right. Whatever order you start in, the cards come out sorted in three rows and five comparisons.

The idea comes from CS Unplugged, a free collection of computing activities without computers, which lists sorting networks for ages 5 to 14 and has a larger version with more lanes.²⁵ **13–14:** ask why two pairs can compare at the same time. Doing many comparisons at once is one way computers speed up big jobs.

In-home — ALG-04 Battleships Search

- **Time:** about 20 minutes.
- **Materials:** two sheets of paper, each with a row of fifteen boxes; thirty small sticky notes; two pencils.
- **Safety and privacy:** none needed.
- **The fun:** a hidden-number duel.
- **The skill:** searching one by one versus searching by halving.
- **The crumb:** the score sheet with the number of looks for each round.

How it goes. Each player secretly writes fifteen numbers in their boxes and covers each with a sticky note. The other player names a target number and uncovers one box at a time to find it, counting each look. Round one: the numbers are in random order. The searcher may need all fifteen looks. Round two: the numbers are written in order from smallest to largest. Now the searcher can open the middle box and throw away half the row, just like Guess My Number. Fifteen sorted boxes take at most 4 looks. CS Unplugged has a printable version of this game.

11–12: play round one and two, and talk about the difference. **13–14:** explain why sorting first pays off only when you'll search more than once.

Out-of-home — OUT-01 Algorithms in the Wild

- **Time:** about 20 minutes during an errand or walk.
- **Materials:** a notebook and a pencil.

- **Safety and privacy:** stand well back from the curb when watching traffic lights. Watch machines; leave other people out of notes and photos. At a self-checkout, watch without using someone else's lane. See Family safety basics in **The Computing Hour**.
- **The fun:** spotting the hidden program in something you see every day.
- **The skill:** writing pseudocode for a real machine.
- **The crumb:** the pseudocode page.

How it goes. Pick one machine: a traffic light, an elevator, or a self-checkout at the store. Watch it for a few minutes and talk about what it seems to do and when. Then sit on a bench and write its algorithm in pseudocode:

```
REPEAT forever
  SHOW green FOR some seconds
  SHOW yellow FOR a few seconds
  SHOW red
  IF a car is waiting on the side road THEN let it go
```

Then hunt for edge cases. What does the elevator do if people on two floors press the button at once? What if someone scans the same item twice? Your student won't know the real program, and that's fine. The point is a plausible one, exact enough to argue about.

13–14: write a second version that handles one edge case, and say which is better and why.

Blocked, then mixed

Blocked practice for the new move. For the first week or two, every Hour includes the same core move: write exact steps, predict, and test them. Sandwich on Monday, pancakes on Wednesday, the morning on Friday.

Mixed practice. After that, mix it up. One Hour might hold a Paper Parsons, a trace table, and a round of Guess My Number. Mixing helps your student learn *when* to use each move, not just how.

One incorrect example to diagnose

Here is an illustration, not a real family. A twelve-year-old writes a flowchart for “get ready for bed.” One diamond reads: “Teeth brushed and pajamas on and homework done?” The “no” arrow points to “do the missing thing.” The parent follows it and stops: “Which missing thing?”

Diagnosis. Three questions in one diamond, so the “no” answer can't say what to do next.

Rescue. One question per diamond. Split it into three diamonds, each with its own “no” arrow to a specific action. Then follow the chart again with your finger.

For the student

This chapter is about something programmers do all day that most people never see: thinking it through before touching the keyboard.

You might picture a programmer typing fast, code flying across the screen. Real programming looks more like this: read the problem, plan, write a little, test it, find out why it didn't work, fix it, test again. A lot of the time goes into planning, reading, testing, and fixing. The typing is often the quick part.

That's why this chapter happens mostly on paper. It might feel like the baby version of coding. It isn't. Paper is where you can see the whole plan at once, and where mistakes are cheap.

A computer does what you say, not what you mean. That's the most important idea in this book. If you write "put peanut butter on the bread" and forget to say "open the jar," a computer won't open the jar. It isn't being stubborn. It has no idea what a jar is. It follows the steps, exactly, in order.

So your job is to write steps so exact that something with zero judgment could follow them. Steps like that are called an **algorithm**. Writing them in plain words with a few keywords (SET, IF, REPEAT) is called **pseudocode**, which means "pretend code." It's the plan before the program.

A tiny example. Here's a plan in pseudocode:

```
SET lives TO 3
REPEAT 2 TIMES
    SUBTRACT 1 FROM lives
IF lives > 0 THEN SAY "Keep going" ELSE SAY "Game over"
```

What will it say? Write your prediction down before you read on.

Start with 3 lives. Lose one: 2. Lose another: 1. Is 1 more than 0? Yes. So it says "Keep going." If you predicted "Game over," find the step where your thinking went a different way. Maybe you counted three trips through the loop instead of two. That's a bug in your trace, and finding it is exactly the skill.

Your Debug Log. Starting this week, you keep a Debug Log: a dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes. Before you run anything, write a prediction. When something surprises you, write two things before you change anything: what you expected, and what actually happened. Then write the clue and the fix. It takes thirty seconds. In a month, it'll show you how much you've learned, and it'll save you from fixing the same bug twice.

The line this book runs on. You'll see it everywhere: *Predict before you run; debug before you rewrite; explain it before you call it yours.* In your own words, it means:

- Guess first, so you'll know if you're right.
- When it breaks, find the broken part. Don't throw the whole thing away.
- If you can't explain it, it isn't yours yet. And always say what help you used, even if the answer is "none."

This part is hard for everyone. Writing steps exact enough for a computer is harder than it looks, for adults too. People who program for a living still write plans that miss a case. The difference is that they expect it, look for it, and fix it without drama.

Try this today. Write the instructions for something you do every day: making toast, tying a shoe, logging in to a game. Then hand them to someone and ask them to follow them exactly, with no

guessing. Watch where it goes wrong. Fix one line at a time.

Explain it back. Tell someone in two or three sentences: why does a computer need more exact steps than a person does? Use your own example.

If you're 13 or older: write a spec that someone else, maybe a younger sibling, has to follow, and then try to break theirs. Look for the edge cases: the jar that's already open, the last slice of bread, the elevator button pressed twice.

Challenge. Play Guess My Number with 1 to 100 until you can always win in 7 guesses or fewer. Then predict: how many guesses for 1 to 1,000? Test it. Can you explain why the answer is so small?

What to save. Your sandwich spec with the fixes, your first Debug Log page, and your Guess My Number counts. Date them all.

If it isn't clicking

The steps are too vague. Play the literal computer again, and be truly literal. "Put it on the plate" gets "Put what? Which plate?" Then have your student read their own spec aloud, slowly, as if they'd never seen a sandwich.

Flowchart spaghetti. Arrows cross everywhere and diamonds hold three questions. One decision per diamond. Start again on a fresh sheet with just three boxes, then add one at a time.

"This is boring. I want to code." Fair enough. Show PY-02 Predict the Print on paper today: a four-line Python program, a written prediction, and a check. It's real code, and it uses exactly the same habit. Install Thonny tomorrow (Chapter 2). Then come back to planning, and point out that the prediction they just made was a trace.

Tracing goes wrong every time. Slow down to one line at a time, and write the boxes after every line, not just at the end of each loop. Use sticky notes as boxes and physically replace the number each time.

Your student is far ahead of you. Wonderful. Your job doesn't change: ask for the prediction, be the first user, and ask for the explanation. Have your student teach you the trace.

A thirteen-year-old new to all of this. Start here anyway. The 13–14 moves go fast.

Tools, including AI

What you need: a pencil, an eraser, paper, index cards, a deck of playing cards, and masking tape. That's the whole list.

Optional: CS Unplugged (free, printable) for the sorting network and searching games.²⁵

AI: none needed for this chapter. Planning, predicting, and tracing are the student's own work, and they're the muscles everything else depends on. For the rules on AI at each age, see the AI box in **The Computing Hour**.

What “done enough” looks like

Your student is ready to move on when they can:

- write pseudocode for an everyday job that includes at least one decision (IF) and one loop (REPEAT);
- trace a five-line program on paper and predict what it will say;
- explain, in their own words, why guessing the middle finds a number so fast;
- write an Expected/Actual line before fixing something;
- say the exit lines without prompting: “I predicted... It did... Next I’d change...” and “Help I used: ...”

11–12: spoken explanations are fine; the pseudocode can be short. **13–14:** written explanations, and at least one comparison of two algorithms by counting steps.

AS-01 Cold Trace (baseline)

At the end of this chapter, do a first Cold Trace. It’s a ten-minute check your student does alone: no help from you, no screen, and no AI. Hand over this program and ask for a prediction of what it says, with a trace table:

```
SET coins TO 4
REPEAT 3 TIMES
  ADD 2 TO coins
IF coins > 8 THEN SAY "Piggy bank full" ELSE SAY "Keep saving"
```

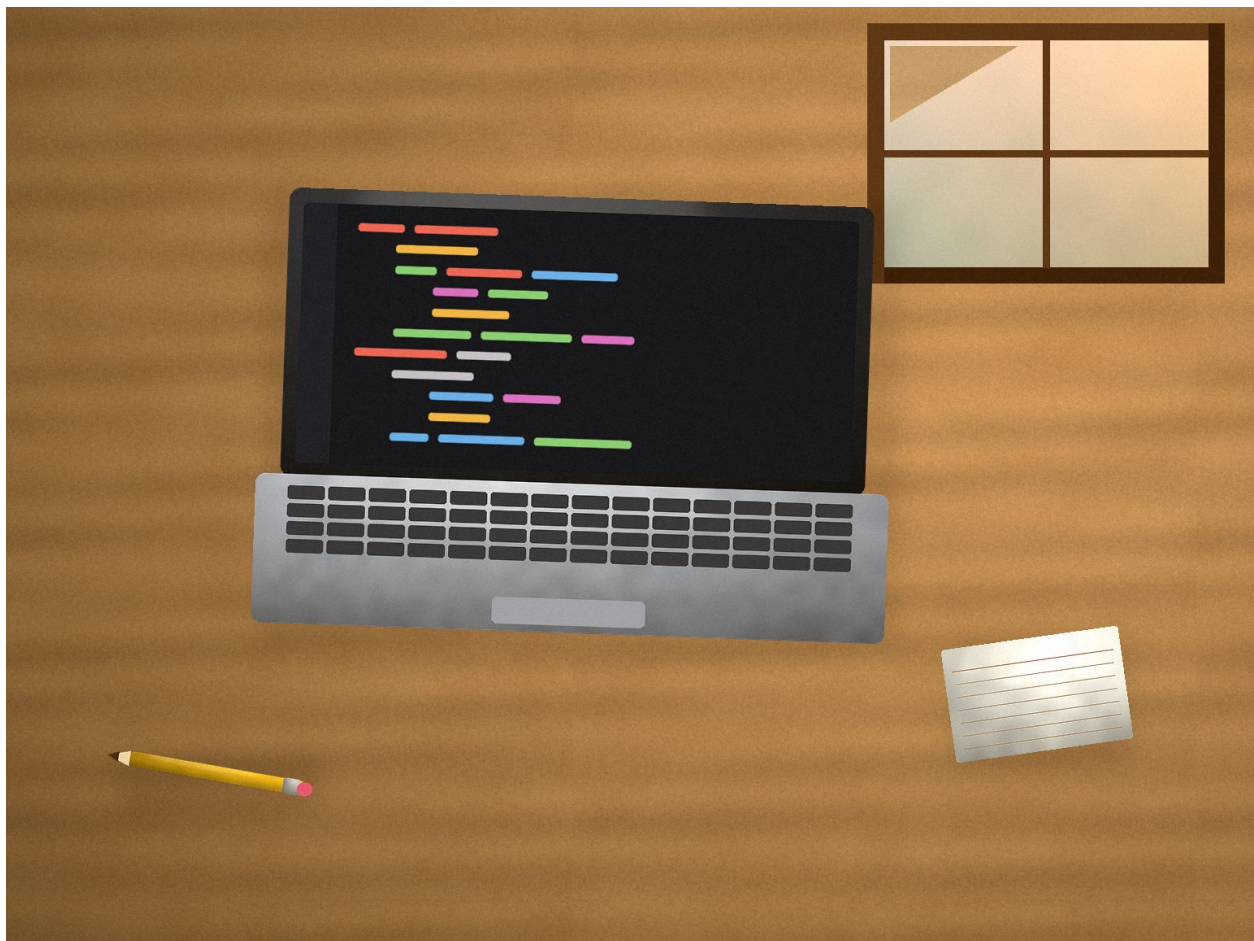
(The answer: coins goes 6, 8, 10, so it says “Piggy bank full.”)

Date the sheet and file it. Right or wrong, this is a baseline, not a grade. You’ll do a Cold Trace once a month, and by spring this first sheet will show your student how far they’ve come.

If most of the list is in place, move on to Chapter 2, where these plans become real Python. Planning never goes away. It just gets a keyboard.

Chapter 2

From Blocks to Text: First Programs in Python



Why this matters

Many students arrive at this chapter with years of block coding behind them. They've snapped together Scratch games with scores, timers, and sprites that bounce off walls. That was real programming. The ideas in those blocks (steps in order, decisions, loops, variables) are the same ideas inside every program on every computer.

Text code is the next step. Most software is written by typing lines in a programming language, and this chapter starts your student doing exactly that, in Python.

Text brings three new powers. It's precise: every character means something. It's fast: once your fingers know the way, typing a line is quicker than dragging a block. And it's readable by others: your student can read code that other people wrote, and the documentation that explains how a language works.

Text also brings a new kind of trouble. In blocks, pieces only fit where they belong. In text, a missing bracket stops everything, a capital letter in the wrong place is a different word, and the computer replies in red. This chapter is about making that move with no drama and no loss of confidence. Red text is information, not judgment. Your student will learn to read it, find the line it names, change one thing, and run it again.

The habit that makes it all work is the first clause of the focus line, now with real code: predict before you run.

For the parent: understand it yourself

You can teach this chapter without being a programmer. You need to know what a handful of Python lines do, well enough to read them aloud and ask good questions. Here they are.

Why Python

Python is a programming language that reads more like English than most. Its makers describe it as “friendly & easy to learn” and open, and it's used for web development, science, education, and games.²⁶ It's free. It's also the language of the home editor this book recommends. Your student won't stay with Python forever, and doesn't need to. The habits learned here move to any language.

Five ideas, one at a time

Printing and running. `print()` shows something on the screen. Running a program means telling the computer to follow its lines from top to bottom.

```
print("Hello, world!")
```

This prints `Hello, world!` The quotation marks tell Python it's text, not an instruction.

Variables. A variable is a labeled box whose contents can be replaced. The equals sign in Python doesn't mean “is equal to forever.” It means “put this value in this box now.”

```
snack = "apple"
count = 2
count = count + 1
snack = "pretzel"
print(count, snack)
```

This prints 3 pretzel. The box called count held 2, then 3. The box called snack held “apple,” then “pretzel.” The old contents are gone.

Types: text versus numbers. Python treats “12” (text, in quotes) and 12 (a number) as different things. You can do arithmetic with a number, not with text. This matters most with `input()`, which asks the person at the keyboard to type something. Python’s documentation says it “reads a line from input, converts it to a string,” which is programmer talk for text, “and returns that.”²⁷ So a typed age arrives as text and has to be turned into a number with `int()`:

```
name = input("What is your first name? ")
age = input("How old are you? ")
next_year = int(age) + 1
print("Hi,", name)
print("Next year you will be", next_year)
```

If your student types Sam and 12, this prints Hi, Sam and Next year you will be 13. Leave out `int()` and write `age + 1`, and Python stops with a `TypeError`: it can’t add a number to text. That error is one of the most common in the first month, and now you’ll recognize it.

Decisions: if, elif, else. `if` checks whether something is true. `elif` (short for “else if”) checks another possibility. `else` catches everything left over.

```
answer = input("Left or right? ")
if answer == "left":
    print("You find a quiet library.")
elif answer == "right":
    print("You find a noisy market.")
else:
    print("You stand still. Nothing happens.")
```

Two equals signs (`==`) mean “is this equal?” One equals sign means “put in the box.” Type `left` and you find the library. Type `Left`, with a capital, and you stand still, because to Python those are different words. The indented lines, four spaces in, belong to the line above them. In Python, indentation is part of the meaning.

Loops: for and while. A `for` loop repeats a set number of times:

```
for number in range(1, 6):
    print(number * 3)
```

This prints 3, 6, 9, 12, and 15, each on its own line. `range(1, 6)` counts from 1 and stops *before* 6, a detail that trips up nearly everyone at first. A `while` loop repeats as long as something stays true, which is perfect for games that go on until the player wins. You’ll see one in the guess-the-number game below.

Blocks to text: what the research says

Studies of block and text programming have mostly been done with high-school students, so read them as a picture of older beginners. In one, students who started with blocks learned more in the first weeks and were more interested in further computing courses, while students who started

with text felt their work was closer to what professional programmers do. A follow-up found the difference had disappeared after the students spent ten weeks in a professional text language.^{28,29}

The practical lesson: blocks were never babyish, and starting text now leaves nobody behind. Scratch is designed especially for ages 8 to 16, so it's fine to keep it around for quick sketches and fun.³⁰ This chapter uses Scratch on purpose in the first weeks, side by side with Python, so your student can see that the ideas they already know have new clothes.

How to teach code: read before you write

The method this book uses is called PRIMM: **P**redict, **R**un, **I**nvestigate, **M**odify, **M**ake. Your student reads a short working program first and predicts what it will do. Then they run it and compare. Then they investigate (“What happens if I change this number?”), modify it to do something a little different, and finally make something of their own. A large study in schools found that students aged about 11 to 14 who learned this way did a little better on a programming test than students who didn't.³¹ It is a small edge, not a miracle, and it matches the Hour's shape.

The weekly arc is the same idea in three words: **Use, Modify, Create**. Use a working program, change one part, then build your own. It grew out of programs for middle- and high-school students.³² It also takes the terror out of the blank page.

The standards for this age ask students to use variables of different types, to analyze how loops, decisions, and variables work in a piece of code, and to “use reference documentation in program development.”¹⁷ PRIMM and Use-Modify-Create cover all three.

Installing Thonny

Thonny is the book's home editor for Python. It's free, it comes with Python built in, it needs no account, and it runs offline once installed. It has a step-by-step mode that runs a program one piece at a time, and a Variables view that shows what's in every box.² Install it yourself before your student does, so you've seen the screens. The top half is where you write the program. The bottom half, called the Shell, is where the output and any red error messages appear.

Wrong answers you should be able to hear

What you hear	What it usually means	What you can say
“The computer is broken.”	Expected and actual differ, and there's no clue yet.	“It did exactly what we told it. Which line is the last one that did what you expected?”
“x = 5 means x equals 5 forever.”	Mixing up “put in the box” with “is equal.”	“It's a box. What's in the box after line 4?”
“I typed it exactly right.”	A capital letter, a missing bracket, or a missing colon.	“Read it to me one character at a time.”
“It says error, so it's all wrong.”	One error stops the run, but most lines are fine.	“Which line does it name? Let's start there.”

What you hear	What it usually means	What you can say
“I’ll just start over.”	Overwhelm.	“Save this as a new version first. Then tell me the one thing that’s wrong.”

How to teach it this week

This week’s Hours run on the five steps from **The Computing Hour**, with Python on the screen. Here’s how a typical Hour fits:

1. **Unplug and Plan.** Read the day’s program on paper, or translate a Scratch script on paper. Ask the talk-box opening.
2. **Predict.** Your student writes a prediction line in the Debug Log for every program before it runs.
3. **Build and Test.** Your student types and runs the program in Thonny, then changes one thing at a time.
4. **Debug and Log.** When it surprises them: an Expected/Actual line first (BUG-01), then read the error’s last line and the line number it names, change one thing, and run again. One line in the Debug Log (BUG-08).
5. **Save and Explain.** A new version, date-first name, in two places. “I predicted... It did... Next I’d change...” and “Help I used: ...” From this chapter on, “Help I used” is a habit, not an occasional question.

Exact words you can say

- Opening: “What do you predict this line will do — and why?”
- When the screen goes red: “Read me the last line. What line number does it name?”
- When your student wants to change three things: “Pick one. Run it. Then we’ll see.”
- When it works: “Before we move on, what did line 3 do?”
- At the end: “Save it, two places. Then tell me: ‘I predicted... It did... Next I’d change...’ and ‘Help I used: ...’”

In-home — BLK-01 Scratch Side-by-Side

- **Time:** about 20 minutes.
- **Materials:** Scratch open on the screen (no account needed to make a project), paper, and Thonny.
- **Safety and privacy:** no sharing projects online; that’s a family decision for later.
- **The fun:** discovering that you already know Python’s ideas.
- **The skill:** matching each block to a line of Python.
- **The crumb:** a two-column sheet: blocks on the left, Python on the right.

How it goes. Build this tiny script in Scratch: *when green flag clicked, set score to 0, repeat 3, change score by 1*, and after the loop, *say score*. Run it. The sprite says 3. Now write the Python

beside it:

```
score = 0
for i in range(3):
    score = score + 1
print(score)
```

Predict, then run it in Thonny. It prints 3. Draw a line from each block to its Python line. Notice what Python needs that Scratch didn't: a colon at the end of the `for` line, and four spaces before the line inside the loop. Those are the jobs the C-shaped block was doing.

11–12: do this twice more with your student's own small scripts. **13–14:** one is enough; move on to BLK-03.

In-home — PY-01 Hello, Editor

- **Time:** about 20 minutes.
- **Materials:** a computer with Thonny installed.
- **Safety and privacy:** files go in a family folder, with a first name only in any file name.
- **The fun:** making the computer say something you chose.
- **The skill:** opening the editor, `print()`, running, and saving with a name a stranger could read.
- **The crumb:** the first file, named `date first`.

How it goes. If you did this in your first week (How to Use This Book This Week), do it again with a twist. Your student types `print("Hello, world!")`, predicts, and presses Run. Then three more print lines, saying anything they like. Save it with the date first, such as `2026-10-12-hello.py`, in two places. Then make a deliberate mistake: delete the closing bracket and run it. Read the red text together, find the line number, and fix it. Your student has now caused, read, and fixed an error on purpose, on day one.

In-home — PY-02 Predict the Print

- **Time:** about 15 minutes.
- **Materials:** a short program on paper; the Debug Log; Thonny.
- **Safety and privacy:** none needed.
- **The fun:** betting against the computer, and sometimes winning.
- **The skill:** predicting the output of a short program, running it, and investigating any surprise.
- **The crumb:** the prediction line and the result line.

How it goes. Use the four-line program from **The Computing Hour** (score starts at 3, then line 2 adds 2) or any short program from “For the parent.” Your student writes a prediction for every line of output, then types and runs the program, and writes what actually happened beside the prediction. If they differ, investigate: “Which line did something you didn’t expect?” Then change one number and predict again.

In-home — PY-03 Variable Boxes

- **Time:** about 10 minutes.
- **Materials:** sticky notes and a marker.
- **Safety and privacy:** none needed.
- **The fun:** being the computer’s memory.
- **The skill:** seeing that a new value replaces the old one, and that text and numbers are different.
- **The crumb:** a photo of the boxes after the trace.

How it goes. Take the snack and count program from “For the parent.” Each variable gets a sticky note with its name on top. Your student reads the program aloud, line by line. Each time a box gets a new value, they cross out the old one, or peel off the note and write a new one. At the end, read what’s in each box. Then ask: “What would happen if count were “2”, in quotes?” Text in quotes can’t be added to a number. Writing a word like “TEXT” or “NUMBER” in the corner of each note helps.

Talk box (this chapter)

Opening: “What do you predict this line will do — and why?”

Follow-ups:

1. Prediction: “And the line after that?”
2. Evidence: “What in the code makes you think so?”
3. Location: “Which line is the last one that did what you expected?”
4. Change: “What’s the one change you’d try first?”
5. Revision: “You can change your prediction. Run it again.”

How to wait. After you ask, count to five in your head. When the question is about code on the screen, let them read as long as they need. Reading code is slow, even for experienced programmers. Look at the screen, not at your student.

What a stuck silence usually means. The program is too long to hold in mind, so cover all but the first two lines with a sheet of paper. Your student doesn’t know what one piece means, such as range or ==, and is embarrassed to say so; ask, “Which word on this line is the least familiar?” They’re waiting for you to fix it, so put your hands in your lap. Or the red text feels like a verdict. Read the last line aloud yourself, calmly, as if it were a weather report.

Don’t say / say instead. Don’t say “You typed it wrong.” Say “Read me line 3, one character at a time.”

This week at a glance

Day	Focus	Try-it
Hour 1	Same ideas, new clothes	BLK-01 Scratch Side-by-Side

Day	Focus	Try-it
Hour 2	The editor, and a first error on purpose	PY-01 Hello, Editor; PY-02 Predict the Print
Hour 3	Boxes that change	PY-03 Variable Boxes, then type and run the program
Other days	Ten minutes	PLAN-07 Paper Parsons with Python lines; a typing warm-up
Errand	A full Hour away from home	OUT-02 Library Computer Hour

Practice that actually builds learning

Over the next four to six weeks, work through these roughly in order. Each new idea gets a week or so of practice before the next arrives.

In-home — BLK-03 Block-to-Line Translator

- **Time:** about 15 minutes.
- **Materials:** two or three Scratch scripts drawn or printed on paper, and a pencil.
- **Safety and privacy:** none needed.
- **The fun:** cracking a code you already half know.
- **The skill:** translating blocks into Python on paper.
- **The crumb:** the translated lines.

How it goes. Give your student a Scratch script on paper, such as *ask “What’s your name?” and wait, then say (join “Hi,” (answer))*. Their job is to write the Python without a computer: `name = input("What's your name? ")` and `print("Hi,", name)`. Then try one with *if ... then ... else*. Type the translation into Thonny afterward to check. **13–14:** translate a script with a loop inside a decision, and get the indentation right on paper first.

In-home — BLK-02 Remix and Explain

For 11–12.

- **Time:** about 25 minutes.
- **Materials:** a working Scratch project, opened from your family’s files or the Scratch site, with you present.
- **Safety and privacy:** no sharing, comments, or account sign-ups during this activity. If you open a project someone else made, it stays a private remix.
- **The fun:** changing one rule in a game and seeing what breaks.
- **The skill:** Use, Modify, Create: changing one part of a working program and explaining the change.
- **The crumb:** the saved remix, with a credit line naming the original maker.

How it goes. Your student plays the project, then changes exactly one rule: the speed, the score for a hit, the number of lives. Before running it, they predict what will be different. Afterward, they explain the change in one sentence and write a credit line: “Original by . ***I changed*** .” That credit line is the start of a habit the whole book cares about.

In-home — PY-04 Ask and Answer

- **Time:** about 20 minutes.
- **Materials:** Thonny.
- **Safety and privacy:** use a first name, or a made-up one, in any program that asks a name.
- **The fun:** a program that talks back.
- **The skill:** `input()`, text versus numbers, and `int()`.
- **The crumb:** the program file.

How it goes. Start with the name-and-age program from “For the parent.” Predict, run, and type Sam and 12 (or any first name). Then plant the bug on purpose: change `int(age) + 1` to `age + 1` and predict what will happen. Run it, and read the `TypeError` together. The last line of the error names the problem. Fix it. Then modify: ask for a favorite number and print double it.

In-home — PY-05 Choose Your Path

- **Time:** about 30 minutes.
- **Materials:** paper for a path map; Thonny.
- **Safety and privacy:** none needed.
- **The fun:** a tiny text adventure where the player’s choices matter.
- **The skill:** `if`, `elif`, and `else`.
- **The crumb:** the program file plus the path map.

How it goes. Start with the left-or-right program from “For the parent.” Before typing, your student draws a path map on paper: a box for each choice and an arrow for each answer. Then they predict what happens for “left,” “right,” “Left,” and “banana,” and run all four. The “Left” surprise is the lesson: to Python, a capital letter makes a different word. Then modify: add a third path, or a second question inside one path. **13–14:** make the program accept “Left” too, and explain how.

In-home — PY-06 Loop It

- **Time:** about 25 minutes.
- **Materials:** Thonny; the Debug Log.
- **Safety and privacy:** none needed.
- **The fun:** making the computer do a hundred things with two lines.
- **The skill:** for loops with `range()`, and counting patterns.
- **The crumb:** the program file.

How it goes. Start with the three-times-table loop from “For the parent.” Predict all five lines of output before running. Then change `range(1, 6)` to `range(1, 11)` and predict the last number.

Many first predictions include the stopping number. That's the lesson: `range` stops just before it. Then modify: print the five times table, or count down from 10.

In-home — PY-07 Turtle Patterns

- **Time:** about 30 minutes.
- **Materials:** Thonny.
- **Safety and privacy:** none needed.
- **The fun:** a program that draws.
- **The skill:** loops that draw shapes, using Python's turtle module.
- **The crumb:** a screenshot of the pattern.

How it goes. Python's turtle module is a little on-screen pen that moves and turns when told. It comes from the drawing tools of Logo, an educational language from 1967.³³

```
import turtle

pen = turtle.Turtle()
for side in range(4):
    pen.forward(100)
    pen.right(90)
turtle.done()
```

This opens a window and draws a square, 100 steps on each side. Before running it, your student predicts the shape. Then: what would make a triangle? (Three sides, turning 120 each time.) A hexagon? A star? Let them predict the angle before each run. **13–14:** draw a square inside a loop that turns a little after each square, and predict the pattern before running it.

In-home — PY-08 Guess-the-Number Game

- **Time:** about 40 minutes, across two weeks.
- **Materials:** Thonny; the Debug Log.
- **Safety and privacy:** none needed.
- **The fun:** a real game, written by your student, that a sibling can play.
- **The skill:** a `while` loop, random numbers, and decisions, in a Use-Modify-Create project.
- **The crumb:** the version 1 and version 2 files.

How it goes. This is Chapter 1's Guess My Number, as a program. Week one is Use: your student types this, predicts what will happen, and plays it.

```
import random

secret = random.randint(1, 20)
guess = 0
while guess != secret:
    guess = int(input("Guess a number from 1 to 20: "))
    if guess < secret:
        print("Too low.")
```

```

elif guess > secret:
    print("Too high.")
print("You got it!")

```

`random.randint(1, 20)` picks a whole number from 1 to 20, including both ends.³⁴ `!=` means “is not equal to.” The loop keeps asking until the guess matches. Ask: “Using the halving trick, what’s the most guesses you should ever need?” (Five.)

Week two is Modify and Create. Save the first version, then save a copy as version 2 and change it: a range of 1 to 100, a different message for each try, or a limit on guesses. **11–12:** one change, predicted and tested. **13–14:** add a guess counter that reports “You got it in 4 guesses,” and a question at the end that offers to play again. Chapter 3 comes back to this game to debug it.

PLAN-05 and PLAN-07, now in Python

Chapter 1’s trace tables and Paper Parsons work just as well with Python. For a trace table (PLAN-05), use the snack and count program, with a column for each variable. For a Paper Parsons (PLAN-07), cut the guess-the-number game into strips, one line each, mix them, and have your student put them back in order, keeping each strip’s indentation. Both make good ten-minute practice on days without a full Hour.

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

- **Time:** about 45 minutes.
- **Materials:** a public library computer, your own USB stick, a notebook, and the day’s program on paper.
- **Safety and privacy:** you sit with your student the whole time. Save only to your own USB stick, not to the library computer. Sign in to nothing personal, and log out and close the browser before you leave. See Family safety basics in **The Computing Hour**.
- **The fun:** a change of scene, and proof that your skills travel.
- **The skill:** running a full Computing Hour on a computer that isn’t yours, and saving to your own drive.
- **The crumb:** the file, saved in two places once you’re home.

How it goes. Before you go, check what the library computers allow: some have Python installed, some allow a browser editor, and some allow neither. Run the five steps as usual: plan on paper, predict in the notebook, build and test, debug and log, and save to your USB stick. At home, copy the file to its second place. For a family with no computer at home, this is how the screen half of this book happens.

Blocked, then mixed

Blocked practice for the new move. For a week, every Hour practices the newest idea: a week of variables, then a week of decisions, then a week of loops.

Mixed practice. Once loops arrive, mix. A good Hour might hold a Predict the Print with a decision in it, a short trace table, and fifteen minutes on the game. Mixing teaches your student to

recognize which tool a problem needs.

One incorrect example to diagnose

Here is an illustration, not a real family. A thirteen-year-old's folder holds `game.py`, `game2.py`, `final.py`, `final_final.py`, and `final_final2.py`. The version that worked last Tuesday is in one of them. Nobody knows which.

Diagnosis. No date in the names, no way to tell what changed, and nothing saved in a second place.

Rescue. Date first, then a short description, then a version number: `2026-10-19-guess-game-v2.py`. One line in the Debug Log says what changed. Every save goes to two places.

For the student

You've probably made things with blocks. Now you're going to type.

Python is a real programming language, used by people who build websites, run science experiments, and make games. It reads a lot like English. But typing code is fussier than snapping blocks together, and the computer is completely literal about it. Here's what you need to know.

Capitals matter. `print` works. `Print` doesn't. To Python, they're two different words, and it only knows one of them. If you type `Print("hi")`, you'll get an error that says name 'Print' is not defined. Python is telling you it's never heard of `Print`. Recent versions even suggest the fix.

Brackets come in pairs. Every `(` needs a `)`. Every opening quotation mark needs a closing one. When one is missing, Python stops and tells you which line the problem is on.

Indentation means something. In Scratch, the C-shaped blocks showed which steps were inside a loop. In Python, the indentation does that job. Lines that belong inside a loop or an `if` start four spaces in. Forget them, and you'll see an `IndentationError` whose message starts `expected an indented block`.

A tiny example. Read this, and predict exactly what it prints before you read on:

```
lives = 3
for turn in range(2):
    lives = lives - 1
print("Lives left:", lives)
```

It prints `Lives left: 1`. The loop runs twice, and each time it takes one away. If you predicted 0, check `range(2)`: it runs two times, not three. If you predicted the message would print twice, look at the indentation: `print` isn't indented, so it's outside the loop and runs once, at the end.

Red text is information, not judgment. Everyone who programs sees error messages every day. Here's how to read one. Look at the last line first; it says what kind of problem it is. Then find the line number. Go to that line, and also check the line just above it, because sometimes the real mistake is a little earlier. Change one thing. Run it again.

Reading the docs. Python has official documentation that explains every built-in function. When you meet a new one, look it up and read just the first sentence or two. For `input()`, the docs say it reads a line and returns it as a string, which means text. That one sentence explains why you need `int()` to do math with what someone types. You don't have to understand the whole page. Looking things up is what programmers do.

This part is hard for everyone. The first weeks of typing code are slow and full of small errors. Missing colons, missing brackets, a capital where it shouldn't be. Every programmer has been there, and the errors get rarer surprisingly fast.

Help I used. From now on, at the end of every Hour, you'll say (or write, if you're 13 or older) what help you used: a book, the docs, a parent, a sibling, a tutorial. "None" is a fine answer when it's true. Saying it every time is how you keep your work honestly yours.

Try this today. Write a program that asks someone's favorite animal and then prints a sentence using it. Predict what it'll print before you run it. Then break it on purpose: delete a bracket, run it, and read the error. Fix it.

Explain it back. Show someone your program and explain each line in your own words. If there's a line you can't explain, that's the line to look at next.

If you're 13 or older: look up one function you haven't used yet in the Python documentation, such as `len()` or `round()`. Read the first sentence, predict what it does in a small program, and test it.

Challenge. Change the guess-the-number game so it tells you how many guesses you took. Predict where in the program the counting line has to go before you try it.

Save like someone will need to find it. Name every file with the date first, then a few words, then a version number: `2026-10-19-guess-game-v2.py`. When you're about to make a big change, save a copy as the next version first. Then if the change goes badly, the version that worked is still there, and you can compare the two to find exactly what broke. A file called `final_final2.py` helps nobody, including you next month.

What to save. Every program, with a date-first name, in two places. Your Debug Log entries. Your version 1 and version 2 of the game.

If it isn't clicking

Typing is slow and painful. That's normal for the first month. Do five-minute typing warm-ups on other days, and keep programs short, under ten lines. Typing speeds up faster than you'd think.

Indentation errors keep coming. Count the spaces with your finger: four per level. If your editor can show spaces as dots, turn that on. Read the line above the error; it often ends in a colon that wants indented lines below it.

"I liked Scratch better." Keep a Scratch project going alongside Python, for fun. Blocks aren't the enemy. Do one more Scratch Side-by-Side so the connection stays visible.

Racing ahead and skipping Predict. No run without a written prediction line. Hands on the table until it's there. The prediction is the part that builds understanding; the run just checks it.

Error messages cause panic. Read the last line aloud yourself, calmly. Then ask only, "What line number does it name?" That's the whole first job.

Your student is far ahead. Let them read the docs for a new function and teach it to you. Your role is the same: ask for the prediction, be the first user, and ask for the explanation.

Tools, including AI

Thonny is the home default: free, with Python built in, no account, and offline once installed.²

Raspberry Pi Code Editor runs Python in a web browser and can be used without creating an account. Saving projects online needs an account, and for creators under 13, a parent or guardian sets it up with their own email address. These rules have been changing, so check the current terms before you start.³⁵ It's a good choice for a library computer or a borrowed laptop.

CodeAI (formerly Code.org) Python Lab runs Python in the browser and is designed for students aged 13 and up.³⁶ Use it at 13–14 only, with you present.

Not Trinket. It shut down on 31 August 2026, and older guides still point to it.³⁷

No Replit or GitHub accounts at 11–12. Both require users to be at least 13, and Replit asks for a parent's permission under 18.^{38,39} Neither is needed for this chapter.

AI: none in this chapter, at either age. At 11–12, there are no coding assistants or AI tutors anyway. At 13–14, keep the first text weeks AI-free too, so your student builds the reading muscle that everything else depends on: reading a line, predicting it, and reading the error. AI tools come back, supervised, in later chapters. The AS checks below are always AI-free. For the rules at each age, see the AI box in **The Computing Hour**.

What “done enough” looks like

Your student is ready to move on when they can:

- write and run a program that asks a question, stores the answer, decides with `if/elif/else`, and repeats with a loop;
- predict the output of a short program before running it;
- read an error message's last line and find the line it names;
- save files with date-first names a stranger could read, in two places;
- say “Help I used: ...” at the end of every Hour without being asked.

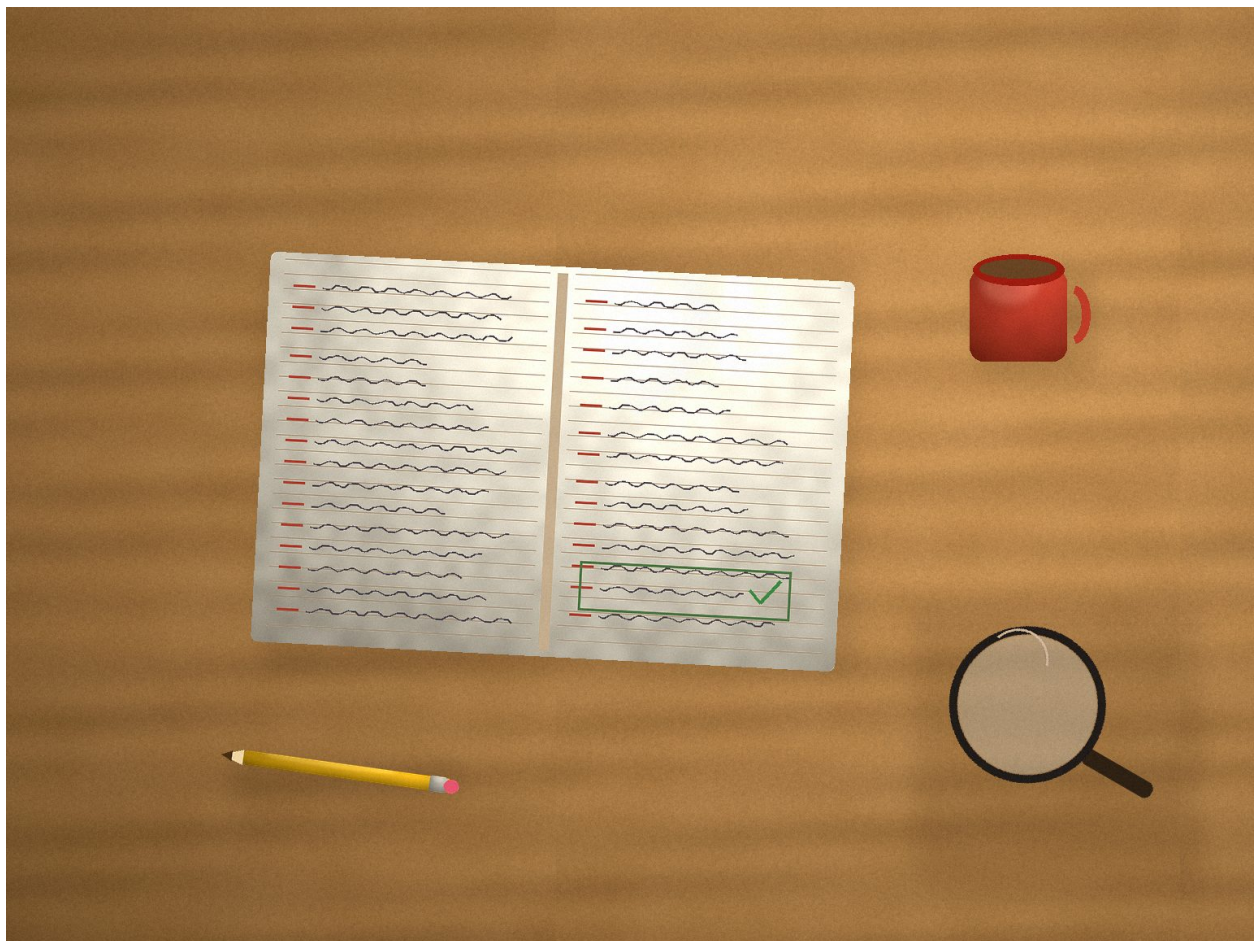
11–12: programs of five to ten lines; reading code aloud together is fine. **13–14:** a `while` loop and random numbers; reads the documentation for one new function alone; the guess-the-number game with a counter and replay.

The checks (always AI-free). Keep the monthly Cold Trace (AS-01), now with a short Python program. Add a Debug on the Spot (AS-02): plant one bug, such as a missing `int()` or a capital `Print`, and have your student fix it while explaining aloud. Then an Explain Your Code (AS-03): your student explains their own game, line by line, to you. If they can explain every line, it's theirs. Write a one-line note on each, dated, for the portfolio.

If most of the list is in place, move on to Chapter 3, where debugging becomes a discipline of its own.

Chapter 3

Debugging as a Discipline: Errors, Tests, and the Debug Log



Why this matters

Here is an illustration, not a real family. Picture a twelve-year-old who has spent forty minutes on a guessing game. She presses Run. The screen fills with red text. She pushes her chair back, says “It’s broken,” and reaches for the key that deletes everything. This is one of the most important moments in learning to program, and it looks like failure. It isn’t. It’s the start of the real work.

A bug is a mistake in the instructions. Debugging is finding and fixing it. Every program bigger than a few lines has bugs on the way to working, and professionals plan for that. Two experienced programmers put it plainly in a well-known book for working developers: no one writes perfect software, so debugging will take up a big part of the day.⁴⁰ That isn't a confession. It's a job description.

This chapter is the middle of the book's focus line: **Predict before you run; debug before you rewrite; explain it before you call it yours.** Chapter 2 taught your student to predict what a line will do. This chapter teaches what to do when the prediction and the screen disagree. The national computer science standards for ages 11–14 describe the same skill in plain words: use systematic ways to test a program, improve it, and write down the changes.¹⁷ “Systematic” is the key word. Debugging is a method, not a mood.

The word has a story your student will enjoy. People called small faults in machines “bugs” long before computers; Thomas Edison talked about bugs in electrical circuits in the 1870s. In 1947, engineers working on the Mark II computer at Harvard found a moth stuck in one of its parts, taped it into their logbook, and labeled it “first actual case of bug being found.” The Smithsonian, which now holds that logbook, says it was probably not Grace Hopper's own, but she and the rest of the team helped make “bug” and “debug” ordinary words.⁴¹ The engineers did something worth copying: they wrote it down. By the end of this chapter, your student will keep a Debug Log of their own: a dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes.

This chapter also carries the feelings that come with mistakes. Students this age care a great deal about how they look to others, and red text can feel like a grade on them rather than a note about the code.¹⁹ You can change that. The way you react to the first error message of the week teaches more than anything on this page.

For the parent: understand it yourself

You don't need to be a programmer to teach debugging. You need a calm voice, a few good questions, and the habit of looking at the code instead of the clock.

The everyday picture. A cake comes out flat. You don't throw away the kitchen. You ask what you expected (a tall cake), what happened (a flat one), and what the best clue is (the baking powder tin is still sealed). Then you change one thing next time. Debugging is that, written down.

The precise picture. Python can go wrong in three ways, and it helps to name them:

- A **syntax error** means the computer can't read it. A colon or a parenthesis is missing, so Python stops before running anything.
- A **runtime error** means it crashes while running. The program starts, reaches a line it can't carry out, and stops with a message.
- A **logic error** means it runs and does the wrong thing. No red text at all. The answer is simply wrong.

Here is a two-line program with a syntax error:

```
score = 0
```

```
if score > 10
    print("Big score")
```

Python stops and reports `SyntaxError: expected ':', with a small arrow under the end of line 2`. Add the colon and it runs.

Here is a runtime error:

```
score = 5
print("Score: " + score)
```

The program starts and then stops on line 2 with `TypeError: can only concatenate str (not "int") to str`. In plain words: you tried to glue words to a number. Writing `str(score)` turns the number into text, and the line works.

And here is a logic error:

```
average = 3 + 6 + 9 / 3
print(average)
```

It prints `12.0`. The student expected `6.0`. Python did the division first, the way math rules say it should, so only the 9 was divided. Parentheses fix it: `(3 + 6 + 9) / 3` prints `6.0`. Nothing turned red. Only a prediction caught it, which is why the book keeps asking for one.

How to read the red text. The Python documentation gives the two habits that matter most. The last line of an error message says what happened. The message also names the line where Python noticed the problem, but that is not always the place that needs fixing; a missing parenthesis on line 4 can be reported on line 5.⁴² So the order is: read the last line first, then the line number, then look at that line and the one above it. Error wording can differ slightly between Python versions. Thonny, the free beginner’s editor this book uses, comes with its own recent Python, so your screen may not match these examples word for word.²

The method. Professional debugging comes down to a short loop, and your student can learn every step:

1. Write the Expected/Actual line: “I expected... It did...”
2. Read the clue: the error message, or the wrong output.
3. Make one guess about the cause.
4. Change one thing.
5. Run it again and look.
6. Write it in the Debug Log.

The “one thing” matters. The same developers’ book notes that if you changed only one thing and the program stopped working, that one thing is probably responsible.⁴⁰ Change five things and you learn nothing, even if it starts working.

Tests before trouble. A test table lists inputs and the outputs you expect before you run anything. The best rows are the awkward ones: zero, a negative number, an empty answer, a very large number. Programmers call these edge cases, and the developers’ book advises testing boundary conditions hard.⁴⁰ A program that works for 5 and breaks for 0 isn’t finished.

Tools for seeing inside. Two help most. A *print detective* adds temporary `print()` lines that show what a variable holds at each step, then deletes them when the bug is found. Thonny also has a step-through debugger. Press `Ctrl+F5` instead of `F5` and the program runs one step at a time; `F6` takes a big step and `F7` a small one. Choosing View, then Variables, shows every labeled box and its contents as the program runs.²

The rubber duck. Many programmers explain their code, line by line, out loud, to a rubber duck. This is a habit from working practice, not a research finding. The developers' book tells of a colleague who kept a small rubber duck on his computer while he coded, and says that the simple act of explaining what the code is supposed to do “often causes the problem to leap off the screen and announce itself.” If no person is around, they add, “a rubber duck, or teddy bear, or potted plant will do.”⁴⁰ It works because saying each step forces you to notice the step you skipped.

Saving versions. A student who saves `2026-10-19-guess-game-v1.py`, then `2026-10-20-guess-game-v2.py`, then `v3`, date first, with one line in a changelog for each and every file saved in two places, can always go back. Nobody has to fear trying a change.

The feelings. Expect some perfectionism, some “I’m bad at this,” and at least one slammed laptop lid. Keep your feedback about the code: “Line 7 is missing something,” not “You always forget.” Praise the find, not only the fix: “You spotted that the arrow was one line late. That’s real debugging.”

Wrong answers you should be able to hear

- **“The computer is broken.”** It almost never is. Reply: “Maybe. What did you expect, and what did it do?” The computer did exactly what it was told. That’s the problem, and also the good news.
- **“I’ll just start over.”** A fresh file throws away the clue along with the bug. Reply: “Deleting the file is not a strategy. Let’s read the last line of the message first.”
- **“It works, so it’s right.”** It worked for one input. Reply: “What happens with zero? With a negative number? Let’s write three tests.”
- **“I fixed it.”** But the student can’t say what changed. Reply: “Great. Which line, and why did that fix it? Put it in the log.”
- **“Python is stupid.”** Frustration talking. Reply: “It’s very literal. Let’s find the line where it stopped doing what you meant.”

Your five-minute warm-up

Tomorrow, before the student arrives, type the three short programs above into Thonny and run each one. Read the last line of each message aloud. Then practice this chapter’s opening question once, out loud, and practice saying nothing afterward while you count to five in your head. That’s the whole preparation.

How to teach it this week

Run the week on the usual five steps of the Computing Hour. Here is how each step leans toward debugging.

- **Unplug and Plan (5–10 minutes).** On paper, write the Expected/Actual line format and the Debug Log headings: date, prediction, bug, clue, fix, what I learned.
- **Predict (2–5 minutes).** Before running, the student says what will appear on screen, including “it will crash on line 3” if they suspect it.
- **Build and Test.** The student runs the program, compares it with the prediction, and fills in a test table.
- **Debug and Log.** One bug, one guess, one change, one log line.
- **Save and Explain.** The student saves a new version and says: “I predicted... It did... Next I’d change...” and “Help I used: ...” (even if it’s “none”).

In-home — BUG-01 Expected/Actual Line

- **Time:** 5 minutes, every time something goes wrong.
- **Materials:** the Debug Log (a notebook or a single document) and a pencil.
- **Safety and privacy:** none needed.
- **The fun:** it turns “ugh” into a sentence, which calms everyone down.
- **The skill:** writing what you expected and what happened before touching the code.
- **The crumb:** one Debug Log line.

Try this exact sentence the first time the screen surprises your student: “Before you change anything, finish this: ‘I expected... It did...’” Write it on a card and prop it against the screen for the first two weeks. The rule is simple: no keyboard until the line is written.

11–12: You can write it while they say it. **13–14:** They write it themselves, and you only check that it exists.

In-home — BUG-02 Read the Red Text

- **Time:** 15 minutes.
- **Materials:** Thonny, the three error programs from the parent section, and an index card.
- **Safety and privacy:** offline; nothing to sign up for.
- **The fun:** decoding the message like a spy reading a note.
- **The skill:** finding the error type and the line number, and knowing which kind of error it is.
- **The crumb:** an error decoder card.

Run each broken program. For each one, the student copies the last line onto the card and writes a plain translation beside it. Add a fourth: the `IndentationError` that appears when the line under an `if` isn’t pushed in. (Python 3.13 reports expected an indented block after ‘if’ statement on line 2; other versions word it slightly differently.) Add a fifth: misspell `score` as `scroe` and get a `NameError`. Recent versions even suggest Did you mean: ‘score’? A finished card might read:

- `SyntaxError` = it can't read this line. Look for a missing colon, parenthesis, or quote.
- `IndentationError` = the spacing is wrong. Lines inside an `if` or loop move in four spaces.
- `NameError` = I used a name Python doesn't know. Check spelling and capitals.
- `TypeError` = I mixed things that don't mix, like words and numbers.

Thonny highlights unclosed quotes and parentheses, which it calls the most common beginner syntax errors.² Point that out; it's a free clue.

In-home — BUG-08 The Debug Log

- **Time:** 5 minutes at the end of each Hour.
- **Materials:** a dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes, kept beside the computer or in the student's project folder.
- **Safety and privacy:** first names only if it's ever shown to anyone.
- **The fun:** watching the list grow is satisfying, and old entries are funny a month later.
- **The skill:** writing down the bug, the clue, the fix, and what was learned, with the date.
- **The crumb:** the monthly log page.

A good entry is short:

6 Oct. Bug: game said "Too high" when I was too low. Clue: both messages said "Too high." Fix: line 11 now says "Too low." Learned: copy-paste can copy the mistake too.

Tell your student, more than once, that a full Debug Log is evidence of good work. A blank one usually means the bugs got fixed in silence, or not at all.

How to fade your help

Follow the rescue order: **ask** → **wait** → **hint** → **short demo on your own copy** → **hand it back**. Ask the opening question. Wait. If silence goes on, give a hint that points to a place, not an answer: "Look at line 7 and the line above it." If that fails, fix a *similar* bug in your own copy while they watch, then hand the keyboard back. The student's file stays the student's.

11–12: Expect to reach "hint" often in the first week. Co-write test tables together. Plant two bugs, not three, for the planted-bug game.

13–14: The student leads the Hour and you are **first user**: you try the program like a stranger, report what happened, and fix nothing. Plant three bugs, including one logic bug. Then swap: the student plants bugs for you.

Ready 12: A twelve-year-old who reads error messages without being asked is ready for the logic-bug round.

A worked Hour (illustration)

Here is an illustration, not a real family. Picture a thirteen-year-old and his father on a Wednesday. In Unplug and Plan, the son copies the Debug Log headings into his notebook. In Predict, his father

points to a guessing game the son wrote last week and asks, “What will happen when you type 5?” The son says, “Too low or too high, then another guess.” He runs it. It crashes on line 10 with `TypeError: '<' not supported between instances of 'str' and 'int'.`

His father asks the opening question and counts to five in his head. The son reads the last line twice. “It says I’m comparing a string and a number. The guess is a string.” He remembers from Chapter 2 that `input()` always gives text. He changes one thing: `guess = int(input("Guess 1 to 10: "))`. He runs it. It works. He writes the log entry, saves `2026-10-21-guess-game-v2.py`, and says, “I predicted it would say too low or too high. It crashed. Next I’d change the message so it tells you how many tries are left. Help I used: none.” Twenty minutes, one bug, one clear win.

Practice that actually builds learning

In-home — BUG-03 Planted Bugs

- **Time:** 20 minutes.
- **Materials:** a working program saved under a new name, Thonny, and a sheet for the bug list.
- **Safety and privacy:** offline.
- **The fun:** a treasure hunt where you made the treasure.
- **The skill:** finding three bugs a parent planted in working code.
- **The crumb:** a bug list with line numbers.

Start from a program that works. You plant the bugs in a copy. (At 13–14, the student plants some for you, too.) Here is a guessing game with three planted bugs:

```
import random

secret = random.randint(1, 10)
guess = 0
tries = 0

while guess != secret:
    guess = int(input("Guess 1 to 10: "))
    tries = tries + 1
    if guess < secret:
        print("Too high")
    elif guess > secret:
        print("Too high")

print("Got it in", trys, "tries")
```

Python first stops at line 7 with `SyntaxError: expected ':'`. Fix that and the game runs, but it says “Too high” for a low guess (line 11, a logic bug). When the player finally guesses right, it crashes on line 15 with `NameError: name 'trys' is not defined`. One bug of each kind. The student writes each one on the list: line, kind, fix.

11–12: Plant two bugs, one syntax and one runtime. Tell them how many there are. **13–14:** Plant

three, including a logic bug, and don't say which line. Then the student plants bugs in a working program for you, and you narrate your hunt. Watching a parent read an error message calmly is a lesson in itself.

In-home — BUG-04 Rubber Duck Explain

- **Time:** 10 minutes.
- **Materials:** a rubber duck, a stuffed animal, or a houseplant, plus the stuck program.
- **Safety and privacy:** none needed.
- **The fun:** it's silly, and it works anyway.
- **The skill:** explaining code line by line out loud.
- **The crumb:** one sentence: the line where I found it.

Say once, plainly, that this is a working programmer's habit, not a classroom game. The student reads each line to the duck and says what it does: "Line 9 adds one to tries. Line 10 checks if the guess is too small..." The rule is every line, no skipping the "obvious" ones, because the bug usually lives in an obvious one. Your job is to leave the room or stay silent.

In-home — BUG-05 One Change at a Time

- **Time:** 15 minutes.
- **Materials:** a buggy program and a change log (three columns: change, result, keep or undo).
- **Safety and privacy:** offline.
- **The fun:** a controlled experiment, like a science lab.
- **The skill:** change one thing, run it again, write it down.
- **The crumb:** the change log.

Give the student a program with one stubborn logic bug. The rule is one edit per run, logged. If a change doesn't help, undo it before trying the next. Students who change five things at once sometimes get lucky, but they can't explain why it works, and the next bug beats them.

In-home — BUG-06 Test Table First

- **Time:** 20 minutes.
- **Materials:** paper, a pencil, and the program about to be written.
- **Safety and privacy:** none needed.
- **The fun:** trying to break your own program before anyone else can.
- **The skill:** writing inputs and expected outputs before coding, including edge cases (zero, negative, empty, very large).
- **The crumb:** a test table with pass/fail marks.

Before writing a small program that gives a reaction to a score, the student fills in a table:

Input	Expected	Actual	Pass?
12	Big score		

Input	Expected	Actual	Pass?
10	Keep going		
0	Keep going		
-3	That can't be right		
(nothing typed)	Asks again		

The row for 10 is the interesting one: is 10 a big score or not? Writing the table forces the decision before the code hides it. **11–12:** Write the table together. **13–14:** The student writes it alone, with at least two edge cases you didn't suggest.

In-home — BUG-07 Print Detective

- **Time:** 15 minutes.
- **Materials:** Thonny and a program that gives a wrong number.
- **Safety and privacy:** offline.
- **The fun:** spying on your own variables.
- **The skill:** adding temporary `print()` lines to watch values change.
- **The crumb:** a before/after snippet.

This program is meant to add $1 + 2 + 3 + 4 + 5$:

```
total = 0
for number in range(1, 5):
    print("number is", number, "total is", total)
    total = total + number
print(total)
```

The student expected 15. It prints 10. The detective line (line 3) shows number going 1, 2, 3, 4 and then stopping. The clue is in plain sight: `range(1, 5)` stops before 5. Changing it to `range(1, 6)` prints 15. Then the detective line comes out. Save the before and after.

Show the same bug with Thonny's step-through (Ctrl+F5, then F7 for small steps, with View, then Variables, open). Some students like the printed trail; others like watching the boxes change. Both are real tools.

In-home — PROJ-02 Version Saves and Changelog

- **Time:** 10 minutes.
- **Materials:** the project folder and a plain text file called `changelog.txt`.
- **Safety and privacy:** save in two places (see Family safety basics in **The Computing Hour**).
- **The fun:** a time machine for code.
- **The skill:** saving v1, v2, v3 with a one-line changelog.
- **The crumb:** the changelog file.

Before any big change, the student saves a copy with today's date first and the next version number, like `2026-10-20-guess-game-v2.py`, and writes one line in the changelog: `2026-10-20 v2:`

guess is now a number; fixed crash on line 10. If v3 goes badly, v2 is waiting. Both live in two places.

In-home — PY-08 Guess-the-Number Game

- **Time:** 40 minutes over two Hours.
- **Materials:** Thonny and the guessing game from Chapter 2.
- **Safety and privacy:** offline.
- **The fun:** a game the family will actually play.
- **The skill:** a while loop, random numbers, and if checks, now debugged and improved.
- **The crumb:** version 1 and version 2 files.

Use the game as this week’s lab. The student makes one improvement per version (a tries counter, a limit of five guesses, a “play again?” question) and logs every bug met along the way. The family plays as **first user**, one at a time. Siblings are excellent at typing “banana” when asked for a number, which is a test case worth having.

In-home — AS-02 Debug on the Spot

- **Time:** 15 minutes, about once a month.
- **Materials:** a short program with one planted bug the student hasn’t seen.
- **Safety and privacy:** offline; no AI, no notes beyond the error decoder card.
- **The fun:** a calm show of skill.
- **The skill:** fixing a planted bug while explaining aloud.
- **The crumb:** your one-line note.

The student reads the program aloud, predicts, runs it, and fixes the bug while saying what they’re doing. You write one line: “Found the missing colon in under two minutes; read the last line first without being asked.” This check is always AI-free.

Out-of-home — OUT-01 Algorithms in the Wild

- **Time:** 20 minutes on a walk or an errand.
- **Materials:** a pocket notebook.
- **Safety and privacy:** road rules; stay together in stores and parking lots; no photos of strangers or staff.
- **The fun:** catching machines making mistakes.
- **The skill:** writing pseudocode (a plan for a program written in plain words) for a machine you meet, then finding its bug.
- **The crumb:** a pseudocode page with the bug circled.

This week the walk becomes a “bug in the world” hunt. Look for systems that follow rules and sometimes get them wrong: a self-checkout that says “unexpected item” when nothing is there, a crosswalk button that seems to do nothing, an automatic door that opens for a passing leaf. The student writes the likely rule (“IF something moves near the sensor THEN open”) and then the

Expected/Actual line for the mistake. At home, they suggest one change to the rule. Keep it about machines, not people, and keep guesses labeled as guesses.

Talk box (this chapter)

Opening: “What did you expect, what happened instead, and what’s your best clue?”

Follow-ups: - Prediction: “If you make that change, what do you think it will do?” - Evidence: “What did you see that makes you say that?” - Location: “Which line is the last one that did what you expected?” - Change: “What’s the one thing you’d change first?” - Revision: “You can change your prediction. Run it again and see.”

How to wait. After asking, 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. That simple shift tells them you’re both working on the program, not watching them.⁵

What a stuck silence usually means. The question was too vague (“What’s wrong?”), so make it smaller: “Read me the last line of the message.” There may have been no wait time. They may fear looking foolish, which is strong at this age, so keep every comment about the code. They may be waiting for you to fix it, because you have before. There may have been no prediction, so nothing to compare. Or the screen is louder than the question; lower the lid for a moment and talk on paper.

Don’t say / say instead. Don’t say “That’s wrong.” Say “Show me the line where it stops doing what you expected.”

Blocked, then mixed

For the first week, practice one error type at a time: a day of syntax errors, a day of runtime errors, a day of logic bugs. Blocked practice builds the decoder card. In the second week, mix them. Hand over three programs without saying which kind of error each has. Naming the kind is half the skill, and only mixed practice teaches it.

One incorrect example to diagnose

Here is an illustration, not a real family. Show your student this folder, from an imaginary student:

`game.py`, `game2.py`, `game_final.py`, `game_final_final2.py`, `game_NEW.py`

Ask: “Which one works? Which one has the fix for the crash? What would you do differently?” Let them find the problem (no way to tell what changed, or which is newest) and propose the fix: numbered versions and a changelog line for each. Students who laugh at this folder rarely build one.

Siblings and co-ops

Planted bugs work beautifully across ages. An older sibling plants, a younger one hunts, and the planter must explain each bug afterward. In a co-op, pairs swap programs and fill in each other’s test tables. The rule for everyone: talk about the code, not the coder.

For the student

Here's something true: every programmer you have ever heard of has seen thousands of red error messages. They didn't get good by avoiding errors. They got good at reading them.

A bug is a mistake in the instructions. Debugging is finding and fixing it. Bugs come in three kinds:

- A **syntax error** means Python can't read your code. It stops before running anything. Usually a missing colon, parenthesis, or quote.
- A **runtime error** means the program starts, then crashes on a line it can't do.
- A **logic error** means the program runs fine and gives the wrong answer. These are the sneaky ones, because nothing turns red.

How to read red text. Start at the bottom. The last line names the error and says what happened. Then find the line number. Look at that line, and the line above it, because Python sometimes notices a mistake one line late.

Here's a worked example. You run this:

```
score = 5
print("Score: " + score)
```

The last line of the message says `TypeError: can only concatenate str (not "int") to str`. Translate it: "You tried to stick words and a number together." Line 2 is where it happened. Your guess: the number needs to become text. Change one thing: `print("Score: " + str(score))`. Run it. It prints `Score: 5`. Write it in your Debug Log:

Date. Bug: crash on line 2. Clue: `TypeError`, words plus number. Fix: `str(score)`.
Learned: `+` only joins text to text.

That's the whole method: **Expected/Actual, clue, guess, one change, run, log.**

One change at a time. When something's wrong, it's tempting to change five things at once. If it starts working, you won't know which change did it. If it gets worse, you won't know which change broke it. One change, one run, every time.

Test your program before someone else does. Before you write a program, write a test table: some inputs and what you expect. Include the awkward ones. What if someone types 0? A negative number? Nothing at all? A huge number? Programs usually work for the easy cases. The edges are where bugs live.

Spy on your variables. If a number comes out wrong, add a temporary line like `print("total is", total)` inside your loop and watch it change. Or press `Ctrl+F5` in Thonny and step through one line at a time with `F7`. Delete the spy lines when you're done.

Talk to the duck. Explain your code out loud, line by line, to a rubber duck or a stuffed animal. Professional programmers really do this. Saying every step forces you to notice the one you skipped.

Keep the log. A Debug Log is a dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes. Add one line on what you learned. It takes two minutes. Months from now, you'll

flip back and see bugs that once took you an hour and now take you ten seconds. That's what getting better looks like from the inside.

Try this today

Ask your parent to plant two bugs in a program that already works. Find them. For each one, write the line number, the kind of error, and your fix.

Explain it back

Tell your parent, without looking at this page, the difference between a syntax error and a logic error. Then give an example of each from your own code.

This part is hard for everyone

Logic errors are hard. There's no message to read, only an answer that's wrong, and your brain keeps reading what you *meant* instead of what you *wrote*. That isn't a sign you're bad at this. It's why programmers use test tables, print lines, and ducks.

The ten-minute rule. If you've been stuck for ten minutes: stand up. Explain it to the duck. Write the Expected/Actual line. *Then* ask for help. And one more thing: deleting the file is not a strategy. The clue is in there.

Challenge

Write a short program that works. Save a copy. Plant three bugs in the copy: one syntax, one runtime, and one logic. Give it to your parent and watch them hunt. You'll learn a surprising amount about bugs by hiding them well.

If you're 13 or older: make your logic bug an off-by-one, like `range(1, 5)` where you meant 1 to 5. Then write a test table that would catch it.

What to save

- Your error decoder card.
- Your Debug Log, dated.
- Your test table with pass/fail marks.
- Each version of your program, named date first (like `2026-10-19-guess-game-v2.py`), and your changelog.

Every entry in that log is proof you solved something. Keep it in two places.

If it isn't clicking

They change five things at once. Slow down to BUG-05. Sit beside them with the change log and say "one" out loud before each run. After a week it becomes habit.

They won't read the error messages. Many students glance at red text and look away, as if it were shouting. Go back to the error decoder card. Cover everything except the last line with an index card, and ask only, "What does this line say, in your words?"

Frustration takes over. The ten-minute rule exists for this. Stand up, take a short walk around the house, and come back to paper. If a lid gets slammed, stay calm and keep it about the code: "That bug is being stubborn. Let's write down what we know before we look again." End the Hour early if you need to, on a small win.

They blame the computer. "It's broken" usually means "I don't know where to look." Hand them a location question: "Which line is the last one that did what you expected?" Then follow the rescue order: ask → wait → hint → short demo on your own copy → hand it back.

If two weeks pass and every bug still needs you, step back to paper. Trace a five-line program by hand, writing each variable's value as it changes. Debugging on screen gets easier once tracing on paper feels natural.

Tools, including AI

Thonny, a free Python editor that works offline, is the main tool here. It highlights unclosed quotes and parentheses, and its step-through debugger and Variables view (described above) show what the program is really doing.² Nothing else is required. A rubber duck helps.

AI and error messages. For the rules, see the AI box in **The Computing Hour**. Here is what's different for debugging. The student's own Expected/Actual line and best guess always come first.

11–12: Rarely, after your student has written Expected/Actual and a guess, you may ask a chatbot on your own account to explain an error message and read the explanation aloud. Then check it against the code together. The AI explains; the student fixes.

13–14: Under the service's terms, with you nearby and the screen visible, your student may ask an AI to *explain* an error message, never to fix the code. Add a line to the Debug Log: "Help I used: asked AI what IndentationError means after my guess."

A preview of something that comes up again in Chapter 8: for 13–14, any code an AI suggests goes through **AI-07 Explain Every Line**. The student explains every line before deciding whether to use any of it, and anything kept goes on a Whose Work card as made by a tool. Debugging is also a good honesty check. A student who can find and fix a bug in their own program wrote that program. AS checks like Debug on the Spot are always AI-free.

Services change their rules often, so check the current terms before anyone uses one.

What "done enough" looks like

Your student has met this chapter's goals when they can:

- name the three kinds of error and give an example of each;
- read an error message from the last line, then the line number, without being reminded;

- find a planted bug while explaining aloud (AS-02 Debug on the Spot);
- show a dated Debug Log with real entries, including what they learned;
- write a test table, with edge cases, before writing a small program;
- save numbered versions with a changelog.

Placement by skill. A twelve-year-old who reads errors calmly and finds logic bugs is ready for the 13–14 moves. A fourteen-year-old who still deletes files when stuck needs more weeks of BUG-01 and BUG-05, and that's fine.

The life of the habit. The Debug Log doesn't end with this chapter. Every program from here on gets Expected/Actual lines and log entries. Once a month, read back through the log together. Your student will see bugs they once found hard and now spot in seconds. That's the best kind of progress report, and they wrote it themselves.

Chapter 4

Functions, Lists, and Data: Bigger Programs and Spreadsheets That Answer Questions



Why this matters

By now your student can write a program of ten or twenty lines. Somewhere around thirty lines, something changes. The same five lines show up three times. A program needs to remember twelve

quiz questions, not one. A question comes up that a single number can't answer. This chapter gives your student the three tools that carry programs past that point: functions, lists, and data.

A **function** is a named chunk of code you can use again. A **list** is items in order, numbered from 0. **Data** is information collected to answer a question. Together they let a student build a quiz game the family will actually play, keep a packing list that updates itself, and turn two weeks of bird-feeder tallies into a chart that tells the truth.

The national computer science standards for ages 11–14 name these skills directly: use a procedure (a function) to organize code for clarity and reuse; use variables to store and change data; and use a computing tool to sort, filter, group, and summarize data. The same standards ask students to see how design choices in a chart change what people conclude, and how decisions about collecting data can lead to misleading results.¹⁷ That last part matters as much as the code. A student who can ask “Who’s missing from this data?” is learning something many adults never do.

There’s a satisfying idea underneath all of this. A spreadsheet formula like SUM is a function, with inputs and an output, just like the ones your student will write in Python. Once they see that, a spreadsheet stops being a mysterious office program and becomes another place to compute.

The focus line still holds: predict before you run; debug before you rewrite; explain it before you call it yours. Functions add one question to the Predict step: what goes in, and what comes out?

For the parent: understand it yourself

Functions

The everyday picture. A recipe card that says “Make the pancake batter” is a function. You write the steps once, give the card a name, and from then on you just say “make the batter.” If the card says “for ___ people,” that blank is an input.

The precise picture. In Python, `def` defines a function. The name in the parentheses is a **parameter**, an input the function needs:

```
def draw_box(size):
    for row in range(size):
        print("#" * size)
```

```
draw_box(3)
draw_box(5)
```

This prints a 3-by-3 square of # marks, then a 5-by-5 square. The four indented lines run only when the function is called. Recall from earlier chapters: a variable is a labeled box whose contents can change, and a loop means do it again. Here `size` is a variable that gets a new value each time the function is called.

Return versus print. This is the idea that trips up nearly everyone. `print` shows a value on the screen and then forgets it. `return` hands the value back so the program can keep using it.

```
def c_to_f_print(c):
```

```

print(c * 9 / 5 + 32)

def c_to_f(c):
    return c * 9 / 5 + 32

answer = c_to_f_print(100)
print("Saved:", answer)
answer = c_to_f(100)
print("Saved:", answer)

```

The output is 212.0, then Saved: None, then Saved: 212.0. The first function showed the answer and kept nothing; Python’s word for “nothing” is None. The second handed the answer back. A function that returns can be tested: `c_to_f(0)` gives 32.0, and `c_to_f(-40)` gives -40.0, which is a pleasing edge case.

Lists

The everyday picture. A row of numbered cubbies, where the first cubby is number 0.

The precise picture. A list holds items in order. The Python documentation describes positions as “zero-based”: the first item is at position 0.^{43,44} You can add to the end with `append`, take out an item with `remove`, count items with `len`, and walk through every item with a `for` loop:

```

packing_list = ["socks", "toothbrush", "book"]
packing_list.append("flashlight")
packing_list.remove("book")
print(len(packing_list))
for item in packing_list:
    print("Pack the", item)
print(packing_list[0])

```

This prints 3, then three “Pack the...” lines (socks, toothbrush, flashlight), then socks. If the student types `packing_list[3]`, Python stops with `IndexError: list index out of range`. Three items live at positions 0, 1, and 2. That off-by-one surprise is worth meeting on purpose.

Data and spreadsheets

Question first. Good data work starts with a question, not a pile of numbers: “Which bird visits our feeder most in the morning?” The question decides what to count, when, and for how long.

Spreadsheets. A spreadsheet is a grid. Each row is one observation (one morning, one bird). Each column is one kind of information (date, time, bird, how many). Each box is a cell, named by its column letter and row number, like B4. Clean data means one kind of thing per column and the same spelling every time: “sparrow,” not “Sparrow” in one row and “sparow” in the next.

Formulas are functions. `=SUM(D2:D31)` takes a range of cells as input and returns their total. `AVERAGE`, `MAX`, and `COUNTIF` work the same way. `=COUNTIF(C2:C31, "sparrow")` counts how many cells in column C say “sparrow.” Spreadsheet programs differ in small ways (some want a

semicolon between the parts instead of a comma), so if a formula won't take, check the program's own help page for its exact form. Any spreadsheet program your family already has will do.

Sort, filter, group. Sorting puts rows in order. Filtering hides the rows you don't need right now. Grouping gathers rows that share something (all the cardinals) so you can total each group.

Charts that tell the truth. The same numbers can tell different stories. If 41 sparrows and 37 cardinals are drawn as bars on an axis that starts at 35, the sparrow bar is three times as tall as the cardinal bar (6 units against 2). Start the axis at zero and the bars look nearly equal, which is the honest picture. The standards ask students to see exactly this: design choices change how people read the same data.¹⁷

Who's missing? If you only count birds at 8 a.m., you'll miss the ones that come at dusk. If you only ask your friends, you'll miss everyone who isn't your friend. Every dataset leaves someone or something out. Saying so is part of an honest answer. Chapter 8 returns to this idea when it explains how AI learns from data.

Data about data. A phone photo carries more than the picture. Phones and cameras automatically record details about when, where, and how a photo was taken, and that hidden information can travel with the photo when it's shared. Phones let you turn off the camera's access to location.⁴⁵ This hidden information is called metadata. It makes a good data lesson here, and Chapter 7 covers what it means for privacy. Before anyone shares a photo, see Family safety basics in **The Computing Hour**.

Wrong answers you should be able to hear

- **"The list starts at 1."** Natural, and wrong in Python. Reply: "Let's test it. Print `packing_list[0]` and see what comes out."
- **"Print and return are the same thing."** They look the same on screen. Reply: "Store the answer in a variable and print the variable. What do you get?"
- **"My chart proves it."** A chart shows; it rarely proves. Reply: "What does the axis start at? Who's missing from the data?"
- **"The average tells you everything."** Reply: "What's the biggest number? The smallest? Would one strange day change the average a lot?"
- **"It works, so I'm done."** Reply: "What does your test table say about zero, a negative number, and an empty list?"

Your five-minute warm-up

Type the `c_to_f` example into Thonny and run it. Say out loud why the first "Saved" line shows None. Then open any spreadsheet, type five numbers in a column, and try `=SUM` and `=AVERAGE` on them. Practice this chapter's opening question once, and then count to five in your head.

How to teach it this week

This chapter is two or three weeks of work, not one. A good order: functions and lists first, then data, then a small project that uses all three.

Use the Hour's steps as usual:

- **Unplug and Plan.** Function machines on paper; index cards for lists; a question card for data.
- **Predict.** For every function call, the student says what goes in and what will come out. For every list line, which position and which item.
- **Build and Test.** Write the function or list program, run the test table.
- **Debug and Log.** Expected/Actual for every surprise; one Debug Log line.
- **Save and Explain.** “I predicted... It did... Next I'd change...” and “Help I used: ...”

In-home — FN-01 Function Machine

- **Time:** 10 minutes.
- **Materials:** a cardboard box or a drawn box, small slips of paper, rule cards.
- **Safety and privacy:** none needed.
- **The fun:** guessing the secret rule.
- **The skill:** input, rule, output.
- **The crumb:** rule cards.

You hold a secret rule card (“double it and add one”). The student feeds in a number on a slip; you pass back the output. After three or four tries, they guess the rule and test it on a number they haven't tried. Then swap roles. Finish by writing the rule as Python: `def machine(n): return n * 2 + 1.`

In-home — LIST-01 Index Cards in a Row

- **Time:** 10 minutes.
- **Materials:** index cards, a marker, a table.
- **Safety and privacy:** none needed.
- **The fun:** acting out a program with your hands.
- **The skill:** positions starting at 0; append and remove.
- **The crumb:** a photo of the row (no faces).

Write a small number above each card position, starting at 0. Lay out three items. Call out commands: “append flashlight” (add to the end), “remove book” (slide the rest left), “what's at position 2?” The student moves cards and answers. The slide-left after a removal is the moment the numbering clicks.

How to fade your help

Follow the rescue order: **ask** → **wait** → **hint** → **short demo on your own copy** → **hand it back**. For functions, the best hint is a question about inputs: “What does this function need to know to

do its job?”

11–12: Start with functions that print and take one parameter. Use one list at a time. In spreadsheets, stick to SUM and AVERAGE at first.

13–14: Add return and test tables for every function. Use two lists side by side, or pairs. Add COUNTIF and grouping. Write a short investigation that names its limits. The student leads; you’re **first user**, trying each program like a stranger and reporting without fixing.

Ready 12: A twelve-year-old who explains why the first “Saved” line shows None is ready for FN-03.

A worked Hour (illustration)

Here is an illustration, not a real family. Picture a twelve-year-old and her mother on a Tuesday. In Unplug and Plan, the girl lays out four index cards and numbers them 0 to 3. Her mother asks, “What goes in, what comes out — and how would you test it?” about a list program the girl has sketched. In Predict, the girl says `print(packing_list[3])` will show “flashlight.” She runs it: `IndexError`. Her mother waits. The girl looks at her cards. “Oh. I removed the book. There are only three now. The last one’s at 2.” She changes one thing, runs it, and writes the log line. “I predicted flashlight. It crashed. Next I’d print the length first. Help I used: my cards.”

Practice that actually builds learning

In-home — FN-02 Define Your Own

- **Time:** 25 minutes.
- **Materials:** Thonny.
- **Safety and privacy:** offline.
- **The fun:** inventing a command Python didn’t have.
- **The skill:** `def` with parameters.
- **The crumb:** the program file.

Start with `draw_box(size)` from the parent section, then let the student design their own: `draw_triangle(height)`, or `greet(name, times)`. Students who did turtle drawings in Chapter 2 (PY-07 Turtle Patterns) can write `draw_square(size)` with the turtle and call it at four sizes to make a pattern. Before writing any function, they write the paper version first: name, inputs, what it does.

In-home — FN-03 Return to Sender

For 13–14; ready 12.

- **Time:** 25 minutes.
- **Materials:** Thonny, paper.
- **Safety and privacy:** offline.
- **The fun:** a converter you can trust.

- **The skill:** return versus print; a unit converter with a test table.
- **The crumb:** the program file plus the test table.

The student writes `c_to_f(c)` and fills in a test table before running it (this is BUG-06 Test Table First, back from Chapter 3): $0 \rightarrow 32$, $100 \rightarrow 212$, $-40 \rightarrow -40$, $37 \rightarrow 98.6$. Then they write the opposite converter and check that converting there and back gives the starting number.

In-home — FN-04 Refactor the Repeats

For 13–14.

- **Time:** 25 minutes.
- **Materials:** a program the student wrote earlier with repeated code.
- **Safety and privacy:** offline.
- **The fun:** making a program shorter and better at the same time.
- **The skill:** finding repeated code and replacing it with a function.
- **The crumb:** before and after files.

Look through an old program for lines that appear twice or more with only a small difference. That difference becomes the parameter. The program must give exactly the same output before and after, which is a test in itself. Save both versions.

In-home — LIST-02 Packing List Program

- **Time:** 25 minutes.
- **Materials:** Thonny.
- **Safety and privacy:** offline.
- **The fun:** a tool for the next family trip.
- **The skill:** append, remove, len, and a loop over a list.
- **The crumb:** the program file.

Start from the packing list example. Add an `input()` loop so the user can type items to add, and “done” to stop. Print the final list with numbers. A good test: what happens if you remove something that isn’t in the list? (Python raises a `ValueError`.^{43,44}) Can the student make the program check first with `if item in packing_list:`?

In-home — LIST-03 Quiz Machine

- **Time:** 40 minutes over two Hours.
- **Materials:** Thonny, a friend or family member to take the quiz.
- **Safety and privacy:** offline; quiz-takers’ first names only.
- **The fun:** a game other people play.
- **The skill:** two lists side by side, a score variable, and feedback.
- **The crumb:** the program file plus a friend’s score.

```
questions = ["7 x 8?", "Capital of France?", "Bits in a byte?"]
answers = ["56", "paris", "8"]
```

```

score = 0
for i in range(len(questions)):
    reply = input(questions[i] + " ")
    if reply.strip().lower() == answers[i]:
        score = score + 1
        print("Right!")
    else:
        print("The answer was", answers[i])
print("Score:", score, "out of", len(questions))

```

If someone answers 56, Paris, and 7, it prints “Right!” twice, then “The answer was 8,” then Score: 2 out of 3. Ask your student to explain `.strip().lower()`: it tidies spaces and capitals so “Paris” still counts. **11–12:** Five questions on a topic they love. **13–14:** Ten questions, and a message that changes with the score.

In-home — LIST-04 Find the Biggest

- **Time:** 20 minutes.
- **Materials:** a deck of cards, then Thonny.
- **Safety and privacy:** none needed.
- **The fun:** beating a famous algorithm (exact steps to reach a goal) at its own game.
- **The skill:** the maximum algorithm by hand, then in code.
- **The crumb:** a trace table and the file.

Deal cards face down in a row. The student turns them over one at a time, keeping only the biggest so far in hand. Then the same in code:

```

temps = [14, 19, 11, 23, 17]
biggest = temps[0]
for t in temps:
    if t > biggest:
        biggest = t
print(biggest)

```

It prints 23. Then compare with Python’s built-in `max(temps)`, which gives the same answer. This is also a good place to revisit ALG-04 Battleships Search from Chapter 1: finding the biggest in an unsorted list means checking every item, just as a search through unsorted numbers does.⁴⁶

In-home — DATA-01 Question First

- **Time:** 15 minutes.
- **Materials:** an index card.
- **Safety and privacy:** pick questions that don’t need other people’s private information.
- **The fun:** choosing a question you actually care about.
- **The skill:** writing a question and deciding what data would answer it.
- **The crumb:** the question card.

On the card: the question, what to count, where, when, for how long, and one thing that might

make the answer wrong. “Which bird visits our feeder most? Count each visit, 8:00–8:15 a.m., for 14 days. Might be wrong because: we’re only watching mornings.”

In-home — DATA-02 Spreadsheet Basics

- **Time:** 25 minutes.
- **Materials:** any spreadsheet program the family already has.
- **Safety and privacy:** keep the file on the family drive.
- **The fun:** seeing a messy notebook become a tidy table.
- **The skill:** rows, columns, cells; entering clean data.
- **The crumb:** the sheet file.

Headings in row 1: Date, Time, Bird, Count. One observation per row. Agree on spellings before typing. Then the student finds three “dirty” entries you slipped in (a typo, a capital, a number typed as words) and cleans them.

In-home — DATA-03 Formulas Are Functions

- **Time:** 20 minutes.
- **Materials:** the sheet.
- **Safety and privacy:** none needed.
- **The fun:** making the spreadsheet do the arithmetic.
- **The skill:** SUM, AVERAGE, MAX, and COUNTIF as functions with inputs and outputs.
- **The crumb:** the sheet with formulas shown.

For each formula the student says, before pressing Enter, what goes in (the range) and what will come out. **11–12:** SUM and AVERAGE. **13–14:** add MAX and COUNTIF. Most spreadsheets have a setting to show formulas instead of results; turn it on for the saved copy.

In-home — DATA-04 Sort, Filter, Group

- **Time:** 20 minutes.
- **Materials:** the sheet.
- **Safety and privacy:** none needed.
- **The fun:** asking the data new questions.
- **The skill:** sorting, filtering, grouping, and summarizing.
- **The crumb:** a summary table.

Sort by bird, then by count. Filter to show only one bird. Then build a small summary table: each bird and its total, using COUNTIF or the spreadsheet’s grouping tool. **13–14:** Compare two groups (weekdays and weekends) and write one sentence about the difference and one about why it might not mean much.

In-home — DATA-05 A Chart That Tells the Truth

- **Time:** 25 minutes.

- **Materials:** the summary table.
- **Safety and privacy:** none needed.
- **The fun:** making a chart lie, then making it honest.
- **The skill:** choosing a chart; spotting a misleading axis.
- **The crumb:** the chart plus a one-line caption.

Make a bar chart of bird totals. Then change the axis so it starts just below the smallest bar and watch the differences balloon. Change it back to zero. The student writes one honest caption: “Sparrows came most often (41 visits), but cardinals were close (37).”

In-home — DATA-06 Who Is Missing?

- **Time:** 15 minutes.
- **Materials:** the question card and the data.
- **Safety and privacy:** none needed.
- **The fun:** catching your own blind spot.
- **The skill:** spotting a biased sample: what or who the data leaves out.
- **The crumb:** a two-sentence note.

Ask: “What didn’t we count? When weren’t we watching? What would change the answer?” The note: “We only watched mornings, so birds that come in the evening are missing. Our answer is about morning visitors, not all visitors.”

Out-of-home — OUT-03 Data Walk

- **Time:** 20 minutes.
- **Materials:** a clipboard, a tally sheet, a pencil.
- **Safety and privacy:** road rules; stay together; count things, not people’s faces; no photos of strangers.
- **The fun:** fieldwork.
- **The skill:** counting something outdoors and bringing the data home.
- **The crumb:** the tally sheet.

Pick something countable from a safe spot: cars by color at a corner, dogs on a park loop, kinds of trees on one block. Agree on categories before you start, including “other.” At home, the tally goes into the spreadsheet for DATA-02 through DATA-06. The student who owns a micro:bit can log light or temperature instead; see the optional PHYS-02 Sensor Logger in Chapter 6.

In-home — PROJ-01 Project One-Pager

- **Time:** 20 minutes.
- **Materials:** one sheet of paper.
- **Safety and privacy:** none needed.
- **The fun:** planning something real.
- **The skill:** problem, user, features, test plan, and schedule on one page.
- **The crumb:** the one-pager.

Before a bigger project (the quiz machine with ten questions, or a bird report with a chart), the student writes one page: what problem it solves, who will use it, three features, how it will be tested, and what gets done in which Hour. PLAN-07 Paper Parsons, from Chapter 1, makes a good warm-up: cut a finished function into strips, mix them up, and put them back in order.

Talk box (this chapter)

Opening: “What goes in, what comes out — and how would you test it?”

Follow-ups: - Prediction: “If you call it with 0, what will it give back?” - Evidence: “What in the output makes you say it’s working?” - Location: “Which position in the list is the last one that’s right?” - Change: “What’s the one thing you’d change to make it work for an empty list?” - Revision: “Want to 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 or the spreadsheet, not at the student.⁵

What a stuck silence usually means. The question may be too big; shrink it to one input. They may not know what “returns” means yet; go back to the function machine. They may be afraid the answer is obvious and they’re missing it. There may be no test table to compare against. Or the spreadsheet is full of colors and charts, and the question got lost; hide the chart for a minute.

Don’t say / say instead. Don’t say “No, lists start at zero.” Say “Let’s ask Python. Print position 0 and see.”

Blocked, then mixed

Spend a few days on functions alone, then lists alone, then spreadsheets alone. Then mix: one Hour might ask for a function that takes a list and returns its biggest number. That mixed task is where students discover the ideas belong together.

One incorrect example to diagnose

Here is an illustration, not a real family. Show your student this function, from an imaginary student who says it finds the biggest temperature:

```
temps = [-5, -2, -9]
biggest = 0
for t in temps:
    if t > biggest:
        biggest = t
print(biggest)
```

It prints 0. Zero isn’t even in the list. Ask: “What did this student assume?” (That temperatures are never below zero.) “What test would have caught it?” (A list of all negative numbers.) “What’s the fix?” (Start with `biggest = temps[0]`.) A test table with edge cases would have found it in a minute.

A week without a screen

Everything in this chapter has a paper version, which is handy for travel weeks or a broken laptop. Function machines need only slips of paper. Index cards act out every list command. A tally sheet and a pencil can do the work of COUNTIF: count the marks, group them by bird, add the groups. Charts can be drawn on graph paper, which is the best way to feel how much an axis choice matters, because you have to decide where zero goes with your own pencil. A student can even “run” a function by hand, writing a trace table with one column for each variable and one row for each step. When the laptop comes back, the code feels familiar instead of new.

Encouragement that means something

Praise the move, not the student. Try these:

- “You wrote the test table before the code. That’s what careful programmers do.”
- “You noticed the axis didn’t start at zero. Plenty of adults miss that.”
- “You turned three copies of the same lines into one function. The program’s easier to read now.”
- “You wrote down who was missing from your data. That makes your answer more honest, not less.”

Specific praise tells your student exactly what to do again. “Great job” tells them only that you’re pleased.

Siblings and co-ops

The quiz machine is made for groups: each student writes five questions on a shared topic, then everyone takes everyone’s quiz. Data walks work well with two counters on two different corners, then a comparison at home. Charts make a good co-op show-and-tell, especially the “lying” version beside the honest one.

For the student

If you’ve copied and pasted the same five lines three times, you’ve found a function.

A **function** is a named chunk of code you can use again. You write the steps once, give them a name, and call the name whenever you need them. Inputs go in the parentheses after the name. Those inputs are called parameters.

```
def draw_box(size):
    for row in range(size):
        print("#" * size)
```

```
draw_box(3)
```

This prints three rows of ###. Call `draw_box(5)` and you get a bigger square, with no new code.

Print or return? `print` shows something on the screen. `return` hands an answer back so your program can keep using it. If you write `answer = some_function(100)` and then `print(answer)` shows `None`, your function printed instead of returning. It showed you the answer and then threw it away.

A **list** is items in order, numbered from 0. That “from 0” part catches everyone at first:

```
pets = ["cat", "dog", "fish"]
print(pets[0])
print(len(pets))
```

This prints `cat`, then `3`. Three items, at positions 0, 1, and 2. Ask for `pets[3]` and Python says `IndexError: list index out of range`. Now you’ll know why. You can add to the end with `pets.append("hamster")`, take out an item with `pets.remove("dog")`, and visit every item with `for pet in pets:`.

Data is information you collect to answer a question. Always start with the question. “Which bird comes to our feeder most?” tells you what to count, when, and how. A spreadsheet holds the data in rows and columns. And here’s the neat part: spreadsheet formulas are functions. `=SUM(D2:D31)` takes a range of cells in and gives a total out. Same idea as Python.

A worked example

Say your question is “Which bird visits most in the morning?” You count for two weeks and type it into a sheet. `=COUNTIF(C2:C40, "sparrow")` tells you how many sparrow visits there were. You do the same for each bird and make a bar chart. Sparrows: 41. Cardinals: 37. Your caption says: “Sparrows came most often, but cardinals were close. We only watched in the mornings, so evening birds are missing.”

That last sentence is what makes it honest.

Functions and lists together

The real power shows up when you combine them. A function can take a whole list as its input:

```
def total(numbers):
    result = 0
    for n in numbers:
        result = result + n
    return result

print(total([3, 2, 1]))
```

This prints `6`. Now you can total any list you like with one short call. That’s how bigger programs stay readable: small functions with clear names, each doing one job, each tested on its own.

A tip for when things go wrong: when a list program gives a strange answer, add `print(my_list)` and `print(len(my_list))` just before the line that misbehaves. Seeing what’s really in the list solves most list bugs in under a minute. Then take those lines out again.

Try this today

Write a function called `double(n)` that returns $n * 2$. Before you run anything, fill in a test table: what should `double(0)`, `double(-3)`, and `double(1000)` give? Then run it and check.

Explain it back

Tell your parent the difference between `print` and `return`, using a function you wrote. Show them what happens when you store the result in a variable.

This part is hard for everyone

Return versus print confuses almost every new programmer, because both put something on the screen when you test them. The difference only shows up when you try to *use* the answer. If it feels slippery, that's normal. Store the answer in a variable and print the variable. That test always tells you which one you wrote.

Lists starting at 0 also feel wrong for a while. Keep index cards with the numbers written above them nearby until it doesn't.

Challenge

Build a quiz machine with two lists: one for questions, one for answers. Keep score. Give it to two people, and write down their scores (first names only). Then improve one thing based on what confused them.

If you're 13 or older: write a function `biggest(numbers)` that returns the largest number in a list without using `max()`. Test it with an empty list. What happens? Decide what it *should* do, and make it do that.

What to save

- Your function files, with a test table for each.
- Your quiz machine and a friend's score.
- Your spreadsheet, with formulas shown.
- Your chart, with a one-line honest caption.
- Your question card and your "who's missing?" note.
- Your Debug Log entries.

Save everything in two places.

If it isn't clicking

Functions feel pointless. That usually means the programs are still small enough not to need them. Go back to FN-01 Function Machine, then find a real repeat in an old program. The point of a function shows up the moment you'd otherwise copy and paste.

Return makes no sense. Step back to paper. The function machine passes a slip back through the slot; that’s return. A machine that shouts the answer and keeps the slip is print. Then run the “Saved:” example again.

Off-by-one errors keep happening. Bring back the index cards with positions written above them. Before each list line, the student points to the card. Add `print(len(my_list))` lines as print detectives.

The spreadsheet feels like busywork. The question isn’t theirs. Let them pick one they care about, even if it’s “Which of my games do I actually play most?” (Track it on paper for a week; no app needed.)

If none of this works, slow down. Use the rescue order: ask → wait → hint → short demo on your own copy → hand it back. It’s fine to spend an extra week on lists.

Tools, including AI

Thonny, the free offline Python editor from earlier chapters, is all the programming needs.² Its Variables view (View, then Variables) is especially good for lists, because you can watch items appear and disappear.

Spreadsheets. Use whatever spreadsheet program the family already has, on a computer in a shared room. Keep files on the family drive.

Optional data source. A student with a micro:bit can collect light or temperature readings and bring them into a spreadsheet; see the optional strand in Chapter 6.

AI. For the rules, see the AI box in **The Computing Hour**. What’s different here:

11–12: AI is parent-operated only. If you use it at all, it might suggest practice problems (answers hidden until your student tries).

13–14: After your student has chosen a chart type and said why, they may ask an AI, with you present, what kind of chart it would suggest and compare the reasons. If an AI suggests a formula or a function, it goes through AI-07 Explain Every Line before anything is used, with a “Help I used” line and a Whose Work card marking it as made by a tool. Check the current terms of any service first.

What “done enough” looks like

Your student is ready to move on when they can:

- write a function with a parameter and explain what goes in and what comes out;
- at 13–14, explain return versus print and test a function with a table that includes edge cases;
- use a list with `append`, `remove`, `len`, and a loop, and explain why the first item is at 0;
- start a data project with a written question;

- use SUM and AVERAGE (and at 13–14, COUNTIF and grouping) and say what each formula takes in and gives back;
- make an honest chart and name who or what is missing from the data.

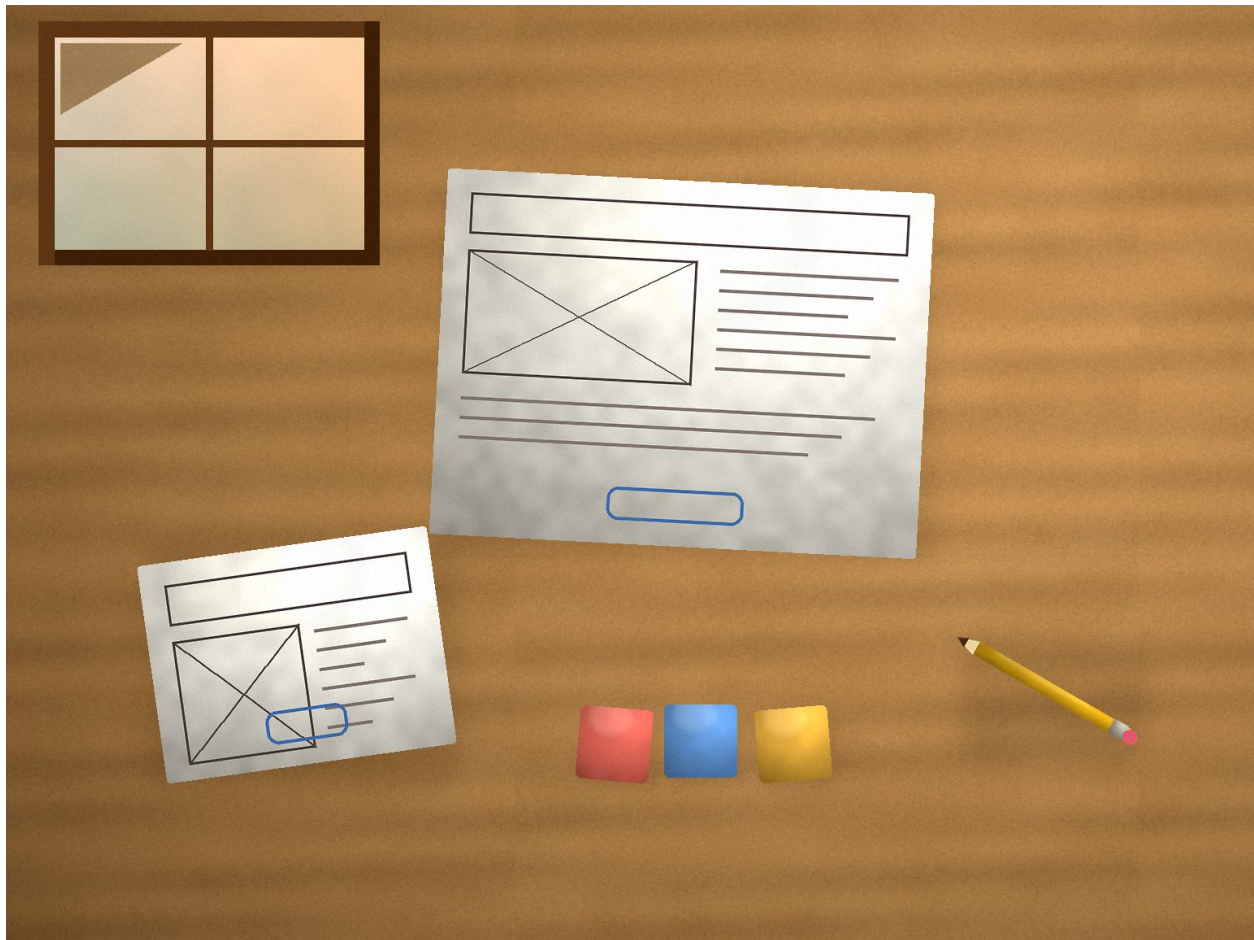
AS checks. Once this month, run AS-01 Cold Trace on a short list program (predict the output without running it), AS-02 Debug on the Spot with a planted off-by-one, and AS-03 Explain Your Code on the quiz machine. All three are AI-free.

Placement by skill. A twelve-year-old who handles return comfortably can take on FN-03. A fourteen-year-old still unsure about positions needs more time with index cards, and that's a fine use of a week.

The life of the habit. From now on, every program bigger than twenty lines should have at least one function, and every question about the world should start with a question card. Keep the bird sheet. It makes a good portfolio piece next to the chart that tells the truth.

Chapter 5

Building for the Web: HTML, CSS, and a First Taste of JavaScript



Why this matters

Every site your student has ever visited started as a blank file. Somebody typed a few lines of text, saved it, and opened it in a browser. That is still how it works, and your student can do it this week

with tools your computer already has.

Building a web page teaches something Python alone doesn't: that a program has an audience. A page is made to be read by someone else, on someone else's screen, possibly with their eyes or possibly with a screen reader that speaks the page aloud. That turns every choice (what goes first, what a link says, what a picture's description says) into a question about another person. It's a gentle way to teach that good code is considerate code.

A web page has three layers, and this chapter takes them in order:

- **HTML** gives the page its structure: headings, paragraphs, lists, links, pictures.
- **CSS** gives it style: colors, fonts, spacing.
- **JavaScript** gives it behavior: a button that does something when clicked.

The national computer science standards for ages 11–14 ask students to use reference documentation while they program and to improve their work based on user feedback, making it easier to use and more accessible.¹⁷ Web pages are a natural place for both. Your student will look things up in MDN Web Docs, a free reference and course for web developers run by Mozilla, and test pages on real people at a small family “demo day.”⁴⁷

One more thing makes the web a good fit at this age: no account is needed. A page saved on your own computer opens in your own browser, and nobody else sees it. Publishing a page online is a separate decision, made by you, later or never. This chapter keeps the work local.

For the parent: understand it yourself

The everyday picture. Think of a house. HTML is the frame and the rooms: this is the kitchen, this is a hallway. CSS is the paint, the furniture, and how far apart things sit. JavaScript is the light switches: press one and something happens.

The precise picture. An HTML file is plain text with **tags** in angle brackets. Most tags come in pairs: `<h1>` opens a top-level heading and `</h1>` closes it. Here is a small but complete page, saved as `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Birds at Our Feeder</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Birds at Our Feeder</h1>
  <p>I counted birds every morning for two weeks.</p>
  <h2>What I saw most</h2>
  <ul>
    <li>Sparrows</li>
    <li>Cardinals</li>
  </ul>
```

```
</body>
</html>
```

Double-click the file and it opens in the browser: a big heading, a sentence, a smaller heading, and a bulleted list. The `<head>` holds information about the page (its title on the browser tab, and a link to the style file). The `<body>` holds what you see.

CSS lives in a second file, `style.css`, and says how things should look:

```
body {
  font-family: sans-serif;
  background-color: rgb(250 248 240);
  margin: 2em;
}
h1 {
  color: rgb(20 80 140);
}
```

Each block starts with a **selector** (which things to style: `body`, `h1`) and lists **properties** and values inside curly braces. A color written as `rgb(20 80 140)` gives amounts of red, green, and blue, each from 0 to 255.⁴⁸ Chapter 6 explains why it’s those three numbers, and why 255.

The box model. Every element on a page is a box. From the inside out: the **content**, then **padding** (space inside the edge), then a **border**, then a **margin** (space outside, pushing other boxes away). Most layout puzzles in CSS are box puzzles.

JavaScript is a programming language that runs in the browser. Students who know Python will recognize variables, `if`, and functions with different punctuation. This chapter uses it for one small job: when a button is clicked, change some text.

Accessibility as good manners. Some people use a screen reader, which reads a page aloud. Many screen readers let a person jump from heading to heading, so headings work like a table of contents.⁴⁹ That’s why a page should use real headings in order (`h1`, then `h2`) instead of just making text big. MDN’s accessibility guide adds three habits a student can learn this week:⁴⁹

- Every picture gets **alt text**, a short description in the `alt` attribute that gives a direct representation of what the picture shows. A purely decorative picture gets an empty one: `alt=""`.
- **Link text makes sense on its own.** “Read about how web pages work” is good; “click here” tells a screen-reader user nothing.
- **Text stands out from its background.** The guide asks for a contrast of at least 4.5 to 1 between text and background color. Pale gray on white usually fails.

Reading the documentation. Programmers look things up all day; it’s part of the job, not a sign of weakness. MDN is the reference this book uses for HTML, CSS, and JavaScript.⁴⁷ Teach your student to look up a tag on MDN before asking anyone else.

Developer tools. Every major browser has built-in tools for looking at a page’s code. You can usually open them by right-clicking part of a page and choosing “Inspect,” or with `Ctrl+Shift+I` (or `F12`) on Windows and `Cmd+Option+I` on a Mac. Safari keeps them turned off until you enable them in its settings. The **Console** tab lists errors the browser meets while running JavaScript, which

is where a silent button usually explains itself.⁵⁰

Your computer or the web? When your student double-clicks `index.html`, the browser reads a file from your own drive. Nothing travels anywhere. When you visit a site online, your browser asks a faraway computer, called a server, for files much like these, and they arrive in small pieces over the internet. MDN draws a useful line here: the internet is the network that connects computers, and the web is one service that runs on top of it.^{51,52} Chapter 6 follows those pieces on their journey. For now, it's enough to know that the page your student builds is made of exactly the same stuff as the pages they visit.

Publishing and privacy. A page on your own computer is private. If your family ever decides to put a page online, that's your call, and it comes with the usual rules: first names only; no faces, school, or town; no photos of other people. Pictures on the page should be made by your student, or credited and used with permission. See Family safety basics in **The Computing Hour**.

Wrong answers you should be able to hear

- **“I changed it, but nothing happened.”** Usually the file wasn't saved, or the browser wasn't refreshed, or the wrong file is open. Reply: “Save, then refresh. Which file is the browser showing?”
- **“The page is broken.”** Browsers try hard to show *something*, so a missing closing tag can make half the page bold instead of showing an error. Reply: “Which part looks wrong? Let's find the tag just before it.”
- **“Alt text is optional.”** Reply: “Close your eyes. I'll read you the page. What do you want me to say when we get to the picture?”
- **“It was online, so I can use it.”** Reply: “Who made it? Did they say we could? Let's draw our own or credit it.”
- **“The AI made the page, but I get it.”** A generated page is made by a tool, not your student's work. Reply: “Great, then build one like it from a blank file and explain every line.”

Your five-minute warm-up

Type the small HTML page above into a plain text editor, save it as `index.html`, and open it in your browser. Change one word, save, refresh, and watch it change. Then practice this chapter's opening question once, aloud, and count to five in your head.

How to teach it this week

Plan on two to three weeks: structure first, then style, then (for students who are ready) one button. Each Hour follows the usual steps:

- **Unplug and Plan.** A paper wireframe, labeled with tag names.
- **Predict.** “When you refresh, what will change, and where?”
- **Build and Test.** Type, save, refresh, look. Then test on a **first user**.
- **Debug and Log.** Expected/Actual for every surprise, written in the Debug Log.

- **Save and Explain.** “I predicted... It did... Next I’d change...” and “Help I used: ...”

The tools: a plain text editor (the one that came with the computer is fine, but not a word processor, which adds hidden formatting) and a web browser. Make one folder for the project. Keep `index.html`, `style.css`, and any pictures together in it.

In-home — WEB-01 Paper Wireframe

- **Time:** 15 minutes.
- **Materials:** paper, pencil, a ruler if you like.
- **Safety and privacy:** none needed.
- **The fun:** designing before building, like an architect.
- **The skill:** sketching a page layout and labeling the parts as tags.
- **The crumb:** the sketch.

The student draws boxes for each part of the page (title, introduction, list, picture, link) and labels each with its tag: `h1`, `p`, `ul`, `img`, `a`. Then you ask the opening question: “Who is this page for — and what will they see first?” The answer often reorders the sketch.

In-home — WEB-02 My First Page

- **Time:** 30 minutes.
- **Materials:** a plain text editor, a browser, the wireframe.
- **Safety and privacy:** saved locally; no account; first names only on the page.
- **The fun:** your words, in a real browser.
- **The skill:** HTML headings, paragraphs, lists, and links.
- **The crumb:** `index.html`.

The student types the page from the wireframe, saving and refreshing after every few lines. That rhythm (small change, save, refresh, look) is the web version of Predict and Build and Test. Add one link: `<a href="https://developer.mozilla.org/">Read how web pages are built on MDN</a>`. Notice the link text says where it goes.

How to fade your help

Use the rescue order: **ask** → **wait** → **hint** → **short demo on your own copy** → **hand it back**. For HTML, the best hints point to a pair: “Every opening tag in this part, can you find its closing tag?”

11–12: One page plus a CSS file. You might type the first `<!DOCTYPE html>` skeleton together, then hand over the keyboard for everything inside `<body>`.

13–14: Three pages linked to each other, sharing one stylesheet, plus a JavaScript button and a round of feedback from first users. The student leads; you’re the **first user**, trying the page like a stranger and reporting what you see without fixing anything.

Ready 12: A twelve-year-old who finishes a styled page and asks “Can I make it *do* something?” is ready for WEB-06.

A worked Hour (illustration)

Here is an illustration, not a real family. Picture a fourteen-year-old and her grandmother on a Thursday afternoon. The girl has three pages about the birds at the feeder. In Predict, she says the heading will turn dark blue when she adds an `h1` rule to `style.css`. She saves, refreshes. Nothing changes. Her grandmother asks the opening question, then, after a pause, “What did you expect, and what happened?” The girl writes Expected/Actual. She opens the HTML and reads the `<link>` line out loud: `href="styles.css"`. She looks at the folder. The file is called `style.css`. One letter. She fixes it, refreshes, and the heading turns blue. Her log entry: “Bug: CSS didn’t load. Clue: no styles at all, not just one. Fix: file name in link line. Learned: if *nothing* is styled, check the link line first.” That last sentence is the kind of thing that turns into skill.

Practice that actually builds learning

In-home — WEB-03 Style It

- **Time:** 30 minutes.
- **Materials:** the page from WEB-02, a new file called `style.css`.
- **Safety and privacy:** local only.
- **The fun:** making it look like yours.
- **The skill:** CSS selectors, color, fonts, and spacing.
- **The crumb:** `style.css`.

Add the `<link rel="stylesheet" href="style.css">` line in the `<head>`. Write one rule at a time and predict each change before refreshing. The student should look up at least one property on MDN by themselves (`font-family`, `text-align`, `border`). **11–12:** Three rules. **13–14:** A shared stylesheet used by all three pages, so one change updates the whole site.

In-home — WEB-04 Paper Box Model

- **Time:** 10 minutes.
- **Materials:** four sheets of colored paper, scissors.
- **Safety and privacy:** none needed.
- **The fun:** nesting rectangles like a picture frame.
- **The skill:** margin, border, padding, and content.
- **The crumb:** a labeled paper model.

Cut a small rectangle (content: write a word on it). Glue it on a slightly bigger one (padding). Then a thin frame (border). Then the biggest sheet (margin). Label each layer. Now write the CSS: `p { padding: 1em; border: 2px solid gray; margin: 1em; }`, refresh, and match the screen to the paper.

In-home — WEB-05 Alt Text and Access Check

- **Time:** 15 minutes.

- **Materials:** the student's page and a printed checklist.
- **Safety and privacy:** none needed.
- **The fun:** seeing your page the way someone else does.
- **The skill:** alt text, heading order, contrast, and link text.
- **The crumb:** the checklist.

The checklist has four lines: every picture has alt text that says what it shows (or `alt=""` if it's decoration); headings go in order with no skipped levels; text is easy to read against its background; every link says where it goes. Then you read the page aloud with your eyes closed, the way a screen reader might, reading alt text when you reach a picture. Students hear the difference immediately. A picture might get: ``.

In-home — WEB-06 A Button That Answers

For 13–14; ready 12.

- **Time:** 30 minutes.
- **Materials:** the page, a new file called `script.js`.
- **Safety and privacy:** local only.
- **The fun:** a page that responds.
- **The skill:** a JavaScript click event that changes text on the page.
- **The crumb:** `script.js`.

Add to the HTML, just before `</body>`:

```
<button id="fact-button">Which bird came most?</button>
<p id="fact"></p>
<script src="script.js"></script>
```

Then, in `script.js`:

```
const button = document.querySelector("#fact-button");
const answer = document.querySelector("#fact");

button.addEventListener("click", function () {
  answer.textContent = "Sparrows: 41 visits in two weeks.";
});
```

Click the button and the empty paragraph fills in. In plain words: find the button, find the paragraph, and when the button is clicked, run a small function that sets the paragraph's text.⁵³ The script line goes at the end of the body so the button already exists when the script looks for it. Using a real `<button>` (rather than a styled box) means keyboards and screen readers know it's a button.⁴⁹

If nothing happens, open the Console with a parent (right-click, Inspect, then the Console tab) and read the error. It's the red text from Chapter 3, in a new place.

In-home — WEB-07 View-Source Detective

- **Time:** 15 minutes.
- **Materials:** a browser and a simple page a parent picks.
- **Safety and privacy:** parent chooses the page and sits alongside; no logging in anywhere.
- **The fun:** peeking behind the curtain.
- **The skill:** reading a simple page's HTML, and asking who made it and how.
- **The crumb:** three tags found and what each does.

Right-click a plain page (a simple MDN example page works well) and choose “View page source.” The student finds three tags they know and writes what each does. Big commercial pages are a jungle of code; choose something small. **13–14:** Use “Inspect” to change a heading's text on screen, then refresh and watch it go back. Nothing you change in your own browser touches the real site, which is a useful thing to know.

In-home — PROJ-01 Project One-Pager

- **Time:** 20 minutes.
- **Materials:** one sheet of paper.
- **Safety and privacy:** none needed.
- **The fun:** planning a real mini-site.
- **The skill:** problem, user, features, test plan, and schedule on one page.
- **The crumb:** the one-pager.

For a 13–14 three-page site, the student writes who the site is for, what each page does, how they'll test it (the access checklist and two first users), and which Hour each part gets done.

In-home — PROJ-03 Demo Day

- **Time:** 20 minutes.
- **Materials:** the finished page or site on the family computer, a feedback sheet.
- **Safety and privacy:** family or co-op only; the page stays local.
- **The fun:** an audience.
- **The skill:** showing a project and collecting user feedback.
- **The crumb:** feedback notes.

Two or three family members try the page while the student watches without explaining. Each visitor answers two questions on the sheet: “What did you look at first?” and “What was confusing?” The student picks one change to make, makes it, and says why. That last step is the standard's “improve based on user feedback” in action.¹⁷

Out-of-home — OUT-01 Algorithms in the Wild

- **Time:** 20 minutes on an errand or walk.
- **Materials:** a pocket notebook.
- **Safety and privacy:** road rules; stay together in stores and parking lots; no photos of strangers or staff.

- **The fun:** reviewing the world’s “pages.”
- **The skill:** reading a real design and writing its rules in plain words.
- **The crumb:** a page of notes with one redesign sketch.

This week the walk becomes “who made this sign?” Signs, posters, menus, and screens are all designed for an audience, just like web pages. The student picks three and asks: Who is this for? What do you see first? What does it want you to do? Is anything hard to read? Then they write the sign’s “rules” as pseudocode (a plan for a program in plain words): “IF a person walks in THEN they see the hours first.” Back home, they sketch one improved version and label it with HTML tags. A self-checkout screen or a bus-stop timetable makes a great subject.

Talk box (this chapter)

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

Follow-ups: - Prediction: “When you refresh, what will change?” - Evidence: “What did you see on the screen that tells you that?” - Location: “Which tag is the last one that’s showing the way you expected?” - Change: “What’s the one thing you’d change so your visitor finds that sooner?” - Revision: “Want to change your prediction? Save, refresh, and see.”

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 page or the code, not at the student.⁵

What a stuck silence usually means. The question was too broad (“Is it good?”); ask about one part. They may not have thought about the visitor at all yet, which is normal; offer a specific visitor (“your grandmother on her phone”). They may be worried their design looks childish; keep comments about what a visitor can find and read. Or the page is so busy with colors that the question got lost; look at the wireframe instead.

Don’t say / say instead. Don’t say “That color’s ugly.” Say “Can you read this paragraph easily from across the table? Let’s check.”

Blocked, then mixed

Spend a few Hours on HTML alone until opening and closing tags feel automatic. Then CSS alone, using a page that’s already finished. Then mix: a mystery page with one HTML mistake and one CSS mistake, and the student has to say which layer each lives in. Knowing “is this structure, style, or behavior?” is half of web debugging.

One incorrect example to diagnose

Here is an illustration, not a real family. Show your student this setup from an imaginary student. The folder holds `index.html` and `style.css`. The HTML has:

```
<link rel="stylesheet" href="styles.css">
```

and the CSS has:

```
H1 {  
  colour: blue;  
}
```

The student says, “None of my styles work.” Ask: “How many problems can you find?” There are two that matter. The link line names `styles.css`, which doesn’t exist, so no styles load at all. And `colour` is spelled the British way; CSS only understands `color`, so even with the right file name, that line would be skipped. (The capital `H1` happens to work, but lowercase is the habit.) A student who finds both has learned the two most common CSS bugs in one go.

A week without a screen

Web design is unusually good on paper. Wireframes are paper by nature. For HTML, write tags on index cards (`<h1>`, `</h1>`, `<p>`, `</p>`) and have the student build a page on the table, matching every opener with its closer. Then a sibling or parent plays “browser”: they read the cards and draw what the page would look like, following the tags literally. If a closing card is missing, the “browser” keeps the whole rest of the page bold, just as a real one might. The paper box model (WEB-04) and the “who made this sign?” walk need no screen either. A student who has played browser on paper reads real HTML much faster.

Encouragement that means something

Name the move you saw:

- “You checked the link line before changing anything else. That saved you ten minutes.”
- “Your alt text tells me exactly what’s in the picture. Someone using a screen reader would know.”
- “You watched your grandmother use the page without explaining it. That’s hard, and it’s what designers do.”
- “You looked that property up on MDN yourself.”

Avoid praising the page as “cool” or “professional-looking” and stopping there. The student can’t repeat “cool” on purpose. They can repeat checking the link line.

Siblings and co-ops

A family or co-op “web day” works well: each student builds one page on a shared theme (a hobby, a book, a local bird), and the pages link to each other through a shared index page on one computer. Everyone tests everyone else’s page with the access checklist. Feedback rules: about the page, specific, and invited.

For the student

Every site you’ve ever visited started as a blank file. Somebody typed some text, saved it, and opened it in a browser. You can do the same thing today, with tools already on your computer.

A web page has three layers:

- **HTML** is the structure: headings, paragraphs, lists, links, pictures.
- **CSS** is the style: colors, fonts, spacing.
- **JavaScript** is the behavior: things that happen when you click.

HTML uses **tags** in angle brackets. Most come in pairs, an opening tag and a closing tag with a slash:

```
<h1>My Bird Report</h1>
<p>I counted birds for two weeks.</p>
```

h1 is the main heading. p is a paragraph. Save this inside a full page skeleton as `index.html`, double-click it, and your browser shows it. That's a real web page. It's on your computer, not the internet, and nobody else can see it unless your family decides otherwise.

A worked example

You want your heading to be dark blue. You make a file called `style.css` in the same folder and write:

```
h1 {
  color: rgb(20 80 140);
}
```

You add `<link rel="stylesheet" href="style.css">` to the `<head>` of your page. Predict: the heading turns blue. Save both files, refresh the browser, and look. If it worked, great. If nothing changed, write Expected/Actual, then check the link line: does the file name match exactly? One wrong letter stops all your styles from loading.

Make it work for everyone

Some people can't see the screen and use a screen reader, which reads the page aloud. Help them by:

- giving every picture alt text that says what it shows: `alt="Pencil drawing of a bird feeder with two sparrows";`
- using headings in order (h1, then h2) instead of just making text bigger;
- writing links that say where they go, not "click here";
- choosing text colors that stand out clearly from the background.

This isn't extra credit. It's good manners, in code.

Save, refresh, look

Web building has a rhythm. Type a few lines. Save. Switch to the browser. Refresh. Look. Predict before you refresh: what should change, and where? If you wait until you've typed fifty lines before looking, and something's wrong, you'll have fifty places to search. If you look after every three lines, you'll have three.

Look it up like a pro

Professional web developers don't memorize every tag and property. They look things up constantly. When you forget how to make a link or center a heading, open MDN Web Docs with your parent and search for it. Read the short example at the top of the page, try it in your own file, and change one thing to see what happens. That's how you learn the tags you actually need, faster than any list.

You can also learn by reading other people's pages. Right-click a simple page your parent picks and choose "View page source." You'll see the same tags you've been writing. Every page on the web is built from them.

Try this today

Draw a wireframe of a one-page site about something you know well. Label each box with its tag. Then build it: headings, one paragraph, one list, one link. Save and refresh after every few lines.

Explain it back

Tell your parent what HTML, CSS, and JavaScript each do, using your own page as the example. Point to one line of each.

This part is hard for everyone

The hardest part of web pages at first isn't the code. It's the files. Saving in the wrong folder, forgetting to save, a file secretly named `index.html.txt`, a link line with one wrong letter. Every web developer has stared at a page that wouldn't change and then realized they were editing a different copy. When nothing happens, check the file before you check the code.

Challenge

Make a second page and link the two pages to each other. Use the same `style.css` for both, so changing one color changes it on both pages.

If you're 13 or older: add a button with JavaScript that changes some text when clicked (WEB-06). Then run a demo day: watch two people use your site without explaining it, write down what confused them, and fix one thing.

What to save

- Your wireframe sketch.
- `index.html`, `style.css`, and (if you made one) `script.js`, all in one project folder.
- Your access checklist.
- Your demo-day feedback notes.
- Your Debug Log entries.

Keep the whole folder in two places. Pictures on your page should be ones you made, or ones you’ve credited and have permission to use. A page or picture made by a tool isn’t your work, even if you asked for it.

If it isn’t clicking

“Nothing changes when I refresh.” Check the file first: Is it saved? Is it the file the browser is showing (look at the address bar)? Is it really called `index.html`? If the computer hides file endings, a file can end up named `index.html.txt` without anyone noticing. Turning on “show file extensions” in the computer’s settings is worth doing once, with a parent.

“My CSS doesn’t work.” If *nothing* is styled, check the link line and the file name. If *one* rule fails, check the selector’s spelling and that every `{` has a matching `}` and every line ends with `;`. Look the property up on MDN to check its spelling.

“My button does nothing.” JavaScript often fails silently on the page. Open the Console with a parent (right-click, Inspect, Console) and read the red text: last line first, then the line number. Check that the `id` in the HTML matches the name in the script exactly, including the `#`.

“I don’t know what to make.” That’s a planning problem, not a coding one. Go back to WEB-01 and ask: who is this for? A page for a younger sibling, a grandparent, or a pet’s fan club is easier to start than a page for “everyone.”

Use the rescue order throughout: ask → wait → hint → short demo on your own copy → hand it back. If a week of HTML is still frustrating, step back to paper wireframes and tag labels until the structure feels natural.

Tools, including AI

A **plain text editor** and a **browser** are all this chapter needs, with no installs and no accounts. Pages stay on your computer.

MDN Web Docs is free and is the reference for HTML, CSS, and JavaScript.⁴⁷ Its “Learn web development” section is a full beginner course if your student wants more.

The Raspberry Pi Code Editor is a free browser-based editor from the Raspberry Pi Foundation that includes HTML projects. It can be used without an account; saving work there needs one, and for children under 13 a parent sets up the account with the parent’s email.³⁵ The rules have been changing, so check the current rules before using it.

CodeAI (formerly Code.org) offers Web Lab inside its free middle-school courses, CS Discoveries and the newer AI Discoveries. It’s an optional course path, not a requirement. A free CodeAI account is required. At 11–12, a parent creates and holds the account. CodeAI describes its AI Tutor as for students 13 and up, and its newer Web Lab includes AI-assisted features, so keep any AI helpers inside the course off at 11–12.^{54,55,11}

AI. For the rules, see the AI box in **The Computing Hour**. What’s different for web work:

- Pages made by an AI are made by a tool and don't go in the portfolio as the student's work.
- **13–14:** After looking a CSS property up on MDN first, your student may ask an AI, with you nearby, to explain it in other words, and then check that explanation against MDN. Any code an AI suggests goes through AI-07 Explain Every Line, with a “Help I used” line and a Whose Work card.
- **11–12:** Parent-operated only, as always.

Check the current terms of any service before anyone uses it.

What “done enough” looks like

Your student has met this chapter's goals when they can:

- build a page from a blank file with headings, paragraphs, a list, and a link;
- style it with a separate CSS file and explain what a selector is;
- label the four layers of the box model;
- check a page for alt text, heading order, readable contrast, and clear links;
- find and fix a broken link line or a misspelled property;
- at 13–14, add a button with JavaScript and improve the site after a demo day.

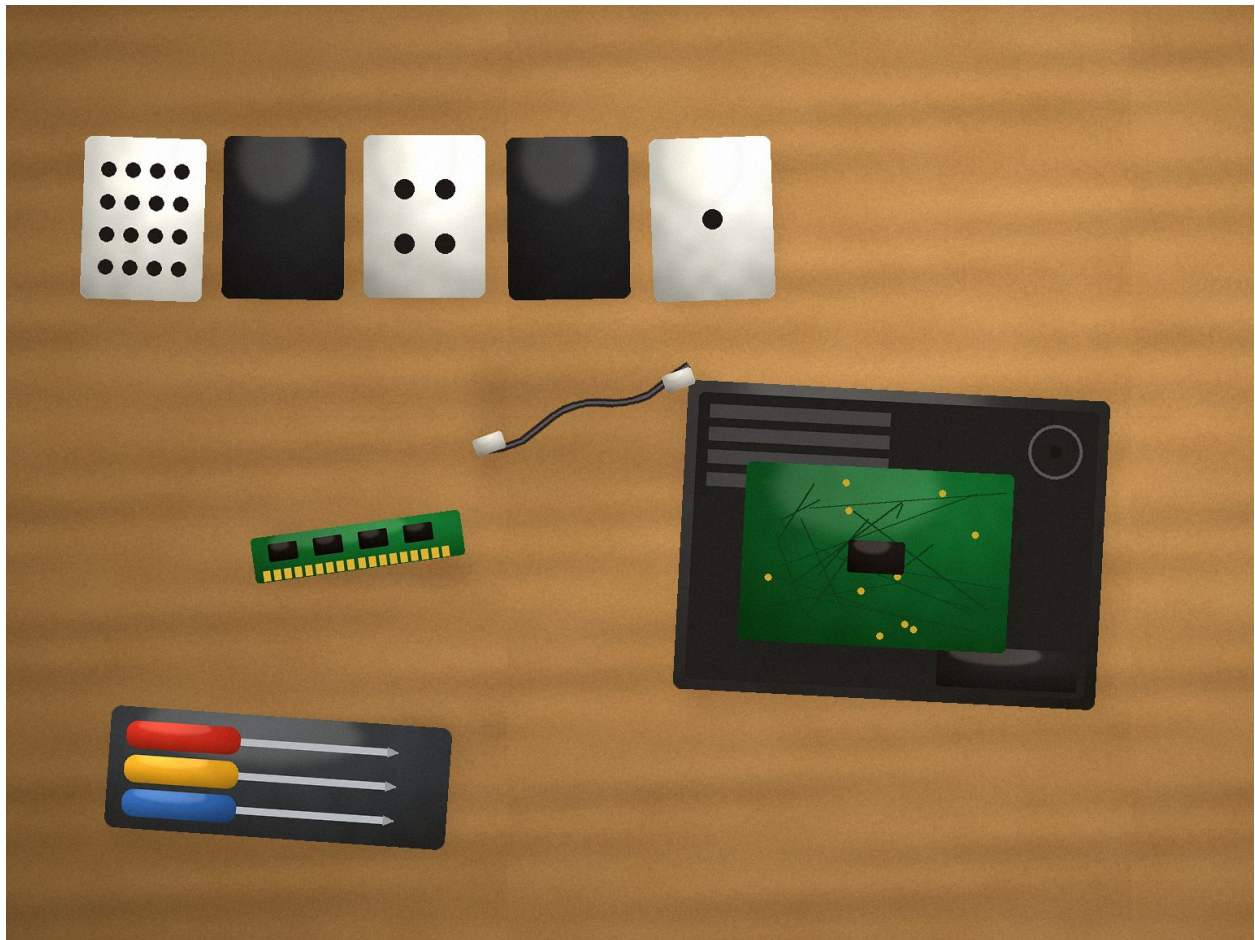
AS checks. Try AS-01 Cold Trace with a short HTML snippet (“What will this look like?”), AS-02 Debug on the Spot with a page that has one planted bug, and AS-03 Explain Your Code on the student's own page. All three are AI-free.

Placement by skill. A twelve-year-old who styles a page confidently can try WEB-06. A fourteen-year-old who still loses track of files needs more time with folders and file names (Chapter 6 helps), and that's worth the time.

The life of the habit. A small personal page is a lovely thing to keep updating over a year: new sections, better styling, a button or two. Every version is a record of growth. Keep the old versions in a dated folder.

Chapter 6

How Computers and Networks Really Work — and Hardware You Can Program If You Like



Why this matters

Everything on a computer is a number, including this sentence. Every letter, every pixel of every photo, every note of every song is stored as a pattern of two states: on or off, written 1 or 0. This

chapter shows your student how that works, and it's one of the most satisfying ideas in computing, because it turns the machine from magic into something they can explain.

Few people of any age can say what happens between tapping a link and seeing a page. By the end of this chapter, your student can. They'll count to 255 with eight paper cards, draw a picture from a grid of digits, act out a message crossing the internet in pieces, and scramble a secret with a paper wheel.

The national computer science standards for ages 11–14 cover all of this: how data is represented and how precise it needs to be; what devices do and how their parts work together; how information is broken into packets, sent, and put back together; why the internet keeps working when parts of it fail; and why keeping data private, accurate, and available matters.¹⁷ In plain words: what is the computer actually doing, and where does the information go next?

An optional strand uses a small programmable board called a micro:bit, with buttons, lights, and sensors, programmed in Python. You don't need to buy anything; a free browser simulator does nearly everything the board does.

Understanding the machine also makes your student safer: a student who knows what the little lock in the browser does and doesn't mean is harder to fool.

For the parent: understand it yourself

Numbers made of two digits

The everyday picture. A row of light switches. Each is either on or off. With one switch you can show two things. With two switches, four (off-off, off-on, on-off, on-on). Every switch you add doubles the count.

The precise picture. Each switch is a **bit**, a 1 or a 0. Eight bits make a **byte**. The bits stand for place values that double from right to left: 1, 2, 4, 8, 16, 32, 64, 128. Add up the places that are “on.” So 00001101 is $8 + 4 + 1 = 13$. All eight on, 11111111, is $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$. That's why so many computer numbers stop at 255: it's the biggest number one byte can hold, and with zero there are 256 possible values.

A pattern of bits means something because everyone agreed what it means. The same eight bits can stand for a number, a letter, or a shade of red.

Text, pictures, and color

Text. Each character has an agreed number, called a character code. Python shows it with `ord()`: `ord("H")` gives 72, and `chr(72)` turns it back into "H".⁴⁴

```
for letter in "Hi!":
    code = ord(letter)
    print(letter, code, format(code, "08b"))
```

This prints H 72 01001000, then i 105 01101001, then ! 33 00100001. Each letter is a number, and each number is eight bits.

Pictures. A screen is a grid of tiny squares called **pixels** (short for “picture elements”). In a black-and-white picture, each pixel can be one bit: 1 for black, 0 for white. In a color picture, each pixel usually gets three numbers, for red, green, and blue, each from 0 to 255.⁴⁸ Your student met those numbers in Chapter 5’s CSS: `rgb(20 80 140)` is a little red, more green, and a lot of blue.

Squeezing it. Pictures take a lot of numbers, so computers compress them, storing the same picture in less space. One simple method is run-length encoding: instead of “white, white, white, white, black, black, white, white,” you write “4, 2, 2,” always starting with white. Fax machines relied heavily on this idea, and a form of it still works behind the scenes in photo formats.⁵⁶ It works well for long runs of one color and badly when colors change every pixel.

The parts of a computer

- The **processor** follows instructions and does calculations, one tiny step after another, very fast.
- **Memory** is fast, short-term space for whatever is happening right now.
- **Storage** (a drive) keeps files for later, even after you switch off, but works more slowly. Unsaved work lives only in short-term memory; saving writes it to storage.
- The **operating system** is the manager. It handles the keyboard and mouse, draws on the screen, moves things around in memory, and decides which program gets which part of the machine.⁵⁷

Some computers also have **graphics chips**: processors with thousands of small parts that do huge numbers of simple sums at the same time. They were built to draw pixels quickly, and the same trick makes them good at training AI.⁵⁸ Big data centers full of those chips use a lot of electricity; the International Energy Agency estimated that data centers used about 1.5 percent of the world’s electricity in 2024.⁵⁹ Chapter 8 returns to this.

Files and folders. Storage is organized as a tree of folders. A file’s full address, its **path**, lists the folders from the top down. The ending after the dot, the **extension** (`.py`, `.html`, `.csv`), tells the computer what kind of file it is. Good names help: `2026-10-06-quiz-v2.py` sorts itself by date and says what it is. And every important file lives in two places.

How the internet moves information

The everyday picture. Mailing a long letter as numbered postcards. Each postcard carries the address, its number, and how many there are in total. They may travel by different trucks and arrive out of order. The receiver sorts them by number and asks again for any that went missing.

The precise picture. That’s almost exactly how it works. Data travels in small chunks called **packets**. Each one carries the sending and receiving addresses and its place in the order. Packets can take different paths and arrive out of order; the receiving computer puts them back together and asks for any that are missing.⁵² **Routers** pass packets along, using **IP addresses**, which work

much like street addresses. Your home “router” is often three devices in one box: a switch, a router, and a modem that connects to your internet provider.⁵¹

People don’t remember number addresses, so the internet has an address book. The **Domain Name System (DNS)** turns a name like `developer.mozilla.org` into the IP address of the right computer.⁵² The internet is the network itself; the web is one service that runs on top of it.⁵¹

Why doesn’t the internet stop when one cable breaks? There are many routes; if one fails, packets find another. MDN describes the internet’s design as making sure that, whatever happens, computers find a way to stay connected.⁵¹

Keeping secrets

Encryption means scrambling a message so only the right person can unscramble it. The oldest classroom example is the **Caesar cipher**: shift every letter three places, so HELLO becomes KHOOR. It’s fun and very weak. In English there are only 25 useful shifts, so anyone can try them all in a few minutes.

Modern encryption uses an idea that sounds impossible at first: **public-key encryption**. Think of a padlock. You can hand out open padlocks to anyone, but only you have the key. Anyone can snap your padlock shut on a box, but only you can open it. In the real system, keys come in pairs; one is public, one is kept private. Anyone can encrypt a message to the owner of the private key, but only that owner can decrypt it. Real systems commonly use this padlock idea to share a secret key safely, then use that faster shared key for the messages themselves.⁶⁰ Longer keys are stronger for the same reason more switches make more patterns: every extra bit doubles the number of possible keys.

The lock in the browser. A web address starting with `https` means the connection is encrypted, which keeps others from reading the data as it travels.⁶¹ It does *not* mean the site is honest. The FBI has warned that scammers use these locks too, and advised: “Do not trust a website just because it has a lock icon or ‘https’ in the browser address bar.”⁶² A private connection to a fake site is still a fake site. Chapter 7 teaches how to spot one.

Wrong answers you should be able to hear

- **“Binary is just 0 and 1.”** True but incomplete. Reply: “Right, so how does 01001000 become the letter H?” The meaning comes from place value and agreement.
- **“The lock icon means the site is safe.”** Reply: “It means nobody can read what we send on the way. Does it tell us who’s on the other end?”
- **“Memory and storage are the same thing.”** Reply: “What happens to a document you never saved when the power goes out?”
- **“The internet is in the air.”** Wi-Fi is the last few feet. Reply: “Where does the signal go after the router? Let’s go look for the cables.”
- **“My Caesar message is uncrackable.”** Reply: “How many different shifts are there? How long would it take to try them all?”

Your five-minute warm-up

Make the eight binary cards described below and show yourself 13 and 255. Then run the `ord()` loop above in Thonny with your own first name. Practice the opening question aloud once, and count to five in your head.

How to teach it this week

This chapter is three to four weeks of work, best taken in three parts: data as numbers, the machine, and networks and secrets. The optional micro:bit strand can run alongside any of them.

Many of these Hours are unplugged, and that's not a lesser version: paper is the clearest way to see what the machine does. Use the usual steps:

- **Unplug and Plan.** Cards, grids, wheels, and role cards.
- **Predict.** "What number will these cards make?" "Which packet will arrive first?"
- **Build and Test.** Do the activity, then check it with a short Python line or by decoding.
- **Debug and Log.** Expected/Actual for every surprise.
- **Save and Explain.** "I predicted... It did... Next I'd change..." and "Help I used: ..."

In-home — BIN-01 Binary Cards to 255

- **Time:** 15 minutes.
- **Materials:** eight index cards and a marker.
- **Safety and privacy:** none needed.
- **The fun:** counting higher than you'd think with only eight cards.
- **The skill:** five, then eight, binary cards; bits and bytes.
- **The crumb:** a card chart.

On the cards, draw dots: 1, 2, 4, 8, 16, 32, 64, 128. Lay them in a row with 128 on the left. A card face up is a 1; face down is a 0. Ask for 13: the student turns up 8, 4, and 1, and reads 00001101. Then: "What's the biggest number you can make?" (All up: 255.) "What happens if we add a ninth card? What would it be worth?"

11–12: Start with five cards (1 to 16) and count from 0 to 31 in order, turning cards like an odometer. Move to eight once the pattern clicks. **13–14:** Go straight to eight, then write numbers in binary on paper and check with Python's `format(13, "08b")`.

How to fade your help

Use the rescue order: **ask** → **wait** → **hint** → **short demo on your own copy** → **hand it back**. For number work, the best hint is a smaller example: "Show me 3. Now show me 6. What did you notice?"

11–12: Five-then-eight-bit cards, black-and-white pixel grids, and the Caesar wheel. Skip compression and the padlock exchange unless the student asks.

13–14: Add color numbers, compression, the padlock box exchange, and why longer keys are stronger. If you have a micro:bit or want to use the simulator, add sensor projects. The student leads; you're **first user** and safety-keeper.

Ready 12: A twelve-year-old who explains why eight bits stop at 255 is ready for BIN-04 and NET-04.

A worked Hour (illustration)

Here is an illustration, not a real family. Picture a thirteen-year-old and his mother at the kitchen table. In Unplug and Plan, he cuts a message into six numbered strips. His mother asks the opening question: “What is the computer actually doing — and where does the information go next?” He says, “Breaking it into pieces. They go to the router.” She counts to five in her head. “And then?” “Different routers. They might go different ways.” In Predict, he guesses the pieces will arrive in order. They play NET-01 with his little sister as a second router. Piece 4 arrives first; piece 2 “gets lost” behind the couch. He has to ask for it again. In Save and Explain he says: “I predicted they’d come in order. They didn’t. Next I’d add a ‘total number of pieces’ to each strip so you know when you have them all. Help I used: none.” That’s the real packet design, discovered at a kitchen table.

Practice that actually builds learning

In-home — BIN-02 Your Name in Numbers

- **Time:** 15 minutes.
- **Materials:** a printed character-code chart for capital and lowercase letters (or Thonny to look them up), paper.
- **Safety and privacy:** first names only.
- **The fun:** a secret-looking name tag.
- **The skill:** text as numbers.
- **The crumb:** the encoded name.

The student writes their first name as character codes ($H = 72$, $i = 105$), then as eight-bit binary using the cards. A sibling decodes it. **13–14:** Write the `ord()` loop from the parent section and check the paper work against it.

In-home — BIN-03 Pixel Grid Picture

- **Time:** 20 minutes.
- **Materials:** graph paper, pencils.
- **Safety and privacy:** none needed.
- **The fun:** a picture that’s secretly a list of numbers.
- **The skill:** images as grids of numbers; color as three numbers.
- **The crumb:** the grid sheet.

On an 8-by-8 grid, the student draws a simple shape (a heart, a house, a cat face). Then they write each row as 1s and 0s. A parent decodes the rows onto a blank grid without seeing the original.

11–12: Black and white only. **13–14:** Pick three colors and write each as red, green, and blue numbers, then try them in a CSS file from Chapter 5.

In-home — BIN-04 Squeeze It

For 13–14; ready 12.

- **Time:** 15 minutes.
- **Materials:** the pixel grid from BIN-03.
- **Safety and privacy:** none needed.
- **The fun:** making a picture smaller without losing anything.
- **The skill:** run-length compression.
- **The crumb:** before and after counts.

Rewrite each row of the grid as run lengths, starting with white: `WWWBBWW` becomes 4, 2, 2. If a row starts with black, it begins with 0. Count how many numbers the original used (64) and how many the squeezed version uses. Then design a grid that compresses *badly* (a checkerboard). In Python:

```
row = "WWWBBWW"
runs = []
count = 1
for i in range(1, len(row)):
    if row[i] == row[i - 1]:
        count = count + 1
    else:
        runs.append(count)
        count = 1
runs.append(count)
print(runs)
```

It prints `[4, 2, 2]`. Challenge: what does it print for `"BBWW"`, and how would you fix it to start with 0?

In-home — SYS-01 Input-Process-Output Hunt

- **Time:** 15 minutes.
- **Materials:** a clipboard.
- **Safety and privacy:** look, don't open or unplug anything.
- **The fun:** finding computers hiding in ordinary things.
- **The skill:** inputs, processing, outputs, and storage in household devices.
- **The crumb:** a labeled list.

For each device in the house (microwave, thermostat, washing machine), the student writes its inputs, what it processes, its outputs, and anything it stores (a saved time, a favorite setting).

In-home — SYS-02 The Operating System Plays Manager

- **Time:** 15 minutes.
- **Materials:** role cards: Manager, three Programs, Memory (a small tray), Screen (a white-board).
- **Safety and privacy:** none needed.
- **The fun:** a play where the boss is the operating system.
- **The skill:** how the operating system shares memory, time, and files among programs.
- **The crumb:** the role cards.

The student plays the operating system. Siblings or parents are programs asking for turns: “I need the screen!” “I need space in memory!” The manager gives each a turn and a space on the tray, and says no when the tray is full.

In-home — SYS-03 Folder Tree and File Names

- **Time:** 15 minutes.
- **Materials:** the family computer, with a parent.
- **Safety and privacy:** student’s own folders only; save in two places (see Family safety basics in **The Computing Hour**).
- **The fun:** taming a messy desktop.
- **The skill:** folders, extensions, date-first names, and two places.
- **The crumb:** a screenshot of the folder tree, cropped to the folders.

Draw the student’s project folders as a tree on paper first. Then build it: one folder per chapter or project, files named with the date first. Turn on “show file extensions” together if it’s off. Finally, copy the whole folder to a second place the family controls, like a USB drive or the family drive.

In-home — SYS-04 Inside an Old Computer

- **Time:** 30 minutes.
- **Materials:** an old desktop computer, a screwdriver, a clear table.
- **Safety and privacy:** an adult opens it, and it stays unplugged the whole time. Leave batteries, the power supply box, and any old tube-style screen alone; those stay closed. Wash hands afterward.
- **The fun:** seeing the real parts.
- **The skill:** naming the processor, memory, and storage.
- **The crumb:** a labeled sketch.

With the case open, the student finds and sketches the processor (often under a fan or metal fins), the memory sticks, and the storage drive, with arrows for how information moves. No old computer? A library book with clear photos works fine.

In-home — NET-01 Packet Relay

- **Time:** 20 minutes.

- **Materials:** a short message written on paper, scissors, two or three helpers.
- **Safety and privacy:** clear floor space.
- **The fun:** a relay race where the message breaks apart.
- **The skill:** messages split into numbered packets that take different routes.
- **The crumb:** the reassembled message.

Cut a message into six numbered strips marked “to” and “from.” Helpers (siblings work well) stand around the house as routers and pass strips along by different paths. The receiver reassembles them in order. Hide one strip; the receiver must notice it’s missing and ask for it again.

In-home — NET-02 The Internet’s Address Book

- **Time:** 15 minutes.
- **Materials:** index cards.
- **Safety and privacy:** make up addresses; no real ones.
- **The fun:** playing the internet’s phone book.
- **The skill:** IP addresses and DNS as a lookup.
- **The crumb:** a lookup chain sketch.

Write made-up site names on cards (“birdfacts”). A parent holds the “address book” matching each name to a made-up number address (like 10.0.0.7). The student looks up a name, gets a number, and delivers a message to the “computer” (a family member) with that number. Then sketch the chain: name, address book, number, computer.

In-home — NET-03 Caesar Wheel

- **Time:** 20 minutes.
- **Materials:** two paper circles (one smaller), a brad fastener, a marker.
- **Safety and privacy:** practice messages only, never real passwords.
- **The fun:** secret messages.
- **The skill:** encrypting and decrypting; why 25 tries break it.
- **The crumb:** the paper wheel and a solved message.

Write the alphabet around both circles, turn the inner one three places, and encrypt a message. Then the real lesson: break a message *without* being told the shift, trying shifts one at a time. How long did it take? That’s why nobody uses this for real secrets.

13–14: Write the Caesar shift in Python:

```
message = "HELLO"
shift = 3
secret = ""
for letter in message:
    number = (ord(letter) - ord("A") + shift) % 26
    secret = secret + chr(number + ord("A"))
print(secret)
```

It prints KH00R. Then write a loop that tries all 25 shifts on a secret message and prints each one. The right answer jumps out.

In-home — NET-04 Padlock Box Exchange

For 13–14; ready 12.

- **Time:** 15 minutes.
- **Materials:** a small box, a padlock with a key, and a note.
- **Safety and privacy:** none needed.
- **The fun:** solving a puzzle that sounds impossible.
- **The skill:** the public-key idea with a padlock and a box.
- **The crumb:** a diagram.

The puzzle: you want a secret note, but you can't meet the sender to give them a key. Solution: send your *open* padlock and keep the key. They lock the note in the box with your padlock and send it back. Only you can open it. The open padlock is the public key; your key is the private key. The student draws the exchange as a diagram with arrows.

In-home — NET-05 Look for the Lock

- **Time:** 10 minutes.
- **Materials:** a parent's device and browser.
- **Safety and privacy:** parent drives; visit only sites the parent chooses; no logging in.
- **The fun:** finding out what the little lock really means.
- **The skill:** what HTTPS does and doesn't mean.
- **The crumb:** a one-sentence note.

On a site you trust, find the lock and the https. Ask: "Could a fake site have a lock too?" (Yes.) The student writes one sentence: "The lock means the connection is private; it doesn't tell you the site is honest." Chapter 7 picks this up.

Out-of-home — OUT-04 Network Spotting

- **Time:** 20 minutes on a walk.
- **Materials:** a pocket notebook.
- **Safety and privacy:** road rules; stay on sidewalks; look, don't touch or climb; no photos of strangers.
- **The fun:** finding the internet, which turns out to be very physical.
- **The skill:** spotting routers, cables, and towers.
- **The crumb:** a spotting list.

Find the router and the cable that leaves the house. Outside, look for cables on utility poles, gray boxes at the curb, and cell towers on the skyline. Back home, draw a map from your router to "the internet," with question marks where you're not sure.

The optional strand: a computer you can hold

A **micro:bit** is a small programmable board with two buttons (A and B), 25 lights in a 5-by-5 grid that also act as a light sensor, a temperature sensor inside its processor, a motion sensor, a compass, and a radio. The newer version (V2) adds a touch-sensitive logo, a microphone, and a speaker. It runs on two AAA batteries or a USB cable.⁶³ Students write real Python for it in the free micro:bit Python Editor, which runs in the browser and needs no logins or passwords.⁶⁴

Everything below has a simulator path. No hardware is required. The programs follow the micro:bit's official MicroPython documentation but weren't tested on a physical board, so treat any surprise as a debugging job.

In-home — PHYS-04 Simulator First

- **Time:** 20 minutes.
- **Materials:** a computer with a browser; a parent opens the micro:bit Python Editor.
- **Safety and privacy:** no account needed; save the file to the family drive.
- **The fun:** programming a gadget you don't own yet.
- **The skill:** programming a micro:bit in the browser simulator.
- **The crumb:** the program file.

The editor's simulator shows a micro:bit on screen with clickable buttons and sliders for light, temperature, and other sensors, and it keeps working without internet once loaded.⁶⁴ The simulator shows the V2 board, so a few programs may not run on an older V1. Try PHYS-01 here first.

In-home — PHYS-01 Name Badge That Scrolls

- **Time:** 20 minutes.
- **Materials:** a micro:bit and USB cable, or the simulator.
- **Safety and privacy:** a first name or a nickname only.
- **The fun:** a badge that talks back.
- **The skill:** a micro:bit program with buttons as events.
- **The crumb:** the program file.

```
from microbit import *

while True:
    if button_a.is_pressed():
        display.scroll("HELLO")
    if button_b.is_pressed():
        display.show(Image.HAPPY)
    sleep(100)
```

Press A and HELLO scrolls across the lights. Press B and a smiley face appears.⁶⁵ Change "HELLO" to a first name. Predict first: what happens if you press both buttons?

In-home — PHYS-02 Sensor Logger

- **Time:** 25 minutes.
- **Materials:** a micro:bit V2 (data logging is a V2 feature), or the simulator.
- **Safety and privacy:** logging stays on the device; nothing is sent anywhere.
- **The fun:** a gadget that collects data while you're away.
- **The skill:** reading a light or temperature sensor and logging data for a spreadsheet.
- **The crumb:** the data file.

```
from microbit import *
import log

log.set_labels("temp", "light")

while True:
    if button_a.is_pressed():
        log.add(temp=temperature(), light=display.read_light_level())
        display.show(Image.YES)
        sleep(500)
        display.clear()
```

Each press of A records a temperature (in degrees Celsius) and a light level. On a real V2 board, the data is saved in a file called MY_DATA on the micro:bit's drive, which opens as a table in a browser and can be copied into a spreadsheet.⁶⁶ Then the spreadsheet skills from Chapter 4 take over: "Is the kitchen windowsill warmer than the hallway?"

In-home — PHYS-03 Reaction Game

For 13–14; ready 12.

- **Time:** 30 minutes.
- **Materials:** a micro:bit or the simulator.
- **Safety and privacy:** first names only on any score list.
- **The fun:** a game the whole family will want to beat.
- **The skill:** variables, random timing, and buttons.
- **The crumb:** the program file plus the best time.

```
from microbit import *
import random

display.show("?")
sleep(random.randint(1000, 4000))
display.show(Image.HEART)
start = running_time()
while not button_a.is_pressed():
    sleep(1)
reaction = running_time() - start
display.scroll(str(reaction))
```

After a random wait, a heart appears; press A as fast as you can, and the board scrolls your time in milliseconds.⁶⁵ Then debug it like Chapter 3: what happens if someone holds A down before the heart appears? (They get a suspiciously perfect time.) How would you catch a cheater?

Talk box (this chapter)

Opening: “What is the computer actually doing — and where does the information go next?”

Follow-ups: - Prediction: “If you add one more card, what’s the biggest number now?” - Evidence: “What did you see that makes you think the packets took different paths?” - Location: “Where exactly is the information right now: memory, storage, or on its way somewhere?” - Change: “What’s one thing you’d change to make the secret harder to break?” - Revision: “Want to change your prediction? Let’s try 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 cards or the code, not at the student.⁵

What a stuck silence usually means. The question was too abstract; put a physical object in their hands (a card, a packet strip). They may be guessing what you want to hear; say “There’s no wrong guess here, only a guess we test.” They may not have a picture in their head yet; draw one together. Or the screen is louder than the question; close it and go back to paper.

Don’t say / say instead. Don’t say “No, the internet isn’t wireless.” Say “Let’s follow the signal. Where does it go after the router?”

Blocked, then mixed

Spend a few days on each part alone. Then mix them in one puzzle: “A photo you send to Grandma: where is it stored, how is it written as numbers, how does it travel, and who could read it on the way?” A student who can answer that has put the whole chapter together.

One incorrect example to diagnose

Here is an illustration, not a real family. An imaginary student writes: “I invented an unbreakable code. I shift every letter by 7 instead of 3, so nobody will guess it.” Ask your student: “Is it unbreakable? How would you break it?” (Try all 25 shifts; shift 7 is just one of them.) “What would make it stronger?” (More possible keys, so trying them all takes too long.) Real encryption uses keys with so many possibilities that trying them all isn’t practical.

For the student

Everything on a computer is a number, including this sentence.

Bits and bytes. A computer stores everything as switches that are on or off, written 1 and 0. Each one is a **bit**. Eight bits make a **byte**. The places in a byte are worth 1, 2, 4, 8, 16, 32, 64, and 128, doubling each time from the right. To read a byte, add up the places that are 1:

$00001101 = 8 + 4 + 1 = 13$

All eight on is 255. That's why you see 255 everywhere in computing, like in color numbers.

Letters are numbers too. Each character has an agreed code. In Python, `ord("H")` gives 72. So "Hi" is really 72 and 105, which in binary is 01001000 01101001.

Pictures are grids of numbers. Each tiny square on a screen is a pixel. In a color picture, each pixel has three numbers: how much red, green, and blue, each from 0 to 255.

A worked example

Let's follow a message you send to a friend. You type it; the keyboard is an **input**. The **processor** turns each letter into numbers. The message sits in **memory** while you write. When you press Send, the computer breaks it into **packets**, small numbered pieces, each with an address. Your **router** passes them on, and other routers pass them along, maybe by different paths. Your friend's device collects the packets, puts them in order by number, asks again for any that went missing, and shows the message. If the connection uses **encryption**, scrambling the message so only the right person can unscramble it, nobody along the way can read it.

Why this helps you. Knowing how things work makes you harder to fool. When you understand that a message travels through many machines, you see why encryption matters. When you know the browser lock only means "private connection," you won't trust a fake site just because it has one.

A computer you can hold. If your family uses the micro:bit simulator or a real board, you'll program buttons, lights, and sensors in Python. It's the same idea as everything above: inputs come in, the processor follows your instructions, and outputs come out.

Try this today

Make eight binary cards (1, 2, 4, 8, 16, 32, 64, 128). Show your age in binary. Then your birthday month and day. Then the biggest number you can make.

Explain it back

Explain to your parent why the lock in a browser's address bar doesn't prove a website is honest. Use the word "encrypted" correctly.

This part is hard for everyone

The padlock idea, where you can give away the way to lock something but keep the way to open it, feels wrong at first. Nearly everyone has to act it out with a real box and padlock before it makes sense. That's fine. That's what the activity is for.

Memory versus storage confuses people too. Here's the short version: memory is your desk (fast, for what you're working on now); storage is the filing cabinet (slower, but it keeps things). Saving

moves your work from the desk to the cabinet. That's why a power outage can lose the work you hadn't saved yet, but not the files you had.

Challenge

Encrypt a message with a Caesar wheel and give it to someone *without* telling them the shift. Time how long it takes them to break it. Then explain why that proves Caesar ciphers aren't safe for real secrets.

If you're 13 or older: a code with 8-bit keys has 256 possible keys. How many does a 9-bit key have? A 16-bit key? Work out the pattern, and explain why longer keys are stronger. If you have a micro:bit or the simulator, build the reaction game and then find a way to stop people cheating.

What to save

- Your binary card chart and your name in numbers.
- Your pixel grid and its rows written as numbers.
- Your labeled sketch of a computer's parts (or a device's inputs and outputs).
- Your packet relay message and your network spotting list.
- Your Caesar wheel and one solved message.
- Any micro:bit programs.
- Your Debug Log entries.

Save your files in two places, in folders with clear, date-first names.

If it isn't clicking

Binary doesn't make sense. Go back to five cards and count from 0 to 31 out loud, turning cards over like a car's odometer. The pattern of flips is the lesson. Skip conversion on paper until the cards feel easy.

Networks feel invisible. That's because they mostly are. Do NET-01 and OUT-04 before any explanation. Seeing the router and the cable leaving the house makes the rest concrete.

The padlock idea won't land. Act it out with a real padlock and box, twice, swapping roles. Then draw it. Then, and only then, use the words "public key" and "private key."

The micro:bit is frustrating. Go back to the simulator. Check that the program starts with `from microbit import *`, and read any error last line first, as in Chapter 3.

Use the rescue order: ask → wait → hint → short demo on your own copy → hand it back. Binary often clicks suddenly after a few days of not clicking.

Tools, including AI

Paper first. Index cards, graph paper, a brad fastener, a box and padlock. Most of this chapter needs nothing else.

CS Unplugged, from the University of Canterbury, is a free collection of computing activities without computers, including binary cards and picture compression, if you want more practice.⁵⁶

Thonny, the free offline Python editor, handles the `ord()`, binary, compression, and Caesar programs.²

The micro:bit Python Editor is free, runs in the browser, needs no logins, and includes a simulator.⁶⁴ A real micro:bit is optional; borrow one from a library or co-op before buying, and check the current price yourself.

AI. For the rules, see the AI box in **The Computing Hour**. Nothing in this chapter needs AI. At 13–14, with you present, a student might ask an AI to explain an idea like DNS in other words *after* doing the activity, then check that explanation against a real source like MDN. At 11–12, AI stays parent-operated. Check the current terms of any service.

What “done enough” looks like

Your student has met this chapter’s goals when they can:

- count to 255 with binary cards and explain why eight bits stop there;
- explain how text and pictures become numbers;
- name the processor, memory, storage, and operating system, and say what each does;
- organize files in folders with clear names, saved in two places;
- explain packets, routers, IP addresses, and DNS using the relay game;
- encrypt and break a Caesar message, and explain why it’s weak;
- say what the browser lock means and doesn’t mean;
- at 13–14, explain the padlock idea and why longer keys are stronger.

AS checks. Try AS-01 Cold Trace on the Caesar program, AS-02 Debug on the Spot with a planted bug in the compression code, and AS-03 Explain Your Code on any micro:bit program. All three are AI-free.

Placement by skill. A twelve-year-old who explains why 255 is the limit is ready for compression and the padlock. A fourteen-year-old still unsure about binary needs more card time, and there’s no shame in that.

The life of the habit. Keep asking the opening question whenever a new device comes into the house: what is it actually doing, and where does the information go? A student who asks that about every gadget is thinking like an engineer.

Chapter 7

Safe, Smart, and Kind Online: Passwords, Scams, Social Media, and Your Footprint



Why this matters

Here is an illustration, not a real family. A twelve-year-old is at the kitchen table on a Tuesday when a text arrives on the family tablet: “Your game account will be DELETED in 24 hours. Verify your

login now,” with a link underneath. She almost taps it. Then she slides the tablet across the table and says, “Is this real?”

That slide across the table is the whole chapter. She didn’t need to know every trick a scammer uses. She noticed the hurry, felt the little tug of worry, and brought it to someone she trusts. Everything in these pages makes that move easier and more automatic.

These are the years of first phones, first accounts, and first group chats. They are also the years when students can understand *why* the habits work. Habits learned now, with you beside them, last long after they’ve stopped asking your permission.

CSTA, the Computer Science Teachers Association, expects middle-schoolers to “evaluate common types of cyber attacks and preventions,” and to weigh the benefits and risks of letting personal data and metadata be collected.¹⁷ In ordinary words: know how people get tricked, know how to stop it, and know what you’re giving away when you share.

The tone here is calm. No horror stories, no scary videos. Four habits do most of the work: spot the phish, use strong passphrases, turn on two-step login, and keep things updated. Around them sit three more: think before you share, be kind when nobody is watching, and when something feels wrong, Stop, Save, Tell. None of this makes anyone perfectly safe. Nothing does. It gives your student a plan and keeps you in the loop.

A timing note. If a phone is arriving this month, teach this chapter now, out of order, starting with SAFE-08 Family Device Agreement. Chapters 4 through 6 will wait.

This week you can learn the four habits well enough to explain each in a sentence. **Today your student can try** sorting three printed messages into “phish” and “fine,” and saying why.

For the parent: understand it yourself

You don’t need to be a security expert. You need the few habits that matter most, why they work, and what to say when your student pushes back.

The everyday picture

Think about your front door. Most break-ins happen because a door was left open, a key was under the mat, or someone knocked, sounded official, and was let in. Most online break-ins work the same way: a reused password, a missing second lock, or a convincing knock: a message that says “urgent” and asks for your key.

The precise picture

An **account** is a locked room on someone else’s computer; logging in proves who you are. A **phishing** message is a fake knock: an email, text, or chat pretending to be a company, a friend, or a teacher, built to get a password, a code, money, or personal details. **Personal data** is anything that points to a real person: name, address, school, birthday, face, location. **Metadata** is data about data: a photo file can carry the date, the device, and sometimes the exact place it was taken.

Four habits that do most of the work

The Cybersecurity and Infrastructure Security Agency (CISA), a federal agency, boils home security down to “four easy ways”: recognize and report phishing; use strong passwords and a password manager; add a second login step; and update software.⁶⁷ This chapter is built on those four.

1. Spot the phish. Phishing messages hurry you (“within 24 hours”). They threaten a loss or promise a prize. They come from a sender that’s almost right, greet you vaguely (“Dear customer”), and ask for a password, a payment, a gift card, or a login code. The Federal Trade Commission’s sample scam email shows exactly these signs: a generic greeting, an account “on hold” for a billing problem, and a link to “update” payment details.⁶⁸ Your student doesn’t have to judge every message alone. The rule is simple: if it hurries you and asks for something, slow down and show a parent.

2. Strong passphrases, one per account. The federal password guidelines, from the National Institute of Standards and Technology (NIST), have changed in ways that make life easier. A password used on its own should be at least 15 characters; services shouldn’t demand mixtures of symbols and capitals; and nobody should be forced to change passwords on a schedule, only when one has leaked.⁶⁹ Length beats tricks. Four or five unrelated words strung together are strong and memorable. The real danger is reuse: one leaked password unlocks every account that shares it. So: a different passphrase for each account, and a password manager (an app that remembers them behind one strong passphrase) in the parent’s hands at 11–12 and shared at 13–14.

3. Two-step login. You’ll see it called **multifactor authentication**, which means logging in with a password plus a second thing. This book calls it two-step login. The FTC sorts the extra pieces into three kinds: something you know (a PIN), something you have (a code on your phone or a security key), and something you are (a fingerprint or face).⁶⁸ A stolen password alone then isn’t enough. Say one rule out loud at the table: a login code is typed only into a login page you opened yourself, never read out to someone who asks for it.

4. Updates on. Updates fix holes criminals already know about. Turn on automatic updates and don’t put off the ones that ask. At 11–12, the parent approves new apps.

Privacy settings and the photo that says too much

Every app and device has privacy settings: who can see the profile, who can send messages, whether location is shared. The company sets the defaults, not your family. A settings tour together once a semester (SAFE-06) is one of the most useful twenty minutes in this book.

A picture of your student in a team shirt in front of your house, with location data still in the file, tells a stranger a great deal. In this family, faces, school names, and towns stay out of anything shared beyond the family, and screenshots are cropped to the work. See Family safety basics in **The Computing Hour**.

The 13 shift

COPPA in two sentences. A federal law, the Children’s Online Privacy Protection Act, says websites and apps aimed at children under 13 (or that know they’re collecting from

a child that young) must get a parent's consent before collecting the child's personal information, and photos, videos, and voice recordings count.⁴ That protection ends at 13, which is why so many services set 13 as their minimum age.

The rule was updated in 2025, and companies have had to follow most of the new version since 22 April 2026.⁷⁰ COPPA governs how companies collect data. It doesn't rate content or make any site safe.

So 13 is a rule for companies, not a readiness test. The day before and the day after a thirteenth birthday, the same child is sitting at your table. The 13 shift changes what companies may do; your family's rules sit on top of it, and you decide how fast they change.

In this book, at 11–12 the parent creates and holds every login, and the student never signs up for anything alone. At 13–14, the student may hold passwords for family-approved tools, the parent keeps a sealed copy or a shared password-manager entry, and two-step login is on. First accounts happen together, under the service's terms, with you at the table.

Social media, phones, and age rules

You'll probably be asked this year, "When can I get Instagram?" Here is what the evidence says.

Platform minimums. Most social media platforms in the United States set 13 as their minimum age.⁷¹ So at 11–12, the answer is "not yet; the platform's own rules say 13."

Phones. The American Academy of Pediatrics' 2026 policy says research does not point to a specific age for teen smartphone readiness, and that it varies by teen. It suggests weighing digital media literacy, truthfulness with caregivers, the ability to handle conflict with friends, and practical needs. It asks caregivers to set and monitor privacy, download, content, purchase, and time settings, and notes that a flip phone or watch can limit access to apps a teen isn't ready for.⁶ The AAP's PhoneReady questions turn this into a conversation: Is your child responsible about things like homework? Do they say sorry when they've hurt a friend's feelings? How well do they follow the tech rules you already have? Are *you* ready to help?⁷²

The advisory. In 2023, the U.S. Surgeon General issued an advisory on social media and youth mental health. Its central line: "we cannot conclude social media is sufficiently safe for children and adolescents."⁷¹ That was a statement about the evidence at the time, not a law or a ban.

What teens report. In a 2025 national survey of teens 13 to 17, roughly three-quarters said they use YouTube every day, and 44 percent of 13- and 14-year-olds said they use Instagram, compared with three-quarters of 15- to 17-year-olds.⁷³

What this book does. This book doesn't set a phone age or a social media age of its own. The evidence doesn't give one. It hands you the platform minimums, the AAP's readiness questions, the advisory, and a family device agreement your student helps write (SAFE-08). That agreement is where your family's answer lives, and it can change as your student grows.

Your footprint, and kindness

A **digital footprint** is everything that can be found about a person online, including things other people posted. Some of it is permanent in practice, because screenshots outlast deletions. At 13–14, a Footprint Audit with a parent (SAFE-05) shows a student their own trail and usually starts a good conversation.

Kindness online is a skill. A message typed in a hurry lands on a real person who can't see your face or hear your tone. The habits are small: reread before sending; don't forward something about someone that they wouldn't want forwarded; and if a chat turns cruel, leave and tell someone. Your student doesn't have to fix the whole chat. They do have to not join in.

Stop, Save, Tell

Your family already has this routine from Family safety basics in **The Computing Hour**. At 11–14 it grows up a little.

Stop. Stop replying. Don't argue, pay, or send anything. **Save.** Screenshot it, note the username and the date, and keep it, even if it's embarrassing. **Tell.** Tell a parent, or another trusted adult if a parent isn't reachable. Fast is good.

Then say, out loud, more than once: **"You are not in trouble for telling. You will not lose your device for telling."** A student who fears losing the phone will hide the problem.

A calm word for parents: if someone asks for pictures, or threatens your child

One online danger is worth naming plainly for 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 a 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 FBI says the money-driven version mainly targets teenage boys and tends to start where young people already spend time: social apps and gaming sites.⁷⁴

Before anything happens, say this once, calmly, as part of your device talk: "If anyone ever asks you for pictures, or threatens you: Stop, Save, Tell. You are not in trouble. Don't pay. Come to us."

If it happens: - Stay calm. Your child is the victim of a crime, not the cause of it. Say so. - Don't pay and don't send anything more. The FBI warns that going along with the demands rarely stops them.⁷⁴ - Keep everything: messages, usernames, profiles. - Block the account and report it through the app's safety tools. - Report it to the FBI at tips.fbi.gov or 1-800-CALL-FBI, or to your local police.⁷⁴ - If an image has been shared or might be, the free **Take It Down** service from the National Center for Missing & Exploited Children can help remove images taken before 18 from participating platforms, without anyone sending or viewing the image.⁷⁵

That's the whole box. No more detail is needed at the table.

Wrong answers you should be able to hear

“Private mode means nobody can see.” *What it usually means:* your student thinks a browser setting hides them from everyone. *What you say:* “Private from whom? Let’s list who could still see it: the website, the network, anyone looking over your shoulder.” Private browsing mostly stops your own device keeping a history.

“The lock icon means the site is safe.” *What it usually means:* HTTPS confusion, a favorite from Chapter 6. *What you say:* “It means the connection is private. Is the site honest? How would we check?” A scam site can have a lock too. NET-05 Look for the Lock, from Chapter 6, makes a good ten-minute revisit: your student finishes the sentence “The lock means... It doesn’t mean...”

“My password is strong. It has an exclamation point.” *What it usually means:* the old advice about symbols stuck. *What you say:* “Length beats tricks. How many characters is it? Could you make a longer phrase you’d still remember?”

“I’d never fall for that.” *What it usually means:* your student has only seen clumsy fakes. *What you say:* “Good. Then let’s see if you can catch this one.” Run SAFE-03 with a well-made fake.

Five-minute parent warm-up

1. Write three short sample messages: one obvious phish, one well-made phish, one ordinary message. Invent the company names. That’s tomorrow’s SAFE-03.
2. Decide where the family password list lives and who holds which passwords. A password manager or a sealed envelope in a drawer you control both work.
3. Open the privacy settings on one device you own, just to know where they are.
4. Say the talk-box opening aloud once: “Who can see this, who could fake it, and who do you tell?” Then count to five in your head.

How to teach it this week

The Computing Hour shape holds: Unplug and Plan, Predict, Build and Test, Debug and Log, Save and Explain. This week most of the work is on paper. The screen comes in briefly, with you beside it.

A worked Hour (illustration)

Here is an illustration, not a real family.

Unplug and Plan (5–10 min). Lay three printed messages face down. “Today you’re the security team. Before you look, write three warning signs a scam message might have.” Your student writes the list in the Debug Log.

Predict (2–5 min). Turn the messages over. “For each one: phish or fine? Write your prediction before we discuss.”

Build and Test (15–20 min). Your student checks each message against the list, adds new signs, and sorts. Then, on your device, hover over the link in your parent-made sample to see where it really goes (SAFE-04).

Debug and Log (5–10 min). A wrong prediction is the best part. “What did you expect, what was it actually, and what fooled you?” One Expected/Actual line goes in the Debug Log.

Save and Explain (3–5 min). File the sorted pile and the list. Your student says, “I predicted... It did... Next I’d change...” and “Help I used: ...” (usually “none” this week).

Exact wording you can say

- To open: “You’re the security team today. Your job is to catch the fakes.”
- When they’re unsure: “What is this message asking you to do? And how fast?”
- To fade help: “What would you check before tapping anything?”
- To close: “Which warning sign was the strongest clue today?”

When to stop talking

When your student is reading a message and deciding, say nothing. Look at the paper, not at them. Noticing the hurry and the ask on their own is the skill. If they ask you, ask back: “What’s it asking you to do?”

In-home — SAFE-01 Passphrase Builder

- **Time:** 15 minutes.
- **Materials:** paper, a pencil, and any book.
- **Safety and privacy:** practice phrases only; never write down a real passphrase.
- **The fun:** ridiculous, memorable word chains (“violin-cactus-harbor-pancake”).
- **The skill:** length beats tricks; one passphrase per account; password managers in parent hands.
- **The crumb:** a rules card in your student’s words, such as “Four or more random words, at least 15 characters, never reused, never shared outside the family.” The rule, never the phrase.

Open the book to a random page and pick a word; repeat four times; count the characters. Compare with “Fluffy2!” and ask which someone who knows the family dog’s name could guess.

11–12: The parent holds every real password; the student builds practice phrases and explains why length matters.

13–14: The student may create one real passphrase for a family-approved tool, typed straight into the password manager with you present.

In-home — SAFE-03 Phish or Fine?

- **Time:** 20 minutes.

- **Materials:** six to ten sample messages you've made up: a fake prize, a fake account warning, a "friend" asking for a login code, an ordinary reminder, a delivery notice, a normal note from a relative.
- **Safety and privacy:** paper only; no real brands, people, or links.
- **The fun:** catching a well-made fake before a parent does.
- **The skill:** spotting hurry, threats, prizes, odd senders, vague greetings, and requests for codes or money.
- **The crumb:** the sorted pile and the reasons.

11–12: Six messages, three obvious; sort into "phish," "fine," and "ask a parent."

13–14: Ten messages, some very good. Your student then writes a fake of their own, hiding three warning signs, for you to catch. You may be worse at this than you expect, which is good for everyone.

In-home — SAFE-08 Family Device Agreement

- **Time:** 20 minutes to draft; a second sitting to sign.
- **Materials:** paper, and everyone who uses devices in the house.
- **Safety and privacy:** first names only if it hangs where visitors can see it.
- **The fun:** your student drafts it, and the adults sign it too.
- **The skill:** shared written rules that are about safety and fairness, not punishment.
- **The crumb:** the signed agreement, dated, with a review date.

What usually goes on it (in your family's words): where devices live (a shared room; phones out of bedrooms overnight; screens off an hour before bed); who holds which passwords; which apps and accounts need a parent's yes; two-step login on where offered; Stop, Save, Tell; first names only, with no faces, school, or town in anything shared; the AI rules from the AI box in **The Computing Hour**; and what happens if a rule is broken. Add one line everyone signs: "Telling a parent about a problem online never costs you your device."

11–12: The parent writes the main rules; the student proposes two and words the Stop, Save, Tell line.

13–14: The student writes the first draft and the parent edits. A teen who wrote the rule is likelier to keep it. Review it each semester.

In-home — SAFE-09 Stop, Save, Tell

- **Time:** 15 minutes.
- **Materials:** a few paper "messages" you write: mean ("nobody likes you"), pushy ("send me your login code"), nosy ("where do you live?"), threatening ("pay me or I'll post this"). Keep them mild and invented.
- **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:** a script card in your student's words.

Hand over a paper message. Your student acts the three steps: hands off, pretend screenshot, comes to you. You say, “You are not in trouble.” Then swap. Ordinary is the goal.

13–14: Add “Who else could you tell if I’m not here?” and write two names on the card.

Band moves

11–12: No social media accounts; the platforms’ own minimum is 13. The parent holds every password and approves every app. The work is paper-first: sorting messages, practice passphrases, rehearsing Stop, Save, Tell, and helping draft the agreement. Settings tours happen on the parent’s device, with the student reading the options aloud.

13–14: The student may hold passwords for family-approved tools, with the parent’s sealed copy. Two-step login goes on together (SAFE-02), a Footprint Audit happens with a parent (SAFE-05), and settings get reviewed each semester. Any first account happens under the service’s terms, with you at the table.

Ready 12: A twelve-year-old who handles SAFE-03 easily can write fake messages for you to catch. Accounts still wait for 13.

Practice that actually builds learning

Blocked practice: three days of Phish or Fine? with new samples, until your student names the warning signs quickly. **Mixed practice** once it’s known: shuffle phish samples, settings questions, and Stop, Save, Tell messages into one pile, so each card asks “which habit is this?” before “what do I do?” Once a week, at the start of any Hour, ask for the four habits from memory. It takes thirty seconds.

Talk box (this chapter)

Opening (exact): “Who can see this, who could fake it, and who do you tell?”

Follow-ups (pick three to five):

1. *Prediction:* “If you post this, who do you think will see it in a week? In a year?”
2. *Evidence:* “What in this message makes you think it’s real, or fake?”
3. *Location:* “Point to the part that’s asking you to do something.”
4. *Change:* “What would you change before sharing it: crop it, remove the name, ask first?”
5. *Revision:* “You can change your answer. Let’s look again.”

How to wait. After you ask, count to five in your head. That pause is a habit drawn from classroom wait-time studies.⁵ When the question is about code on the screen, let them read as long as they need, and the same goes for a message or a settings screen. Look at the screen, not at your student.

What a stuck silence usually means. The question was too big (ask one part: “Who could see this?”). There was no wait (ask again and count). Your student is worried about something already

done (say, “Nobody’s in trouble. We’re just looking.”). Or they’re waiting for you to answer, because you have before. Wait longer.

Don’t say / say instead. Don’t say, “How could you fall for that?” Say, “That one was well made. Which sign would catch it next time?”

In-home — SAFE-02 Two-Step Login Walk-Through

For 13–14.

- **Time:** 15 minutes.
- **Materials:** one family-approved account, the device that will receive codes, and you beside them.
- **Safety and privacy:** no photos or notes of any codes; backup codes go into the password manager or the sealed envelope.
- **The fun:** watching a stolen password become useless.
- **The skill:** turning on two-step login, and knowing what to do if an unexpected code arrives (tell a parent and change the passphrase).
- **The crumb:** a dated note that two-step login is on for that account. No codes.

In-home — SAFE-04 Hover Before You Click

- **Time:** 10 minutes.
- **Materials:** a sample email or document you made on your own device, with a link whose words say one thing and whose address says another (point it somewhere harmless, like a library page).
- **Safety and privacy:** you make the sample and hold the device.
- **The fun:** catching the mismatch.
- **The skill:** checking where a link really goes before tapping, and finding the real site name in an address.
- **The crumb:** a cropped screenshot of the sample link.

In-home — SAFE-05 Footprint Audit

For 13–14.

- **Time:** 20 minutes.
- **Materials:** the parent’s device, a search engine, and a private notebook.
- **Safety and privacy:** the parent drives. If something upsetting turns up, stop and talk privately. The list stays private; it isn’t a portfolio piece.
- **The fun:** detective work on yourself.
- **The skill:** seeing what’s public and deciding what to tighten. Ask the talk-box opening for each result.
- **The crumb:** a private list of what was found and one change made.

In-home — SAFE-06 Privacy Settings Tour

- **Time:** 20 minutes.
- **Materials:** one device or app the student uses.
- **Safety and privacy:** the parent holds the device at 11–12; at 13–14 the student drives with the parent beside them.
- **The fun:** finding settings neither of you knew existed.
- **The skill:** who can see me, who can contact me, and what’s shared (location, contacts, microphone, camera).
- **The crumb:** a dated checklist. Repeat each semester and on every new app.

In-home — SAFE-07 Age Rules Map

- **Time:** 15 minutes.
- **Materials:** a three-column paper table (service, minimum age, date checked) and the parent’s device.
- **Safety and privacy:** the parent looks up the terms; nobody signs up for anything.
- **The fun:** spotting the pattern.
- **The skill:** noticing that 13 shows up again and again, and explaining why (the 13 shift).
- **The crumb:** the table with dates. Terms change, so check the current terms before relying on any row. Chapter 8 adds AI services to this map.

Out-of-home — OUT-05 Scam Spotting Out and About

- **Time:** 15 minutes, folded into an errand.
- **Materials:** a notebook; a store, parking lot, post office, or library.
- **Safety and privacy:** stay together; follow road rules in parking lots; no photos of strangers or staff; don’t scan any QR code you find.
- **The fun:** finding warnings that are already everywhere.
- **The skill:** noticing gift-card scam warnings near the checkout, QR stickers on parking meters (sometimes stuck over the real one), “urgent” signs, and posted rules about who to call.
- **The crumb:** three examples, each with its warning sign. Ask the talk-box opening about each one.

One incorrect example to diagnose (illustration)

Here is an illustration, not a real family. A thirteen-year-old films a short how-to video about his model rockets. He’s in the backyard, he says “Hi, I’m Sam from Riverside Middle,” and the house number shows behind him. He asks to post it.

What went wrong: together, his name, school, and house tell a stranger where to find him.

What to do, calmly: praise the rocket video first; it’s a real project. Then run the talk-box opening together and let him find the school name and house number himself. Reshoot with no school, no house, and a first name only, or keep it in the family. He made the safety call himself.

For the student

Here's the honest truth: most online problems don't come from movie-style hackers. They come from ordinary tricks that work on ordinary people, adults included. A reused password. A message that says "URGENT." A photo that shows more than you meant. A few habits stop most of it, and you can learn them this week.

The four habits.

1. **Spot the phish.** A scam message almost always wants two things: for you to hurry, and for you to give something (a password, a code, money, your address). Hurry plus a request is your signal to slow down and show a parent.
2. **Long passphrases, never reused.** Four or five random words beat a short password full of symbols. "harbor-violin-cactus-pancake" is stronger than "Fluffy2!" and easier to remember. Every account gets its own.
3. **Two-step login.** A second step, usually a code, means a stolen password isn't enough. A login code is only ever typed into a page *you* opened. If someone asks you to read a code out to them, that's the person the code is protecting you from.
4. **Updates on.** Updates patch holes criminals already know about.

A worked example. Here's a message:

"Hey! It's Jordan from the team. I got locked out, can you send me the 6-digit code that just came to your phone? Need it fast!!"

Check it against the habits. It hurries you. It asks for a login code. It says it's a friend, but you can't see who actually sent it. That code came to you because someone typed *your* username; whoever wrote this is trying to get into your account. The right move: don't reply, take a screenshot, show a parent. If it really was Jordan, a phone call sorts it out in thirty seconds.

Your password is yours, and your parent's. Not your best friend's. At 11 or 12, your parent holds your passwords. That's not about trust; it's how accounts work for under-13s.

If you're 13 or older: you may hold passwords for tools your family has approved, with your parent keeping a sealed copy. Turn on two-step login together, and check your privacy settings each semester. Thirteen is when companies' rules change; your family's rules change at the speed your family decides.

If you clicked something you shouldn't have, tell a parent right away. You're not in trouble. A password changed in the first ten minutes saves a lot of trouble later.

Stop, Save, Tell. If anything online is mean, scary, pushy, or threatening: stop replying, save it (screenshot, don't delete), and tell a parent. If anyone ever asks you for pictures or threatens you, the same three words apply, plus one more: **don't pay.** You are not in trouble. You will not lose your device for telling. Your parents would much rather hear about it than not.

Kindness when nobody's watching. Before you send something, read it in a flat voice, as if you were the person getting it. In a group chat you don't have to fix everything, but you do have to not join in when it turns cruel. Leaving the chat and telling someone is a strong move.

This part is hard for everyone. Good fakes fool grown-ups too. Getting fooled once doesn't mean you're careless; telling someone fast is what keeps you safe.

Try it today. Ask your parent for three sample messages. For each, write “phish” or “fine” and one reason *before* you discuss them. Then check.

Explain it back. Tell someone in your family why a long passphrase beats a short password with symbols. Use the word “guess.”

Challenge. Write a fake scam message on paper, from a made-up company, with three warning signs hidden in it. See if your parent finds all three.

If you're 13 or older, a coding challenge. This function checks whether a practice phrase is long enough. Predict the three lines it prints before you run it.

```
def long_enough(phrase):
    return len(phrase) >= 15

tests = ["sunflower", "purple-otter-jumps-tuesday", ""]
for t in tests:
    print(t, long_enough(t))
```

It prints `sunflower False`, then `purple-otter-jumps-tuesday True`, then a line that starts with a blank (the empty phrase) followed by `False`. Add a test of your own, using a made-up phrase, never a real one.

What to save. Your passphrase rules card (never a real passphrase), your sorted Phish or Fine? pile with reasons, your Stop, Save, Tell card, and your copy of the signed agreement.

If it isn't clicking

“That would never happen to me.”

What it usually means: your student has only seen clumsy fakes.

Next move: make a very good fake for SAFE-03, built around an invented club or hobby they care about. Let them get it wrong once at the table, where it's free, then talk about what fooled them.

Eye-rolling, or “You don't trust me.”

What it usually means: the rules feel like a verdict on their character.

Next move: hand over the pen. Let your student draft the agreement and explain each rule's reason, and make clear the adults sign too. Say plainly that the 13 shift and the password rules are about how accounts work, not about trust. If “all my friends have it” comes up, keep it short and kind: “Other families decide for themselves. In our house, the agreement is the rule, and we'll look at it again next semester.” Then do.

Fear: “I don’t want to go online at all now.”

What it usually means: something landed too heavily.

Next move: come back to the four habits and to Stop, Save, Tell. “You have a plan, and I’m right here.” Spend a few days on the fun parts, like writing fakes for you to catch. If a fear seems tied to something real they saw, talk privately, outside the lesson.

When to slow down, go ahead, or step back

Slow down if Stop, Save, Tell isn’t automatic or the four habits don’t come from memory. **Go ahead** when your student sorts a mixed stack confidently and explains why. **Step back to paper** whenever a screen activity turns into scrolling.

The rescue order holds: ask → wait → hint → short demo on your own copy → hand it back. Here, “your own copy” means your own sample message or your own settings screen.

Tools, including AI

- **A password manager** in the parent’s hands at 11–12 and shared at 13–14, or a sealed envelope in a drawer you control. What matters is one strong, unique passphrase per account.
- **Parental controls** on the router, phones, tablets, game consoles, and accounts. The AAP lists time, downloads, contacts, and purchases as settings caregivers can manage.⁶
- **Two-step login** on every account that offers it, starting at 13–14 with SAFE-02.

AI in this chapter. An AI-written list of safety rules doesn’t replace your family agreement; your student’s own words do the work. Private messages, chat screenshots, photos, and names stay out of chatbots, even when you’re asking for advice about a problem. For everything else, see the AI box in **The Computing Hour**, and check the current terms of any service before you use it.

What “done enough” looks like

Move on when most of these are true. Skill, not birthday.

- ☐ Your student names the **four habits** from memory, with a reason for each.
- ☐ They spot **three phishing signs** in a new sample without prompting.
- ☐ They state a **strong passphrase rule** in their own words, without revealing a real passphrase.
- ☐ They say and act out **Stop, Save, Tell**, including “don’t pay,” and can name who they’d tell.
- ☐ They explain **the 13 shift** in a sentence: a rule for companies, not a readiness test.
- ☐ The **family device agreement** is drafted with their help, signed by everyone, and dated with a review date.
- ☐ **13–14:** two-step login is on for at least one account, and they’ve done a Footprint Audit with a parent.
- ☐ They answer the talk-box opening about one real photo or message.

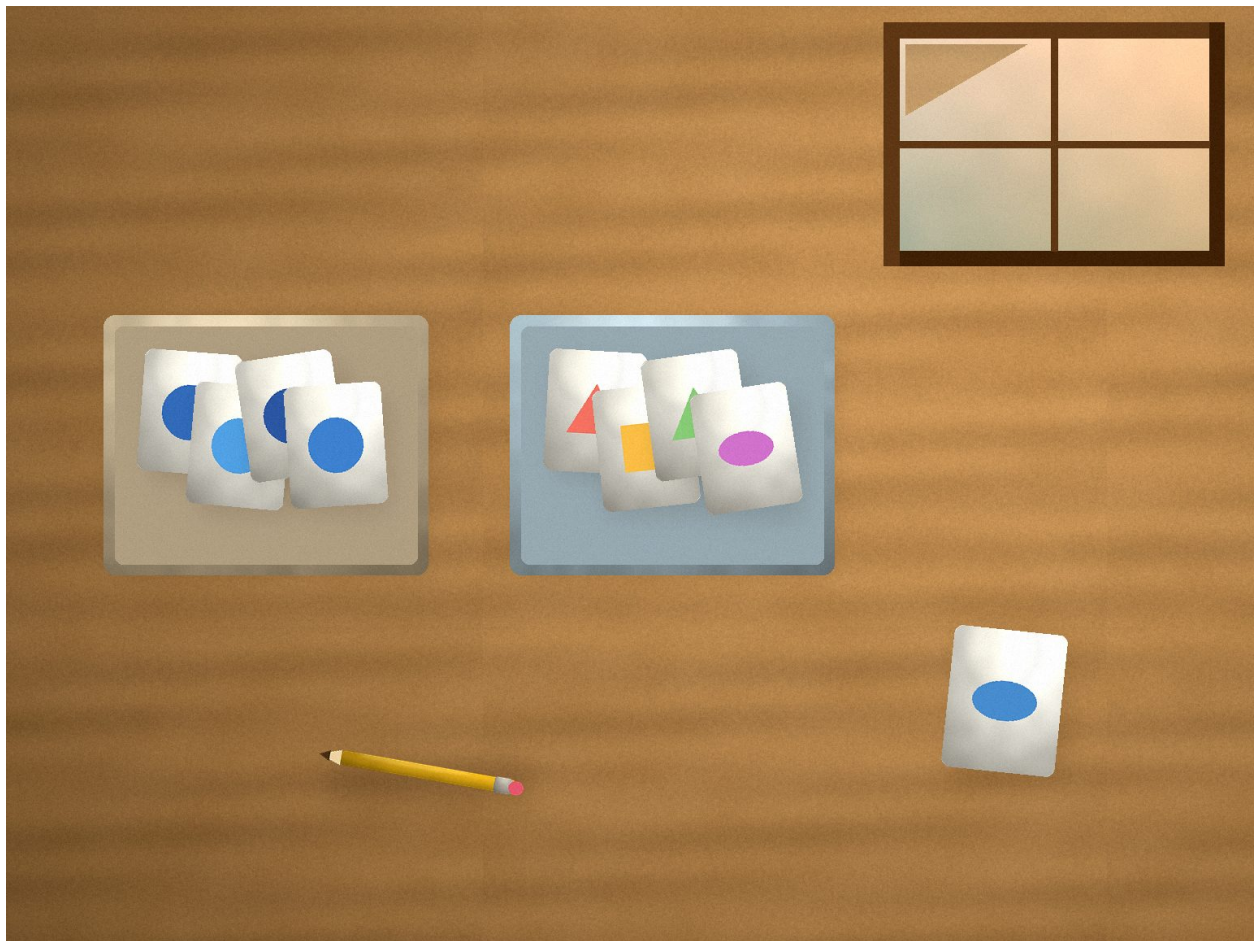
A short check for the file (Chapter 9 gathers these): **AS-03 Explain Your Code**, safety version. Your student explains why each rule in the agreement is there; at 13–14, they also explain the `long_enough` function line by line. Write down one sentence they say. Add any Expected/Actual line from a wrong Phish or Fine? prediction to the Debug Log: “Expected: fine. Actual: phish. Clue I missed: the code request.”

Keeping it alive. Re-run SAFE-03 with fresh samples once a semester, and review the agreement whenever a new device arrives.

Nobody can make the internet perfectly safe. You’ve given your student four habits that stop most trouble, a routine for the rest, and a clear promise that telling is always safe. That’s a great deal.

Chapter 8

AI Up Close: Training Data, Bias, Prediction, and Whose Work It Is



Why this matters

Here is an illustration, not a real family. A thirteen-year-old asks a chatbot, with his mother beside him, for three books about the history of rockets. It answers at once: three titles, three authors, three

tidy summaries. His mother suggests they look the books up at the library website. Two are real. The third doesn't exist. The title sounds right, the author's name sounds right, and the summary is lovely. It's just made up.

He's annoyed, then fascinated. "Why would it do that?"

That question is the door into this chapter. By the end, your student will be able to answer it in plain words: a chatbot is built to predict what words are likely to come next, and a likely-sounding answer isn't always a true one. They'll also know where an AI's "knowledge" comes from, why it can be lopsided, and why anything it produces is made by a tool, not by them.

Your student is growing up with AI in the search box, the photo app, the store, and the bank. Most teens already use chatbots: in a 2025 national survey, 64 percent of U.S. teens 13 to 17 said they had used one.⁷³ Whether your family uses AI a lot or hardly at all, understanding it is now part of computing, the way understanding how a network works is.

CSTA, the Computer Science Teachers Association, puts it in grown-up terms. Middle-schoolers should be able to guess how a machine learning model makes its predictions, investigate how better examples in the training data improve accuracy and reduce bias, and evaluate "when it is appropriate to use AI and other emerging technologies."¹⁷ Its guidance for schools adds a sentence worth keeping: using an AI tool "only becomes CS learning when the lesson addresses how the tool works."¹⁵ That's this chapter. Almost all of it happens on paper, with no chatbot at all.

This week you can learn how AI learns from examples and why that can go wrong. **Today your student can try** guessing a hidden rule from a stack of example cards, exactly the way a sorting model does.

For the parent: understand it yourself

You don't need to know any math to teach this chapter. You need four ideas: AI learns from examples; the examples can be lopsided; chatbots predict likely words; and whose work it is depends on who made it and who can explain it.

The everyday picture

Picture teaching a new helper at a farm stand to sort apples from pears without ever telling them a rule. You just hand over fruit, one at a time, saying "apple" or "pear." After a few dozen, they start getting it right. They've learned a pattern from examples. If you only ever showed them red apples, though, the first green apple might get called a pear. They learned exactly what you showed them, gaps and all.

That's most of what "machine learning" means.

The precise picture

Artificial intelligence (AI), as most people use the word today, usually means programs that learn patterns from examples instead of following rules a person typed one by one. The examples are the

training data. What the program builds from them is a **model**: a big set of learned numbers that turn an input into a guess.

A **classifier** is a model that sorts things into groups: spam or not spam, cat or dog, fish or trash. You train it on labeled examples, then **test** it on examples it hasn't seen. The share it gets right is its **accuracy**. The test examples have to be new; testing on the training examples is like grading a quiz with the answer key the student studied.

Bias, in this chapter, means a model that is wrong more often for some groups or situations than others. It often comes from **lopsided training data**: too few examples of some things, or examples labeled by people with their own blind spots. CSTA asks middle-schoolers to analyze how choices made at each stage of working with data “can lead to biased data, misleading conclusions, and compromised AI models.”¹⁷ The key sentence for your student is short: **people choose the data**. A model isn't neutral because a computer runs it.

Next-word prediction: how chatbots talk

A **chatbot** is built on a language model trained on an enormous amount of text. Underneath, its main job is to predict a likely next word (really, a likely next piece of a word), then the next, then the next. Done at huge scale, with careful training afterward, this produces answers that are fluent, organized, and often useful.

But notice what the job is. It's *likely*, not *true*. A chatbot that has seen thousands of book titles and author names can produce a new title and author that look exactly like real ones. That's how the rocket book appeared. **Fluent isn't the same as true**. A chatbot can make up facts, quotes, and sources, and it says them in the same confident voice as everything else. So the family rule is simple: check one claim against a real source before trusting it, and open any source it cites yourself.

Chips and data centers

Training and running these models takes a great deal of computing. It happens on chips that do huge numbers of small sums at the same time, in big data centers that use a lot of electricity. The International Energy Agency estimates that data centers used about 1.5 percent of the world's electricity in 2024, and that a single AI-focused data center can draw as much electricity as a power-hungry factory.⁵⁹ Your student doesn't need the figures. They need the idea that every AI answer runs on real machines somewhere, using real energy, and that this is one of the things to weigh when deciding whether AI is the right tool.

When AI is a good tool, and when it isn't

CSTA's standard asks students to weigh AI by its capabilities, its limits, and its environmental impact.¹⁷ In family terms, AI tends to help with explaining an idea a second way, giving practice questions, or suggesting where a bug might be *after* your student has looked. It tends to hurt when it does the thinking the lesson exists to build, when the answer has to be true and nobody checks, or when the question involves private information.

The College Board’s guidance for AP students warns that AI tools can produce “incomplete code, code that creates or introduces biases, code with errors, inefficiencies,” or code too complicated to understand and explain.⁷⁶ That’s good advice for a thirteen-year-old too.

Whose work is it?

Here is the rule this book uses, adapted from the AP Computer Science Principles course. The College Board allows AI as a “supplementary resource” for understanding, building, and debugging code, but “it is the student’s responsibility to review and understand any code co-written with AI tools,” students must be “prepared to explain their code in detail,” and they should acknowledge “any code created by generative AI tools.”²³ In short: **understand it, explain it, acknowledge it.**

For this book, that becomes three habits:

1. Anything an AI or generator produces is made by a tool. It is never your student’s own work and never a portfolio piece on its own.
2. Every project carries a Whose Work Is It? card (AI-06): who or what helped, what was asked, what came back, what was kept, what was changed, and how it was tested.
3. Every Save and Explain ends with “Help I used: ...”, even when the answer is “none.”

The same goes for code borrowed from a book, a friend, or a website. Remixing is a fine way to learn, with credit. CSTA asks students to “justify the importance of attribution and intellectual property.”¹⁷ Credit isn’t a confession. It’s what honest builders do.

One more thing: AI “detectors” aren’t used to judge a student in this book. You have something better. “Explain it to me” is the check.

The age split, in one line

Most consumer AI services set 13 as their minimum age, and at least one major one sets 18.^{9,10} So at 11–12 any AI use is parent-operated, on your account, with you reading every exchange; at 13–14 it’s supervised use under the service’s terms, with you nearby. Google offers a parent-enabled, supervised way for children under 13 to use its Gemini apps; this book names it so you know it exists. It isn’t a recommendation, and this book still keeps the parent-operated rule at 11–12. CodeAI (formerly Code.org) labels its AI Tutor for students 13 and up, so it stays off at 11–12.^{11,55,12} Check the current terms before you start. For the full rules, see the AI box in **The Computing Hour**, including its caution about AI helpers that share one computer.

Wrong answers you should be able to hear

“It’s basically a person. It knows things.” *What it usually means:* fluent language feels like a mind. *What you say:* “It predicts likely words. Let’s build a tiny one out of paper and see.” Then run AI-04.

“If it sounds sure, it’s right.” *What it usually means:* confidence is being read as evidence. *What you say:* “Fluent isn’t true. Pick one claim. How would we check it?” (AI-05)

“Computers can’t be biased. It’s just math.” *What it usually means:* your student hasn’t seen where the data comes from. *What you say:* “Who chose the examples? Who’s missing from them?” (AI-03, DATA-06)

“More data always fixes it.” *What it usually means:* half right. *What you say:* “More of *what*? If we add a hundred more red apples, does the green apple get easier?”

“The AI wrote it, but I get it.” *What it usually means:* reading code feels like understanding it. *What you say:* “Great. Explain line 6 without looking at the chat.” (AI-07)

Five-minute parent warm-up

1. Make twelve index cards for AI-02. Draw or write simple objects that follow a hidden rule you choose, such as “things with wheels” or “things bigger than a shoebox.” Write the rule on a separate card and hide it.
2. Ask yourself one question you’ll ask your student: “What did it learn from?”
3. Find the Whose Work Is It? card template from the AI box in **The Computing Hour**, or write the six lines on an index card.
4. Say the talk-box opening aloud once: “What did it learn from, where might it be wrong — and whose work is this?” Then count to five in your head.

How to teach it this week

The Computing Hour shape holds: Unplug and Plan, Predict, Build and Test, Debug and Log, Save and Explain. Almost everything this week is unplugged.

A worked Hour (illustration)

Here is an illustration, not a real family.

Unplug and Plan (5–10 min). “Today you’re the AI. I’ll show you example cards marked yes or no. Your job is to figure out the rule.” Your student writes in the Debug Log: “Rule I’ll be looking for: ____.”

Predict (2–5 min). After eight training cards, before seeing the test cards: “Write your best guess at the rule.”

Build and Test (15–20 min). Show four new test cards, one at a time. Your student says yes or no for each, then you reveal the answers. Count the hits: that’s the accuracy.

Debug and Log (5–10 min). If a test card fooled them, ask, “What about the training cards led you there?” One Expected/Actual line in the Debug Log.

Save and Explain (3–5 min). “I predicted... It did... Next I’d change...” and “Help I used: ...”. Keep the rule guessed beside the real rule.

Exact wording you can say

- To open: “You’re the model today. Learn from the examples.”
- When they’re stuck: “What do all the yes cards have in common?”
- To connect it: “That’s what a sorting model does, just with thousands of examples.”
- To close: “What would have made the training cards better?”

When to stop talking

When your student is staring at the cards, hunting for the rule, say nothing. That hunt is the lesson. If they ask for a hint, point at two cards and ask, “What’s the same about these?”

In-home — AI-02 Human Classifier

- **Time:** 15 minutes.
- **Materials:** twelve to sixteen index cards you’ve made, with a hidden rule; a pencil.
- **Safety and privacy:** objects only. No people’s faces or names on the cards.
- **The fun:** being the AI, and catching the parent’s sneaky rule.
- **The skill:** learning a rule from labeled examples, then testing on new ones.
- **The crumb:** the rule guessed beside the real rule, and the accuracy (for example, “3 of 4”).

11–12: An easy rule with clear examples. Then let your student make a deck for you.

13–14: A trickier rule (two features at once). Ask, “Could two different rules both fit the training cards?”

In-home — AI-03 Lopsided Training Set

- **Time:** 15 minutes.
- **Materials:** the AI-02 cards, plus new ones; a tally sheet.
- **Safety and privacy:** objects only.
- **The fun:** fooling the “model” on purpose.
- **The skill:** how training data that leaves things out produces wrong guesses.
- **The crumb:** an error tally: which test cards went wrong, and why.

Train your student on a lopsided deck: every “fruit” card is red, say. Then test with a green apple, a yellow banana, and a red ball. Tally the errors. Then ask the key question: “How would you fix the training deck?”

11–12: Keep it to fruit, vehicles, or animals.

13–14: Connect it to real systems in one sentence: “If a model learns from photos that mostly show one kind of thing, it does worse on what it rarely saw.”

In-home — AI-04 Next-Word Guesser

- **Time:** 25 minutes.

- **Materials:** paper, a pencil, a short sentence or two (a nursery rhyme works), and at 13–14, a computer with Python.
- **Safety and privacy:** use a public rhyme or your own made-up sentences; no private messages or diaries.
- **The fun:** making nonsense that sounds almost right.
- **The skill:** how a text predictor chooses a likely next word from what it has seen.
- **The crumb:** the word-pair table and one generated sentence labeled “made by our table.”

Write the text out. Make a table: for each word, list every word that came right after it. “the” might be followed by “cat,” “mat,” “cat,” and “door.” Start with “the,” pick one of its followers at random (roll a die or draw slips), then look up *that* word’s followers, and keep going. Read the sentence aloud. It will sound like the text, and it may say something the text never said. That’s the point: fluent, not necessarily true.

11–12: Paper only. Ask, “Did our table *know* anything about cats?”

13–14: Build the same table in Python. Predict what kinds of sentences it can make before running it.

```
import random
text = "the cat sat on the mat and the cat ran to the door"
words = text.split()
word = "the"
sentence = [word]
for step in range(5):
    options = []
    for i in range(len(words) - 1):
        if words[i] == word:
            options.append(words[i + 1])
    if options == []:
        break
    word = random.choice(options)
    sentence.append(word)
print(" ".join(sentence))
```

The output changes each run. You might see the cat sat on the mat, the cat ran to the cat, or the mat and the door, which stops early because “door” has nothing after it. “The cat ran to the cat” is a fluent sentence the original text never said. Talk about that one.

In-home — AI-06 Whose Work Is It? Card

- **Time:** 10 minutes.
- **Materials:** an index card and the project your student just finished.
- **Safety and privacy:** first names only on the card.
- **The fun:** a project’s “credits,” like the end of a film.
- **The skill:** saying plainly what a tool did and what you did.
- **The crumb:** the card, filed with the project.

The six lines: who or what helped; what was asked; what came back; what was kept; what was

changed; how it was tested. A card that says “no help used” is a good card. Fill one out this week for any project from Chapters 3 to 6, even if nobody used AI. The habit matters more than the content.

Band moves

11–12: All AI is parent-operated: your account, your device, you reading every exchange. The chapter is mostly unplugged (AI-02, AI-03, AI-04 on paper, AI-06). A trained model (AI-01) and checking a chatbot (AI-05) happen on your screen, with your student directing and you typing. No student accounts on chatbots or AI tutors.

13–14: Supervised use under the service’s terms, with you nearby and the screen visible. Add the Python next-word guesser and AI-07 Explain Every Line. Your student fills out the Whose Work card without being reminded.

Ready 12: A twelve-year-old who explains the next-word table clearly can do the Python version with you at the keyboard. Accounts still follow the terms.

Practice that actually builds learning

Blocked practice: two or three rounds of Human Classifier with new decks until your student talks about “training cards” and “test cards” without prompting. **Mixed practice:** a shuffled set of questions where each needs a different idea: “Why did it get the green apple wrong?” (lopsided data), “Is this book real?” (fluent isn’t true), “Who wrote this function?” (whose work). Once a week, ask the talk-box opening about any AI your student met, at home or out.

Talk box (this chapter)

Opening (exact): “What did it learn from, where might it be wrong — and whose work is this?”

Follow-ups (pick three to five):

1. *Prediction:* “Before we ask it, what do you think it’ll say? What might it get wrong?”
2. *Evidence:* “How could we check that claim? Where’s a source we trust?”
3. *Location:* “Point to the line you didn’t write. Which part came from the tool?”
4. *Change:* “If we gave it different examples, what would change?”
5. *Revision:* “Want to change your answer about whose work this is?”

How to wait. After you ask, count to five in your head.⁵ When the question is about code on the screen, let them read as long as they need. Look at the screen, not at your student.

What a stuck silence usually means. The question was too big (ask one part: “What did it learn from?”). There was no wait (ask again and count). Your student is worried that using a tool was wrong (say, “Using help is fine. We’re just writing down what helped.”). Or they honestly can’t explain a line, which is useful information, not a failure. Say, “Then it isn’t yours yet. Let’s make it yours.”

Don't say / say instead. Don't say, "Did you cheat?" Say, "Walk me through line 4. What does it do?"

In-home — AI-01 Train a Sorting Model (Objects Only)

- **Time:** 25 minutes.
- **Materials:** your device and a free classroom tool for training a simple model. CodeAI's AI for Oceans, part of its Hour of AI and listed for grades 3 to 12, has students train a model to tell sea creatures from trash.⁷⁷
- **Safety and privacy:** objects only, never photos of people. At 11–12, the parent runs it on the parent's device, with no student account.
- **The fun:** training a model, then watching it make funny mistakes.
- **The skill:** training and testing a classifier, and seeing how the examples shape the result.
- **The crumb:** an accuracy note: what it got right, what it got wrong, and why you think so.

11–12: Your student decides each label; you click.

13–14: Ask your student to try to make the model worse on purpose with a lopsided set, then fix it.

In-home — AI-05 Check the Chatbot

- **Time:** 20 minutes.
- **Materials:** the parent's device, a chatbot your family uses under its terms, and a library website or encyclopedia.
- **Safety and privacy:** no names, photos, or personal details typed in. At 11–12, the parent types and the student dictates.
- **The fun:** fact-checking a machine.
- **The skill:** checking three claims against sources; noticing confident errors.
- **The crumb:** a three-row claim/check table: claim, where we checked, true or not.

Ask the chatbot a question with checkable facts, such as dates or book titles on a topic your student likes. Pick three claims. Check each one.

In-home — AI-07 Explain Every Line

For 13–14; ready 12 with the parent at the keyboard.

- **Time:** 20 minutes.
- **Materials:** a short piece of AI-suggested code (under 15 lines), on paper or screen; a pencil.
- **Safety and privacy:** the suggestion comes through the parent's or a supervised account under the terms.
- **The fun:** finding the line that's wrong before running it.
- **The skill:** explaining AI-suggested code line by line before using any of it.
- **The crumb:** the annotated code, with a note on each line and a "Help I used: ..." line.

Your student writes, beside each line, what it does in plain words. Any line they can't explain gets a question mark, and nothing with a question mark goes into the project until it's understood or rewritten.

In-home — AI-08 When Not to Use AI

- **Time:** 15 minutes.
- **Materials:** a dozen scenario cards you write; three piles labeled “fine,” “not fine,” and “ask first.”
- **Safety and privacy:** invented scenarios only.
- **The fun:** arguing about the tricky ones.
- **The skill:** weighing capabilities, limits, privacy, and energy before using AI.
- **The crumb:** the sorted cards, with a reason on each tricky one.

Sample cards: “Ask it to explain an error message after you’ve guessed.” “Have it write your whole project.” “Paste in a group chat to ask whether a friend is mad at you.” “Use it during a Cold Trace.” “Ask for five practice questions on loops.” “Ask it the weather when a window is right there.” There’s no perfect sort. The reasons are the point.

In-home — DATA-06 Who Is Missing?

- **Time:** 15 minutes.
- **Materials:** a small data set from Chapter 4, or a made-up one, such as a survey of “favorite snacks” asked only of people at a basketball game.
- **Safety and privacy:** invented or family data with first names only.
- **The fun:** spotting the gap.
- **The skill:** biased samples: who the data leaves out.
- **The crumb:** a two-sentence note: who’s missing, and how that changes the answer.

In-home — SAFE-07 Age Rules Map

- **Time:** 15 minutes.
- **Materials:** the age-rules table from Chapter 7 and the parent’s device.
- **Safety and privacy:** the parent looks up the terms; nobody signs up.
- **The fun:** finding the one that says 18.
- **The skill:** reading AI services’ minimum ages and seeing why 13 keeps appearing (the 13 shift).
- **The crumb:** the table with new AI rows and dates checked. Check the current terms; they change.

Out-of-home — OUT-06 Where Is AI Here?

- **Time:** 20 minutes, folded into an errand.
- **Materials:** a notebook; a store, bank, or library.

- **Safety and privacy:** stay together; no photos of staff or customers; ask questions politely and only if staff aren't busy.
- **The fun:** spotting the hidden machines.
- **The skill:** noticing automated decisions (self-checkout, recommended books, fraud alerts, chat help on a sign) and asking about them.
- **The crumb:** a list of what you spotted, with one question asked and the answer.

A good question for staff: “Does a computer decide that, or a person?” Afterward, ask the talk-box opening about one item.

One incorrect example to diagnose (illustration)

Here is an illustration, not a real family. A thirteen-year-old needs the average of three quiz scores for a small project. She asks a chatbot, under supervision, and pastes in its answer:

```
scores = [8, 9, 10]
total = 0
for s in scores:
    total = total + s
average = total / 2
print(average)
```

She expected 9.0. It prints 13.5. She had looked the code over and thought she understood it.

What went wrong: line 5 divides by 2 instead of by the number of scores. A careful line-by-line explanation (AI-07) would have caught it: “Line 5 divides the total by... wait, why 2?” The fix is `average = total / len(scores)`, which prints 9.0. There's also no “Help I used: ...” line and no Whose Work card.

What to do, calmly: praise her for testing it and noticing the wrong number; that's real debugging. Then ask her to explain each line aloud, and let her find line 5. She writes the Expected/Actual line in the Debug Log, fixes it herself, and fills out the card: “Help I used: a chatbot suggested the code; I found and fixed the divide-by-2 bug.” That's honest, and now the fix is hers.

For the student

AI is everywhere you look right now, so let's look at it closely. Not with fear, and not with hype. Just: how does it actually work?

AI learns from examples. Most of what people call AI today is a program that learns patterns from a huge pile of examples, called **training data**. Show a program thousands of photos labeled “cat” or “dog,” and it builds a **model**, a big set of numbers, that can guess the label for a new photo. Nobody typed in a rule like “cats have pointy ears.” The program found its own patterns. You'll do the same thing this week with index cards, playing the AI yourself.

The examples decide what it learns. If every apple in the training data is red, a green apple confuses it. That's not the computer being silly; it learned exactly what it was shown. When a model is wrong more often for some groups than others, that's called **bias**, and it usually starts with

the data. And here's the important part: **people choose the data.** So "it's a computer, so it's fair" isn't true. A model is only as fair as its examples.

Chatbots predict likely words. A chatbot is trained on an enormous amount of text. Its main job is to predict a likely next word, then the next, then the next. Done at giant scale, that can produce answers that are clear and helpful. But "likely" isn't "true." A chatbot can invent a book, a quote, or a fact that sounds perfect and doesn't exist. So:

- If it sounds sure, check anyway.
- If it gives a source, open the source.
- If it matters, check one claim against something you trust.

A worked example. Here's a tiny "language model" you can make in two minutes. Take the sentence "the cat sat on the mat and the cat ran to the door." For each word, write down what came after it. After "the" came: cat, mat, cat, door. After "cat" came: sat, ran. Now start with "the," pick a follower at random, then pick a follower of *that* word, and keep going. You might get "the cat ran to the cat." That sentence sounds fine, and it's nonsense, and it was never in the original. Real chatbots are vastly bigger and much better trained, but the idea of predicting what's likely is the same.

AI runs on real machines. Every AI answer runs on chips in big data centers that use a lot of electricity. That's one reason not to use AI for things you could easily do yourself, like asking the time.

Whose work is it? This is the rule that matters most for you: **anything an AI produces is made by a tool. It isn't your work.** Your work is what you planned, wrote, tested, and can explain. If you can't explain a line, it isn't yours yet. That's the same rule the AP Computer Science Principles course uses, in three words: understand, explain, acknowledge.

So every project gets a Whose Work Is It? card, and every Save and Explain ends with "Help I used: ...". Writing "Help I used: none" is great. Writing "Help I used: a chatbot explained what a KeyError means" is great too. The only bad card is a dishonest one.

If you're 13 or older: you may use AI services whose terms allow your age, with a parent nearby and the screen where they can see it. When AI suggests code, explain every line in writing before you use any of it. The checks in Chapter 9 (Cold Trace, Debug on the Spot, Build From a Blank) are always AI-free, so the skills have to live in your head, not in the chat.

If you're 11 or 12: AI happens on your parent's account, with your parent. You can still direct it: you decide the question, you read the answer, you check it. That's most of the skill anyway.

This part is hard for everyone. Fluent text feels smart, and it's easy to trust a confident answer. Adults fall for it too. The habit that protects you is small: check one thing.

Try it today. Ask your parent to make a deck of example cards with a secret rule. Learn the rule from eight cards, then test yourself on four new ones. Write down your accuracy.

Explain it back. Tell someone in your family why a chatbot can invent a book that doesn't exist. Use the words "likely" and "true."

Challenge. Make a lopsided deck for your parent: train them so they'll get a test card wrong. Then explain how you'd fix your deck.

If you're 13 or older, build the next-word guesser in Python (it's in this chapter). Feed it a longer text, such as a nursery rhyme, and see what it makes.

What to save. Your rule-guess sheet, your error tally, your word-pair table with one sentence labeled "made by our table," and a Whose Work card for this week's project.

If it isn't clicking

"It's basically a person."

What it usually means: fluent language feels like a mind. That's a very human reaction.

Next move: build the next-word guesser by hand (AI-04) and read its sentences aloud. Then ask, "Did our table know anything? Did it mean what it said?" Bigger models are vastly more capable, but the paper version shows that sounding sensible and knowing things aren't the same.

"I'm never using AI. It's cheating."

What it usually means: your student has heard strong opinions and wants a clear rule.

Next move: AI-08 When Not to Use AI. Sorting real scenarios shows there are fine uses, bad uses, and "ask first" uses. Using AI isn't the problem; hiding it and not understanding it are. A student who chooses not to use AI is welcome to; the Whose Work card still says "none."

"Everyone uses it for homework."

What it usually means: your student is testing whether the rule is real.

Next move: keep it calm. "Maybe. In our house, you explain every line, and the checks are AI-free." Then run AS-03 Explain Your Code on something recent. If they can explain it, fine. If they can't, it isn't theirs yet, and now you both know.

"I don't get how it learns."

What it usually means: the abstract version came too early.

Next move: back to cards. Run AI-02 twice more with very easy rules, then once with a lopsided deck. The word "training" should mean "the cards I learned from" before it means anything bigger.

When to slow down, go ahead, or step back

Slow down if your student still says the AI "knows" or "wants" things. **Go ahead** when they explain training data, lopsided data, and next-word prediction in their own words. **Step back to paper** whenever a chatbot session turns into chatting.

The rescue order holds: ask → wait → hint → short demo on your own copy → hand it back. Here, “your own copy” means your own card deck or your own copy of the code.

Tools, including AI

Most of this chapter needs index cards, paper, and a pencil.

- **CodeAI’s AI for Oceans** (Hour of AI) for AI-01, parent-run at 11–12.⁷⁷
- **CodeAI’s AI and Machine Learning materials.** Its AI Discoveries course says its AI tools are “built for students ages 11 and up,” but its AI Tutor is labeled for students 13 and up, so in this book the tutor stays off at 11–12.^{11,55,12}
- **Python** (Thonny or any editor from Chapter 2) for the next-word guesser at 13–14.
- **A library website or encyclopedia** for checking claims in AI-05.

AI in this chapter. The chapter is *about* AI, but most of it runs without one. When you do use a chatbot (AI-05, AI-07), follow the AI box in **The Computing Hour**: parent-operated at 11–12, supervised under the terms at 13–14, no private details, no photos of children. AI detectors aren’t used to judge a student’s work; “explain it to me” is the check.

What “done enough” looks like

Move on when most of these are true. Skill, not birthday.

- ☐ Your student explains **training data** and a **classifier** using the card game as an example.
- ☐ They explain how **lopsided data** causes wrong guesses, and suggest one fix.
- ☐ They explain why a chatbot can **invent a source**, using the words “likely” and “true.”
- ☐ They’ve **checked at least one AI claim** against a real source (AI-05).
- ☐ They can name one way AI uses **real resources** (chips, data centers, electricity).
- ☐ They fill out a **Whose Work Is It? card** for a project without reminders, and end each Save and Explain with “Help I used: ...”.
- ☐ **13–14:** they’ve explained AI-suggested code line by line before using any of it (AI-07).
- ☐ They answer the talk-box opening about one AI they met this week.

A short check for the file (Chapter 9 gathers these): **AS-03 Explain Your Code** on the next-word guesser (paper or Python). Your student explains each step or line; you write one sentence they say. The Whose Work card from AI-06 goes in the portfolio too.

Keeping it alive. Each semester, run one AI-05 check on a fresh topic and one OUT-06 errand. AI tools change fast. The questions in the talk box don’t.

You’ve given your student something many adults don’t have: a clear picture of what AI is, where it goes wrong, and an honest rule for using it. That will matter for a long time.

Chapter 9

Records, Portfolio, and Transcript



This chapter is for the parent. It gathers the habits from every other chapter into a file that shows what your student can do, answers anyone who needs to know, and doesn't take over your Fridays. The short version: a dated Debug Log, a handful of finished projects, a few honest checks each month, and a one-page summary at the end of the year.

Why keep records

Records do three jobs.

They show growth. In September, your student's Debug Log entry might read, "It didn't work." By spring, it reads, "Expected 10, got 9. The loop stopped one early. Changed `range(1, 10)` to `range(1, 11)`." You won't remember that change unless it's written down. Neither will your student, and seeing it is one of the most encouraging things a young programmer can do.

They answer a reader outside the house, if there is one. Some states ask for a portfolio, a test, an evaluation, or just a list of subjects. Some ask for very little at this age.

They give a future school, co-op, or program something readable. A one-page summary with a plain course name, a short description, and a few strong pieces of work travels well.

What the law usually asks

Homeschool law is set state by state, and it changes. This book can't tell you what your state requires, and this isn't legal advice. What it can do is show the range, using five states with very different rules.^{78,79}

- **New York** lists required subjects for grades 7 and 8: English, history and geography, science, mathematics, physical education, health, art, music, practical arts, and library skills. Computing isn't on the list.
- **Pennsylvania** lists required courses at the elementary and secondary levels, and asks for a portfolio that includes a log made at the time of instruction, samples of the student's work, and, in grade 8, standardized test results in reading and math. Computing isn't on the required list.
- **Texas** asks families to follow a written course of study that includes good citizenship, along-side reading, spelling, grammar, and math.
- **North Carolina** asks for an annual standardized test that measures English grammar, reading, spelling, and mathematics.
- **Virginia** asks the parent to send a list of the subjects to be studied each year.

None of the five names computer science for homeschooled students in grades 6 to 8. That doesn't mean computing doesn't count. It means teaching it is your choice, and your records show what happened. It's also a choice fewer families make than you might think: in a 2016 federal survey, about 17 percent of homeschooled students in grades 6 to 8 had computer science as part of their home instruction that year.⁸⁰ Read your own state's current law, and HSLDA's state pages are a convenient starting point.^{78,79}

The portfolio

A portfolio is a folder of work that shows what your student can do. Think of it as a few strong pieces with notes, not a box of everything.

What goes in, over a year:

- **Six to eight finished projects** at 11–12 (up to ten at 13–14), each with:
 - version saves and a one-line changelog (PROJ-02);
 - a short test table: input, expected, actual;
 - a Whose Work Is It? card (AI-06), even when the card says “no help used.”
- **The Debug Log**, the dated notebook (paper or a plain file) that holds predictions, bugs, clues, and fixes. Three or four sample pages per semester are plenty for the portfolio; the whole log stays at home (BUG-08).
- **A monthly Cold Trace** (AS-01), dated.
- **One Build From a Blank per semester** (AS-04), dated.
- **A demo-day feedback page** for at least one project (PROJ-03).
- **Screenshots cropped to the work.** First names only; no faces, school, or town. See Family safety basics in **The Computing Hour**.
- **For 13–14:** a note that two-step login is on for the accounts your student uses, and the signed family device agreement from Chapter 7.

A sample Debug Log page. Here is an illustration, not a real family.

12 November. Guessing game v4. *Predicted:* “Too low” when I guess 3 and the number is 7. *Actual:* It said “Too high.” *Clue:* I swapped the < and > in the if. *Fix:* Swapped them back. Tested 3, 7, and 9. *Learned:* Test a number on each side, not just one.

A page like that shows prediction, testing, and reasoning in six lines. Three or four of those per semester say more than a stack of finished programs.

How to name files. Date first, then a plain description: 2026-10-14-guessing-game-v3.py. Files sort themselves into order, and anyone can find them (SYS-03, from Chapter 6). Keep a top folder per school year, with subfolders for projects, checks, and the log.

What stays out. Generated code, pages, or pictures filed as your student’s work. Copied tutorial code without credit. Anything your student can’t explain. Real passwords, login codes, or the private Footprint Audit list from Chapter 7.

Projects and demo days

A project every four to six weeks is a good rhythm. Each one has a beginning, a middle, and an end you can file.

In-home — PROJ-01 Project One-Pager

- **Time:** 20 minutes.
- **Materials:** one sheet of paper.
- **Safety and privacy:** first names only.
- **The fun:** planning something your student actually wants to build.
- **The skill:** problem, user, features, test plan, and schedule on one page.
- **The crumb:** the one-pager, filed at the front of the project folder.

11–12: You ask the questions and your student answers: “Who’s it for? What does it need to do first?” Three features is plenty.

13–14: Your student writes it, and you, as the first user, sign off on the test plan.

In-home — PROJ-02 Version Saves and Changelog

- **Time:** 10 minutes, at the end of each work session.
- **Materials:** the project folder and a plain text file named `changelog`.
- **Safety and privacy:** files live in two places.
- **The fun:** watching the project grow from v1 to v6.
- **The skill:** saving numbered versions, each with a one-line note.
- **The crumb:** the changelog file: “v3: added a score. v4: fixed the score resetting to zero.”

In-home — PROJ-03 Demo Day

- **Time:** 20 minutes.
- **Materials:** the finished project, an audience (family, grandparents on a video call, or a co-op group), and a feedback sheet.
- **Safety and privacy:** show the project, not personal details; video calls with family only.
- **The fun:** a real audience, applause included.
- **The skill:** explaining your work to users and collecting their feedback.
- **The crumb:** feedback notes, with one change your student would make next.

Your student shows the project, explains one part of the code, and asks two questions: “What was confusing?” and “What would you add?” The feedback page goes in the portfolio.

Competitions. If your student enters a contest from Chapter 10, such as Bebras or the Congressional App Challenge, file the entry, the date, and any certificate as an artifact. A program’s certificate is evidence of something your student did. It isn’t a course, a credit, or a grade.

The check set

Six short checks, gathered from across the book, tell you more than any test. Three of them are always AI-free and tutorial-free: Cold Trace, Debug on the Spot, and Build From a Blank. They show what lives in your student’s head.

- **AS-01 Cold Trace** (10 minutes, monthly). A short program on paper your student hasn’t seen. They predict the output, line by line, without running it. Date the sheet.
- **AS-02 Debug on the Spot** (15 minutes). You plant one bug in a working program. Your student finds and fixes it while talking aloud. You write one line: what they did and how they found it.
- **AS-03 Explain Your Code** (10 minutes). Your student explains one of their own programs line by line. You write down one sentence they say.

- **AS-04 Build From a Blank** (30 minutes, once a semester). A small spec, an empty file, no tutorial, no AI. For example: “Ask for three numbers and print the largest.” Date the file and file it as is, bugs and all.
- **AS-05 Whose Work Audit** (10 minutes). Pick a project, check its Whose Work card against the code, and ask about two lines. Your student signs an audit line: “This card is accurate.”
- **AS-06 Portfolio Pick with Reasons** (15 minutes, end of semester). Your student chooses the pieces that best show what they can do and writes one sentence about each.

Sample Build From a Blank specs. At 11–12: “Ask for a name and print a greeting three times,” or “Ask for two numbers and say which is bigger.” At 13–14: “Write a function that takes a list of scores and returns the average, and test it with three lists,” or “Make a page with a heading, a list, and one button that changes the text.” Keep each spec to one or two sentences, and keep the old ones; next semester’s spec should be a little harder.

Recording what you see. Once a semester, mark each big idea as seen, growing, or not yet. This is a record of what you’ve watched happen, not a score, a percentile, or a validated rubric.

Big idea	Seen	Growing	Not yet
Planning (one-pagers, pseudocode)			
Python basics (variables, decisions, loops)			
Debugging (Debug Log, Expected/Actual)			
Functions and lists			
Data (collecting, charting, questioning)			
Web (HTML, CSS, a little JavaScript)			
Systems and networks			
Security habits			
AI understanding and honest use			

Add a date and a one-line note for anything marked “seen”: “Explained why the loop ran one short (AS-03, 12 March).”

A portfolio-pick conversation (optional)

At AS-06 time, use the talk-box opening for this chapter: “Which piece shows what you can really do — and could you explain every line of it?”

Here is an illustration, not a real family.

Parent: Which piece shows what you can really do — and could you explain every line of it? **Student:** The quiz game. Except the part that shuffles the questions. I copied that from the book. **Parent:** Good catch. Is it on the Whose Work card? **Student:** It is now.

Count to five in your head after you ask. When the question is about code on the screen, let them read as long as they need. If your student picks the flashiest project instead of the one they understand best, ask, “Which line in it would you most like to explain?”

A year of records at a glance

- **September:** set up the year folder (two places), start the hours log and the Debug Log, and sign or review the family device agreement.
- **Every week:** the Friday routine (below).
- **Every month:** one Cold Trace.
- **Every four to six weeks:** a finished project with its one-pager, changelog, test table, and Whose Work card, and a demo when you can.
- **Each semester:** a Build From a Blank, a Whose Work Audit, a portfolio pick, and the seen/growing/not-yet table.
- **June:** the one-page year summary and the narrative.

That's the whole system. Most weeks, it's ten minutes.

Hours and the year summary

The hours log is simple: date, minutes, and what you did, one line per session. “14 Oct, 55 min, guessing game v3, fixed the score bug.” A paper calendar or a spreadsheet both work. It's a record, not a quota. Many families find that two or three Computing Hours a week, plus a few errands and short checks, add up faster than they expected.

The year summary is one page. It holds the course name, a one-line description, the main topics, a list of projects, the check set dates, the seen/growing/not-yet table, approximate hours, and a short narrative paragraph written by you.

Writing the narrative

The narrative paragraph is where you sound like yourself. Three or four sentences are enough. These starters help:

- “This year, [name] learned to...”
- “The piece that best shows this is... because...”
- “A habit that grew: ...”
- “Still growing: ...”
- “Next year: ...”

Be specific. “Learned loops” says little. “Can predict how many times a loop runs and fix it when it's off by one” says a lot. Name one thing that's still growing; an honest “not yet” makes the rest of the page more believable.

If someone asks for a grade

Most families don't need letter grades at this age. If your state, an evaluator, or a school asks for one, set the criteria in writing at the start of the semester, and grade against them at the end. For example:

- projects finished, each with a test table and a Whose Work card;
- the monthly Cold Traces done, and the errors explained;
- one Build From a Blank that runs, even if it's simple;
- Debug Log entries that show Expected/Actual and a fix.

Then the grade describes work you can point to. Write the criteria on the summary page so a reader can see what the grade means.

Transcript labels

Use a course name a stranger can read.

- **Computer Science** is the default for most years with this book.
- **Introduction to Programming** fits a year that was mostly Python and projects.
- **Digital Literacy** fits only a year that was mostly safety and everyday use.

Put the details in the description line, not the title. Never use a brand, a program, a tool, or a book title as the course name, and don't claim a certificate from this book; there isn't one. At ages 11 to 14, the summary doesn't need credits.

If your fourteen-year-old's year will count for high school. Some families count a fourteen-year-old's year as ninth grade. Whether this year counts toward high school is a family decision within your state's rules. If it does, title the course plainly (for example, "Introduction to Computer Science"), keep the hours log carefully, and keep the check set going. A common homeschool guideline from HSLDA suggests at least 150 logged hours for one full credit in a core course and 120 or more for an elective. That's a guideline, not a law.⁸¹ Note on the record that the credit is family-assigned. For designing high-school computing courses, see *Computing for Emerging Adults*, the next book in this series.

Honest and dishonest records

Records are only useful if they're true. That's not a moral lecture; it's practical. A record that overstates what your student can do sets them up for a class they're not ready for.

Honest	Not honest
"Wrote a quiz game in Python; borrowed the shuffle code from the book, with credit."	"Wrote a quiz game in Python," with borrowed code unmarked.
"Help I used: a chatbot explained a KeyError; I fixed it myself."	Generated code filed as the student's own.
A Build From a Blank filed with its bugs.	Only the polished final versions.
"Bebras Cadets, participation certificate" under artifacts.	The certificate listed as a course or credit.
"About 90 hours, from the log."	A round number nobody wrote down.

Anything an AI produces is made by a tool, never your student's own work, and never a portfolio piece on its own. The Whose Work card travels with every project, and the AI-free checks show what your student can do without help. For the full rules, see the AI box in **The Computing Hour**.

Special situations

Two students at different levels. Keep separate folders and separate logs, even if they share a Computing Hour. An older sibling's demo day can be a younger sibling's audience, and that counts for both.

A student who dislikes writing. Let them dictate the Debug Log and the Whose Work card while you write, or record a short audio explanation of a project. Oral explanation is real evidence; the AS-03 check is spoken anyway.

A long break. Illness, a move, or a new baby happens. Note the dates in the log ("No Computing Hours, 3–28 February: family move") and pick up where you left off. A record with an honest gap is better than one that pretends.

A student who races ahead. Let the portfolio show it: harder projects, a competition entry, a Build From a Blank with a tougher spec. Keep the checks going. A fast student still benefits from explaining every line.

A student who needs more time. Move at their pace. Seen, growing, and not yet don't come with deadlines. A year where Python basics went from "not yet" to "seen" is a good year.

Where records live

Every file lives in two places: the family drive (or a USB stick) and one backup. Not in a chatbot, and not on a shared bot computer. 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.¹⁴

A Friday routine keeps it painless. Ten minutes, once a week:

1. Copy the week's files to both places.
2. Add up the week's minutes in the hours log.
3. Pick one Debug Log page or test table for the portfolio.
4. Check that every finished project has its Whose Work card.
5. Ask your student, "What are you proudest of this week?" and write down the answer.

Sharing the portfolio. If an evaluator, a co-op teacher, or a school needs to see the work, share copies of the pieces they need, not the whole drive. Check that screenshots are cropped, that only first names appear, and that nothing private from Chapter 7 is in the folder you send.

Two sample records

These are illustrations, not real students.

Example A: an eleven-year-old in grade 6

- *Course:* Computer Science
- *Description:* Algorithms and pseudocode; first Python programs (variables, decisions, loops); debugging with a log; spreadsheets and charts; binary and networks; online safety habits; AI concepts (parent-operated).
- *Evidence:* six projects with version saves; monthly Cold Traces; Debug Log samples; the family device agreement.
- *Hours:* about 90 (log attached).
- *Narrative:* “This year she learned to predict before running and to keep a Debug Log. Her best piece is a number-guessing game she can explain line by line. Next year: functions and a first website.”

Example B: a thirteen-year-old in grade 8

- *Course:* Introduction to Programming
- *Description:* Python (functions, lists, testing); a three-page website with HTML, CSS, and a little JavaScript; a data investigation; encryption concepts; cybersecurity habits; supervised AI use with disclosure.
- *Evidence:* nine projects with Whose Work cards; a Build From a Blank each semester; demo-day feedback.
- *Artifacts:* Bebras (Cadets) participation certificate.
- *Narrative:* “He plans with a one-pager, tests with a table, and explains AI-suggested code before using any of it. His demo-day feedback led to a clearer menu in his quiz game.”

If your student moves to a school

Bring the one-page year summary, two or three portfolio pieces your student can explain, and a recent Cold Trace and Build From a Blank. A teacher can read those in five minutes. Ask your student to be ready to explain one program aloud; that does more than any page.

Looking ahead

Your student will leave this book with a year or more of honest records: projects they can explain, checks done without help, and a Debug Log that shows how they think. At fifteen or so, *Computing for Emerging Adults* picks up from here, with high-school course design, credits, and transcripts.

Keep the folder. Years from now, that September page that said “It didn’t work” will be a pleasure to read.

Chapter 10

Resources



This chapter is for the parent. It lists tools, books, programs, and references that fit this book, sorted by what they're for. It isn't a ranking, and nothing here is an advertisement. You can teach the whole book with a library card, paper, a pencil, and one shared computer. Everything below is optional.

Three habits make the list easy to use:

- **Library first.** Borrow before you buy, whether it's a book, a board, or an hour on a library

computer.

- **Free first.** Every core tool in this book has a free option. Prices aren't listed here, because they change.
- **Check current terms and dates.** Sites rename, change their age rules, and occasionally close. Every link below was checked on 6 October 2026. Before you rely on one, look again, and check the current terms for your student's age.

How to read this list

Each entry tells you what it's for, which ages it fits, and whether it needs an account. The account rules follow Family safety basics in **The Computing Hour**: at 11–12, you create and hold every login; at 13–14, your student may hold passwords for family-approved tools, with your sealed copy, and only on services whose terms allow their age.

When a resource includes an AI feature, the rules in the AI box in **The Computing Hour** apply, whatever the resource's own page says.

Choosing a tool: five questions

Before adding anything new, run it past five questions. They take a minute, and they answer most of what you need to know.

1. **Is it free, or is there a free version that does what we need?**
2. **Does it need an account, and do its terms allow my student's age?** If an account is needed at 11–12, you create and hold it.
3. **Does it have AI features, and can I turn them off or supervise them?** The AI box in **The Computing Hour** decides, not the tool's own settings.
4. **Can my student's work be saved to our own computer?** Files that live only on a website disappear when the website does. Keep everything in two places.
5. **Does it help the habit, or replace it?** A tool that lets your student predict, test, and debug helps. A tool that writes the program for them doesn't, however impressive it looks.

Start at the library

Your public library is the best computing resource most families have. It usually offers:

- **Books** on Python, web design, and how computers work, at several reading levels. Ask a librarian for beginner Python books written for young teens, and look at the publication date; for Python, newer is usually better.
- **Public computers** for a family with one device or none.
- **Programs and clubs**, sometimes including coding clubs or maker spaces with boards like the micro:bit to borrow or use on site.
- **Interlibrary loan** for the book your branch doesn't have.

- **E-books and audiobooks** through the library's apps, on a library card. Programming books read better on paper or a large screen, but books about computing history and how things work are fine on a tablet.

A good librarian is a resource in their own right. Tell them your student's age and what they're learning, and ask what's new.

What to look for in a book. A good beginner programming book for this age uses a current version of Python 3, shows short programs your student types in and changes, explains errors rather than hiding them, and includes small projects with room to make them your own. Skim a chapter before borrowing: if the examples are long and copied without explanation, keep looking. Books on how computers and networks work, and on the history of computing, make good reading-aloud companions for Chapters 6 to 8.

Free, unplugged

About half of this book's work happens away from the screen. These keep it going.

CS Unplugged. A free collection of computer science activities with cards, string, chalk, and games, from the University of Canterbury in New Zealand. Activities are marked by age, and several fit this book well: its searching activities are listed for ages 11 to 14, and its sorting networks for ages 5 to 14. The materials are under a Creative Commons license that allows sharing with credit.²⁵ *Fit:* unplugged practice, co-op days, a sibling who's younger.

Bebras Computing Challenge. A free, international puzzle challenge that tests computational reasoning without coding. In the United States it usually runs in November (9 to 20 November in 2026); students have 45 minutes, and the age groups include Benjamins (10–12), Cadets (12–14), and Juniors (14–16). An adult coordinator registers students, and past puzzles make good practice.⁸² *Fit:* a short, low-pressure first competition.

This book's own unplugged try-its. Every chapter has activities that need only paper, cards, or a walk. They're the backbone of the Hour.

Free, on a screen

Python and its documentation. Python is free from python.org, and its official documentation, including a beginner's tutorial, lives at docs.python.org.⁸³ *Fit:* the main language of this book. The tutorial is dense; use it as a reference at 13–14 rather than a first lesson.

Thonny. A free, beginner-friendly Python editor that comes with Python built in, runs offline, needs no account, and lets your student step through a program line by line and watch the variables change.² *Fit:* this book's default Python tool, especially for Chapters 2 to 4.

MDN Learn web development. Mozilla's free, well-organized lessons on HTML, CSS, and JavaScript.⁴⁷ *Fit:* Chapter 5, mostly as a parent reference at 11–12 and a student reference at 13–14. All you need is a text editor and a browser.

Raspberry Pi Code Editor and projects. A free browser editor for Python and web pages, with a large library of free step-by-step projects from the Raspberry Pi Foundation.³⁵ *Fit:* Python in a browser when you can't install software. You can use the editor without an account; saving work online needs one, and for creators under 13 a parent or guardian sets it up with their own email address. Check the current rules before you start.

micro:bit Python Editor. A free browser editor for the micro:bit, a small programmable board. Its guide says it supports “students aged 11-14,” and it includes a simulator, so your student can write and test programs even without a board. It runs in a web browser and can work offline.⁶⁴ *Fit:* physical computing, before or instead of buying hardware.

Scratch. MIT's free block-based language. At this age it's best for quick sketches: planning a game's logic before writing it in Python, or showing a younger sibling an idea. Anyone under 16 registering a new account needs a parent's or guardian's permission by email; without it, projects can be made and saved, but not shared.⁸⁴ *Fit:* sketches and siblings, not the main language.

CodeAI (formerly Code.org) courses. CodeAI offers a free middle-school course, CS Discoveries, listed for grades 6 to 10, and an AI Discoveries course whose AI tools are “built for students ages 11 and up.” Both need an account; at 11–12, you create and hold it. CodeAI's AI Tutor is labeled for students 13 and up, so in this book it stays off at 11–12 and, at 13–14, is used only with you enabling and supervising it.^{54,55,11,36,12} CodeAI's Python Lab is “designed for students aged 13+.”^{54,55,11,36,12} *Fit:* a ready-made course spine for families who want one, used alongside this book's habits.

Not recommended now: Trinket. It was a popular browser Python tool, but it shut down on 31 August 2026, and its content is no longer available.³⁷ If an older book or worksheet points to it, use Thonny or the Raspberry Pi Code Editor instead.

A starter kit by band

For 11–12. Paper, pencils, index cards, and a Debug Log notebook. Thonny on the family computer. A library book on beginning Python. CS Unplugged for unplugged days. The micro:bit simulator if your student likes gadgets. Any accounts, such as a CodeAI course, are yours. That's a full year.

For 13–14. All of the above, plus MDN for web pages, the Raspberry Pi Code Editor for working away from home, and a first competition if your student wants one (Bebras is the gentlest). Accounts your student holds follow the terms, with your sealed copy and two-step login on. A Git preview if they're curious.

When a tool changes or closes. It happens; Trinket is the recent example. If your student's files live in two places, a closure is an inconvenience, not a loss. Open the project in another editor and carry on. When a site renames itself, as Code.org did when it became CodeAI, the bookmarks change and the lessons usually don't.

Fit at a glance

Need	Fit	Ages and accounts
Python at home, offline	Thonny	Free; no account
Python in a browser	Raspberry Pi Code Editor	Free; check current account rules
Python in a course	CodeAI Python Lab	Designed for 13+
A full middle-school course	CodeAI CS Discoveries or AI Discoveries	Free; account, parent-made at 11–12; AI Tutor 13+
Blocks for sketches	Scratch	Under-16 accounts need a parent's email
Web pages	Text editor, browser, MDN	No account
Physical computing	micro:bit Python Editor with simulator	Free; hardware optional
Unplugged	CS Unplugged	Free
Puzzle challenge	Bebras	Free; an adult coordinator registers
Security practice	CyLab Security Academy	13–14, with a parent
Version control (preview)	Git with a hosted service	13+ for most services; optional

Resources by chapter

Chapter	Most useful here
1. What computing looks like at 11–14	CS Unplugged; a library book on how computers work
2. From blocks to text: Python	Thonny; Scratch for a quick sketch; docs.python.org at 13–14
3. Debugging	Thonny's step-through view; the Debug Log
4. Functions, lists, and data	Thonny; CS Unplugged searching and sorting
5. Building for the web	A text editor, a browser, and MDN; Raspberry Pi projects
6. How computers and networks work	CS Unplugged; the micro:bit simulator; a library book
7. Safe, smart, and kind online	CISA Secure Our World; the FTC's COPPA page; the AAP Family Media Plan
8. AI up close	CodeAI's AI for Oceans, parent-run at 11–12; the AI literacy framework (for you)
9. Records	HSLDA's state pages; your own state's official text

Optional hardware

None of this is required. Borrow first if you can.

- **A micro:bit.** A small, sturdy board with lights, buttons, and sensors, programmed in Python. Try the simulator first; if your student loves it, a board is a natural next step. Some libraries lend them.
- **An Arduino-style board** for a 13–14-year-old who enjoys the micro:bit and wants wires, sensors, and motors. These take more patience. If an online account is needed for the editor, you create and hold it. Arduino links every account for a child under 18 to a parent, and a parent creates or approves accounts for children under 14. Check the current terms.⁸⁵
- **A used laptop.** Any reasonably recent used laptop that runs a current, updated operating system will run Thonny and a browser comfortably. An adult wipes and sets it up. See Family safety basics in **The Computing Hour** on old hardware.

Before you buy anything, ask: Can we borrow it? Is there a free simulator? Will we use it more than twice? Does it need an account, and who will hold it?

Competitions and programs

All optional. Competitions suit some students and not others, and a student who'd rather build quietly is doing just as well. Dates and rules change every year; check with each program before you plan around it. Anything your student earns is filed as an artifact, never as a course or credit (Chapter 9).

Deciding whether to enter. Ask your student, not just yourself. A good first contest is short, free, and low-stakes, and the point is the puzzles, not the ranking. Watch how your student feels afterward. If it was fun, try another next year. If it was stressful, that's useful to know, and building projects at home is every bit as good.

- **Bebras.** Above. A gentle first contest.
- **American Computer Science League (ACSL), Junior Division.** For “junior high and middle school students who are learning programming or who have coding experience,” with no student beyond grade 9. It combines a short written test with a programming problem. An adult advisor registers, not the student; registration typically closes at the end of December (31 December 2026 for the 2026–27 season).⁸⁶ *Fit:* a 13–14-year-old who likes Python puzzles.
- **CyberPatriot, Middle School Division.** A team cybersecurity competition from the Air & Space Forces Association, with two to six students and an adult coach. The current rules book lists homeschools among the groups that may field middle school teams, with verification steps for independent homeschools. Check with the program about eligibility and dates before you plan; registration typically closes in early October.⁸⁷ *Fit:* a co-op or a group of families, at 13–14.
- **Congressional App Challenge.** Students build an app in any language and enter it in their congressional district. Entrants must be middle or high school students, alone or in teams of up to four; the deadline is usually in late October (26 October in 2026).⁸⁸ *Fit:* a 13–14-year-old with a finished project and a reason to build it.
- **CyLab Security Academy (formerly picoCTF).** Carnegie Mellon University's free cybersecurity learning platform, with practice challenges and competitions. Older picoCTF logins

created before 8 May 2026 still work.⁸⁹ *Fit:* 13–14, with a parent and an account that follows the terms.

- **USA Computing Olympiad (USACO).** Free online programming contests aimed at “high-school computing students at all levels.”⁹⁰ *Fit:* a strong 13–14-year-old who’s ready for harder problems. It’s a stretch, and that’s fine to wait on.

A preview of version control

Professional programmers use a tool called **Git** to save versions of their work and share it. Hosted services such as GitHub keep copies online. At 13–14, a curious student can try it, with a parent-created account; GitHub’s terms say a user “must be at least 13 years of age.”³⁸ It’s never required. The version saves and changelog in Chapter 9 teach the same idea with a folder and a text file.

For the parent: references worth knowing

These are for you. Your student doesn’t need them.

CSTA standards for grades 6 to 8. The Computer Science Teachers Association’s 2026 standards describe what’s age-reasonable for middle school. Use them as a map, not a checklist; they’re written for schools. CSTA’s guidance pictures middle-school computing as the equivalent of a yearlong course in the grade 6–8 band, which is roughly what this book covers. Licensed CC BY-NC-SA 4.0.^{17,15,16}

Reading a standard’s code. Take MS-PRO-TR-19. **MS** is the band (middle school, grades 6–8). **PRO** is the concept: Algorithms & Design (ALG), Programming (PRO), Data & Analysis (DAT), Systems & Security (SYS), or Computing & Society (SOC). **TR** is a two-letter topic tag within the concept (the TR standards are about testing and refining). **19** is the number. MS-PRO-TR-19 reads: “Use systematic strategies to test, refine, and document changes to a computing technology to meet the intended purpose.” That’s the Debug Log and the changelog, in standards language.

AI literacy framework. The OECD and the European Commission have published *Empowering Learners for the Age of AI*, an AI literacy framework for primary and secondary education. It describes AI literacy as knowledge, skills, and attitudes that help learners understand how AI works, evaluate its outputs, and use it ethically and creatively.⁹¹ Useful if you want the bigger picture behind Chapter 8.

Children’s privacy. The FTC’s page on protecting your child’s privacy online explains COPPA in plain words.⁴

Online safety. CISA’s Secure Our World pages cover the four habits from Chapter 7, with short printable tips.⁶⁷

Phones and media. The American Academy of Pediatrics’ Family Media Plan is a free, fill-in tool for writing your family’s rules, and its HealthyChildren article on a child’s first phone walks through readiness.^{92,93}

If an image is shared. NCMEC’s Take It Down service helps remove images taken of someone under 18 from participating platforms.⁷⁵

Homeschool law. HSLDA’s state pages summarize each state’s requirements; read the official text too.⁷⁹

A preview: AP Computer Science Principles. Years from now, your student may take AP Computer Science Principles, a high-school course from the College Board. The College Board says its AP computer science courses require familiarity with concepts from Algebra 1 and don’t require any prior coding experience.⁹⁴ AI is allowed as a supplement, but students must review and understand any code written with AI, be ready to explain it in detail, and acknowledge it.²³ That’s years away, and the habits in this book are the right ones for it.

Low-device and no-device homes, co-ops, and siblings

One device, or none. Most U.S. households had at least one type of computer in 2021, but “at least one” can mean a single phone.⁹⁵ If yours has one shared computer, this book works: about half the work is unplugged, and the screen parts are short. With no computer, library computers and a USB stick for files cover the Python chapters. A phone is a poor place to write code, but it’s fine for reading this book’s examples and taking photos of paper work for the portfolio.

Co-ops. Many try-its work better with a group: Phish or Fine? as a team game, demo days with a real audience, a CyberPatriot team. One parent can lead the Hour for several families. Keep each student’s records separate, and first names only on anything shared.

Siblings. A younger sibling can play the user in a demo, try the unplugged activities, or work through *Computing for Little Thinkers* alongside. An older sibling can be the audience, the tester, or a patient helper, as long as they use the rescue order and don’t take over the keyboard.

Different learners. Most editors let you enlarge the text, and a bigger font helps everyone. A student who finds typing slow can plan on paper first and type less. A student who reads slowly does well with the unplugged try-its and with spoken explanations, which count fully in the check set (Chapter 9). Move at your student’s pace; the “done enough” lists are about skill, not speed.

Where to go next in this series

- *Computing for Little Thinkers* for a younger sibling, or for a gentle unplugged start with an older child.
- *Computing for Emerging Adults* for ages 15 to 18: high-school courses, credits, and transcripts.
- *Computing Refresh for Adults* for a parent who’d like to understand the ideas in this book more deeply.

Pick one tool, one library book, and one project. That’s enough to start.

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 6 October 2026. A few pages refused automated reading that day, and the notes say how their wording was checked. The CDC's page on young teens was checked against an archived copy of the same page. Meta's AI terms were checked against a search engine's copy of the official page, because the age section does not open for automated readers. The PRIMM study's numbers (its control group and effect size) come from the authors' accepted manuscript as shown in a search engine's excerpt, because the publisher's copy would not load. The Parsons-problem study is described from its abstract. 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 middle-schoolers 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. The Python samples were run in Python 3.13 to check their output, while Thonny installs Python 3.14, so an error message may be worded slightly differently on your screen. The micro:bit programs follow the micro:bit's official documentation but were not run on a physical board. The five-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 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: 6 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 6 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. Thonny, “Thonny: Python IDE for beginners,” thonny.org, 2026, <https://thonny.org/>, accessed 6 October 2026. Free, with no account needed; version 5.0.0 installers for Windows, macOS, and Linux include Python 3.14 (“Thonny comes with Python 3.14 built in, so just one simple installer is needed”). The page says to “select View → Variables” to watch variables; “Just press Ctrl+F5 instead of F5 and you can run your programs step-by-step, no breakpoints needed. Press F6 for a big step and F7 for a small step”; and that Thonny “highlights syntax errors. Unclosed quotes and parentheses are the most common beginners’ syntax errors.” Thonny is open-source software under the MIT License (Thonny project, “LICENSE.txt,” GitHub, <https://github.com/thonny/thonny/blob/master/LICENSE.txt>, accessed 6 October 2026).
3. Colleen M. Lewis, “Is Pair Programming More Effective than Other Forms of Collaboration for Young Students?,” *Computer Science Education* 21, no. 2 (2011): 105–134, <https://doi.org/10.1080/08993408.2011.579805>, accessed 6 October 2026. Two summer enrichment classes for students entering sixth grade, using Scratch. Students who pair-programmed and students who worked on their own computers with frequent collaboration “did not differ significantly in their performance on daily quizzes or responses to attitudinal survey questions,” and students on their own computers “completed exercises more quickly.” A small study of two classes, not a general verdict on pairing.
4. Federal Trade Commission, “Protecting Your Child’s Privacy Online,” FTC Consumer Advice, May 2026, <https://consumer.ftc.gov/articles/protecting-your-childs-privacy-online>, accessed 6 October 2026. “As a parent, you have control over the personal information companies collect online from your kids under 13.” The Children’s Online Privacy Protection Act (COPPA) Rule covers sites and services, including apps, “directed to children under 13,” and general-audience sites that know they are collecting personal information from children that age; they must “notify parents directly and get their approval.” Personal information includes a child’s full name, address, phone number or email address, physical location, “photos, videos and audio recordings of the child,” government identifiers, biometric identifiers, and persistent identifiers such as IP addresses. COPPA governs how companies collect children’s data; it does not rate content or certify a site as safe.
5. Mary Budd Rowe, “Wait-Time and Rewards as Instructional Variables: Their Influence on

Language, Logic, and Fate Control,” paper presented to the National Association for Research in Science Teaching, April 1972, ERIC ED061103, <https://eric.ed.gov/?id=ED061103>; and Robert J. Stahl, “Using ‘Think-Time’ and ‘Wait-Time’ Skillfully in the Classroom,” ERIC Digest, May 1994, ERIC ED370885, <https://eric.ed.gov/?id=ED370885>; both accessed 6 October 2026. Rowe’s abstract reports that “Elementary science teachers allow an average of one second for a response to a question,” and that when wait times “are extended to three to five seconds,” responses grow longer and students offer more unprompted answers and ask more questions. Stahl’s digest reports that “after at least three seconds of uninterrupted silence, a significant number of positive outcomes occur for students and teachers.” These are classroom studies. “Count to five in your head” is a pause habit drawn from them, not a measured rule for a kitchen table.

6. Tiffany Munzer et al., Council on Communications and Media, American Academy of Pediatrics, “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 6 October 2026. The statement suggests family-tailored limits ranging up to “1 to 2 hours/d or more of entertainment (not school-related) media for school-aged children and teens,” adding 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.” On phones: “Research does not point to a specific age for teen smartphone readiness, and it varies by teen. Considerations include: teen digital media literacy, truthfulness with caregivers, ability to navigate conflict in social situations, and practical needs. Caregivers should set and monitor privacy, download, content, purchase, and time settings. Various smartphones, flip phones, or watches can minimize access to apps teens may not be ready for.” On controls: set “parental controls (time, downloads, contacts, purchases) using tools on wireless routers, smartphones, tablets, gaming consoles/platforms, social media accounts, and school devices.”

7. World Health Organization, *WHO Guidelines on Physical Activity and Sedentary Behaviour*, WHO, 25 November 2020, <https://www.who.int/publications/i/item/9789240015128>, accessed 6 October 2026. For ages 5–17: “an average of 60 minutes per day of moderate- to vigorous-intensity” physical activity across the week, and children and adolescents “should limit the amount of time spent being sedentary, particularly the amount of recreational screen time.” The guidelines give no numeric screen limit.

8. 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 6 October 2026: “Children 6 to 12 years of age should sleep 9 to 12 hours per 24 hours” and “Teenagers 13 to 18 years of age should sleep 8 to 10 hours per 24 hours.” The Centers for Disease Control and Prevention page “About Sleep” (15 May 2024, <https://www.cdc.gov/sleep/about/index.html>, accessed 6 October 2026) lists the same ranges (school age, 6–12 years, 9–12 hours; teens, 13–17 years, 8–10 hours) and cites this statement.

9. Consumer AI terms, all accessed 6 October 2026; terms change, so check the current versions. OpenAI, “Terms of Use,” effective 1 January 2026, <https://openai.com/policies/terms-of-use/> (the version for users outside the European Economic Area, Switzerland, and the UK, which includes the United States): “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 to use the Services." SpaceXAI, "Terms of Service – Consumer," last updated 11 September 2026, <https://x.ai/legal/terms-of-service>: at least 13, and "If you are a teenager between the ages of 13 and 17 years old, you must have your parent or legal guardian's permission." Microsoft, "Microsoft Copilot Supplemental Terms of Use," effective 18 August 2026, <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"; "If you're under 18, we might turn off or limit some features for legal or safety reasons." Meta, "Meta AIs Terms of Service," effective 13 May 2026, <https://www.facebook.com/legal/ai-terms>: "you cannot access AIs if you are under the age of 13 (or such greater age required in your country or territory)." Anthropic, "Consumer Terms of Service," effective 8 October 2025, <https://www.anthropic.com/legal/consumer-terms>: "You must be at least 18 years old or the minimum age required to consent to use the Services in your location, whichever is higher."

10. Google For Families Help, "Guide your child's Gemini Apps experience," Google, 2026, <https://support.google.com/families/answer/16109150>, accessed 6 October 2026. Children under 13 use Gemini Apps only through a parent-supervised Google account: "A parent must enable access before their child under 13 (or the applicable age in your country) can use Gemini Apps with a supervised account," and Google says its filters are "not perfect." This book names that supervised option so families know it exists; it isn't a recommendation, and this book keeps the parent-operated rule at 11–12.

11. CodeAI (formerly Code.org), "AI Tutor," CodeAI, 2026, <https://code.org/en-US/tools/ai-tutor>, accessed 6 October 2026: "For students age 13+"; the tutor is opt-in, turned on by a teacher, and available in selected course units; "All interactions with the AI Tutor are stored securely and automatically deleted after 90 days." In this book the AI Tutor stays off at 11–12 even where a course page offers it.

12. CodeAI, "Code.org has become CodeAI," 2026, <https://code.org/en-US/codeai>, accessed 6 October 2026: "CodeAI is the same organization" under a new name.

13. Hamsa Bastani, Osbert Bastani, Alp Sungu, Haosen Ge, Özge Kabakçı, 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): e2422633122, <https://doi.org/10.1073/pnas.2422633122>, accessed 6 October 2026. A field experiment with nearly 1,000 high-school math students in Turkey. Students with unrestricted access to a chatbot did better on practice problems and worse on a later exam taken without it; a tutor version with guardrails largely avoided the harm. Off-age evidence for this book: not ages 11–14 and not programming. It is used once, in the AI box in **The Computing Hour**.

14. SpaceXAI, "Frequently Asked Questions," Grok Bot documentation, SpaceXAI Docs, last updated 29 September 2026, <https://docs.x.ai/grok-bot/faq>, accessed 6 October 2026: "Every Bot on your account uses one persistent cloud computer. They share its files, browser sessions, and logins"; "The computer is assigned per user, not per Bot. Do not use separate Bots as a security boundary"; and after a bot is deleted, "Because Bots share a computer, files and logins on that computer may remain."

15. Computer Science Teachers Association (CSTA), *2026 CSTA PK–12 Computer Science Stan-*

dards: Middle School Implementation Guidance, CSTA, 2026, <https://csteachers.org/pk12standards/>, accessed 6 October 2026: “In middle school (grades 6–8) and high school (grades 9–12), students experience the equivalent of a yearlong course at each grade band.” On AI, the guidance says that using an AI tool in another subject “only becomes CS learning when the lesson addresses how the tool works,” and that students learn to “attribute AI-generated code, images, and other content they use.” The guidance is written for schools and districts; it is a reference, not a homeschool requirement.

16. Computer Science Teachers Association (CSTA), *2026 CSTA PK–12 Computer Science Standards*, overview page, CSTA, 2026, <https://csteachers.org/pk12standards/>, accessed 6 October 2026. The standards name five concepts: Algorithms & Design; Programming; Data & Analysis; Systems & Security; Computing & Society. 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.” The standards are licensed CC BY-NC-SA 4.0. They are a map written for schools, not a homeschool requirement.

17. Computer Science Teachers Association (CSTA), “Middle School (Grades 6–8, Ages 11–14),” *Foundational Standards by Level*, 2026 CSTA PK–12 Computer Science Standards, CSTA, 2026, <https://csteachers.org/pk12standards/>, accessed 6 October 2026. Middle school is one grade band, grades 6–8. Standards quoted or matched in this book: MS-ALG-PS-02 “Model a given algorithm with a flowchart or pseudocode that includes a combination of control structures and procedures”; MS-ALG-PS-03 “Verify the accuracy of an algorithm for given inputs”; MS-ALG-ML-06 “Hypothesize how a machine learning model generates classifications or predictions”; MS-ALG-ML-07 “Investigate ways to improve the accuracy of a machine learning model and reduce bias by refining the quality of examples and nonexamples in the training data”; MS-ALG-ML-08 “Evaluate the features and limitations of a machine learning model”; MS-ALG-IM-10 “Examine evidence of beneficial and harmful impacts, ethical issues, and biases of algorithms encountered in daily life”; MS-PRO-PD-12 “Use a procedure to structure code for clarity and reusability”; MS-PRO-PD-13 “Use reference documentation in program development”; MS-PRO-PD-14 “Justify the importance of attribution and intellectual property when developing computing technologies”; MS-PRO-VD-16 “Use variables of multiple data types to store, access, and manipulate data within a program”; MS-PRO-RD-17 “Analyze the roles of iteration, selection, variables, and procedures in a segment of code”; MS-PRO-RD-18 “Analyze AI-generated code for accuracy and usability in a programming project”; MS-PRO-TR-19 “Use systematic strategies to test, refine, and document changes to a computing technology to meet the intended purpose”; MS-PRO-TR-20 “Refine a computing technology based on user feedback to improve its usability and accessibility”; MS-DAT-DC-21 “Evaluate how different levels of precision and granularity in data collection affect accuracy, storage, and analysis”; MS-DAT-DC-22 “Explain how data and its associated metadata can be used to answer questions”; MS-DAT-DC-23 “Use a computational tool to sort, filter, group, and summarize structured data”; MS-DAT-DI-26 “Create data visualizations to show how different design choices can impact the interpretation of the same data”; MS-DAT-DI-27 “Summarize a data investigation process, including potential biases, limitations, and supporting evidence”; MS-DAT-IM-28 “Explain the benefits and risks of allowing personal data and metadata to be collected and used in datasets, including issues of data ownership, privacy, and sovereignty”; MS-DAT-IM-29 “Analyze how decisions made at different stages of working with data can lead to biased data, misleading conclusions, and compromised AI models”; MS-SYS-HW-30 “Examine differences between com-

puting systems based on user needs, system requirements, and potential societal, environmental, and ethical impacts”; MS-SYS-HW-31 “Describe computing devices used in various industries, their basic functions, and how they are used to accomplish tasks or solve problems”; MS-SYS-SE-32 “Explain the effects of not using the CIA triad when working with data”; MS-SYS-SE-33 “Evaluate common types of cyber attacks and preventions”; MS-SYS-NT-34 “Model how information in a network is broken down into packets, transmitted between devices, and reassembled”; MS-SYS-NT-35 “Explain how the resilience of the internet depends on interconnected devices and their roles and functions within the network”; MS-SOC-ET-40 “Evaluate when it is appropriate to use AI and other emerging technologies to solve a problem based on their capabilities, limitations, and environmental impacts”; MS-SOC-HU-43 “Analyze how the decisions humans make when using computing technologies have ethical and social consequences”. How to read a code: MS is the middle-school band; ALG, PRO, DAT, SYS, and SOC are the five concepts (Algorithms & Design, Programming, Data & Analysis, Systems & Security, Computing & Society); then come a two-letter topic tag and the standard’s number. In MS-SYS-SE-32, the CIA triad means confidentiality, integrity, and availability. The materials are licensed CC BY-NC-SA 4.0. The activities matched to these standards are this book’s own teaching moves, and the standards are a reference map of what is age-reasonable, not a homeschool requirement.

18. Centers for Disease Control and Prevention, “Positive Parenting Tips: Young Teens (12–14 years old),” CDC, <https://www.cdc.gov/child-development/positive-parenting-tips/young-teens-12-14-years.html>, accessed 6 October 2026. Young teens “have more ability for complex thought” and “develop a stronger sense of right and wrong”; they “make more of their own choices about friends, sports, studying, and school” and “become more independent ... although parents are still very important.” The CDC suggests parents be clear about goals and “allow your teen input on how to reach those goals.” The CDC site refused automated reading on the access date, so this wording was checked against an archived copy of the same page.

19. Leah H. Somerville, “The Teenage Brain: Sensitivity to Social Evaluation,” *Current Directions in Psychological Science* 22, no. 2 (2013): 121–127, <https://doi.org/10.1177/0963721413476512>, accessed 6 October 2026. A review of evidence that adolescents are especially sensitive to being evaluated by others. This book uses it for one practical habit, keeping feedback about the code, not the coder; it is not a claim about any one child.

20. Allison Master, Andrew N. Meltzoff, and Sapna Cheryan, “Gender Stereotypes about Interests Start Early and Cause Gender Disparities in Computer Science and Engineering,” *Proceedings of the National Academy of Sciences* 118, no. 48 (2021): e2100030118, <https://doi.org/10.1073/pnas.2100030118>, accessed 6 October 2026. Stereotypes that girls are less interested in computer science and engineering are endorsed by children and adolescents “as early as age six (first grade)”; the authors conclude that “addressing interest stereotypes may help improve educational equity.”

21. 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 6 October 2026. Across 105 studies and 539 effect sizes, the meta-analysis reports a moderate overall transfer effect ($g = 0.49$), larger for skills close to programming ($g = 0.75$) than for skills further away ($g = 0.47$); effects were smaller against active control groups, and “School achievement and literacy, however, benefited the least.” Studies vary widely in age and design, so this book treats transfer as

possible, not promised.

22. Shuchi Grover, Roy Pea, and Stephen Cooper, “Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students,” *Computer Science Education* 25, no. 2 (2015): 199–237, <https://doi.org/10.1080/08993408.2015.1033142>, accessed 6 October 2026. A seven-week course designed and studied by its developers with seventh and eighth graders in one middle school. Students showed learning gains and “were able to transfer their learning from Scratch to a text-based programming context”; loops and variables were the hardest concepts for many of them.

23. College Board, “What is the acceptable usage of generative AI?,” AP Computer Science Principles FAQ, AP Central, 2026, <https://apcentral.collegeboard.org/courses/ap-computer-science-principles/course/faq/generative-ai-usage-guidelines>, accessed 6 October 2026: “Students can use Generative AI tools as supplementary resources for understanding coding principles, assisting in code development, and debugging. It is the student’s responsibility to review and understand any code co-written with AI tools, ensuring its functionality. Additionally, students must be prepared to explain their code in detail, as required on the end-of-course exam. Students should include appropriate acknowledgment (i.e., through citation, attribution, and/or by reference) for any code created by generative AI tools.” AP CSP is a high-school course; this book borrows its “explain it” rule because it fits ages 11–14 well.

24. Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick, “Solving Parsons Problems versus Fixing and Writing Code,” *Proceedings of the 17th Koli Calling International Conference on Computing Education Research* (2017): 20–29, <https://doi.org/10.1145/3141880.3141895>, accessed 6 October 2026. Undergraduates who solved Parsons problems spent less time than those who fixed or wrote the equivalent code, with similar learning. Not a study of middle-schoolers.

25. CS Unplugged, University of Canterbury Computer Science Education Research Group, home page and “Topics,” 2026, <https://www.csunplugged.org/> and <https://www.csunplugged.org/en/topics/>, accessed 6 October 2026. Free activities that teach computing without computers, labeled by age range; for example, “Data structures for searching” is marked “Ages 11 to 14” and “Sorting networks” “Ages 5 to 14.” Materials are released under a “Creative Commons Attribution-ShareAlike 4.0 International license.”

26. Python Software Foundation, “About Python,” Python.org, 2026, <https://www.python.org/about/>, accessed 6 October 2026. The page describes Python as “friendly & easy to learn” and “Open,” and lists applications including web and internet development, scientific and numeric work, education, and software and game development. Python is distributed under an open-source license at no cost.

27. Python Software Foundation, “Built-in Functions: input(),” *The Python Standard Library*, Python 3.14 documentation, <https://docs.python.org/3/library/functions.html#input>, accessed 6 October 2026: “The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.”

28. David Weintrop and Uri Wilensky, “Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms,” *ACM Transactions on Computing Education* 18, no. 1 (2017): 1–25, <https://doi.org/10.1145/3089799>, accessed 6 October 2026: in a five-week

study, “students in the blocks condition showed greater learning gains and a higher level of interest in future computing courses,” while “students in the text condition viewed their programming experience as more similar to what professional programmers do.” A study of high-school students.

29. David Weintrop and Uri Wilensky, “Transitioning from Introductory Block-Based and Text-Based Environments to Professional Programming Languages in High School Computer Science Classrooms,” *Computers & Education* 142 (2019): 103646, <https://doi.org/10.1016/j.compedu.2019.103646>, accessed 6 October 2026. The same high-school classes, followed after they moved to Java: after ten weeks of Java, the difference between the blocks and text groups was no longer present.

30. Scratch, “About Scratch,” MIT Media Lab Lifelong Kindergarten Group, 2026, <https://scratch.mit.edu/help/about/>, accessed 6 October 2026: “Scratch is designed especially for ages 8 to 16, but is used by people of all ages.” The page adds that Scratch “is provided free of charge.”

31. Sue Sentance, Jane Waite, and Maria Kallia, “Teaching Computer Programming with PRIMM: A Sociocultural Perspective,” *Computer Science Education* 29, no. 2–3 (2019): 136–176, <https://doi.org/10.1080/08993408.2019.1608781>, accessed 6 October 2026. A study in 13 schools with 493 students aged 11 to 14, compared with a control group of 180 students taught otherwise; after 8 to 12 weeks, the PRIMM group scored higher on a programming test, a small effect ($r = .13$). Developed and studied by the method’s authors.

32. Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner, “Computational Thinking for Youth in Practice,” *ACM Inroads* 2, no. 1 (2011): 32–37, <https://doi.org/10.1145/1929887.1929902>, accessed 6 October 2026. The authors present a “use-modify-create” framework, drawn from programs for middle- and high-school youth, “representing three phases of students’ cognitive and practical activity.” A framework from practice, not a controlled trial.

33. Python Software Foundation, “turtle — Turtle graphics,” *The Python Standard Library*, Python 3.14 documentation, <https://docs.python.org/3/library/turtle.html>, accessed 6 October 2026: “Turtle graphics is an implementation of the drawing tools introduced in Logo in 1967. It was created as an educational tool.”

34. Python Software Foundation, “random — Generate pseudo-random numbers,” *The Python Standard Library*, Python 3.14 documentation, <https://docs.python.org/3/library/random.html>, accessed 6 October 2026: `random.randint(a, b)` will “Return a random integer N such that $a \leq N \leq b$.”

35. Raspberry Pi Foundation, Code Editor, <https://editor.raspberrypi.org/>; Projects, <https://projects.raspberrypi.org/>, including “Getting started with the Code Editor” guides for Python and HTML; and Code Club (Raspberry Pi Foundation), “Accounts for creators,” <https://codeclub.org/en/resources/accounts-for-children>; all accessed 6 October 2026. The editor and projects are free. The Code Club page says an account is “not a requirement; creators can use the Code Editor and access all our digital making projects without an account”; that accounts let creators “save their work in our Code Editor”; that “creators under the age of 13 years will need a parent or guardian to use their email address to set up the account”; and that creators aged 13 and over can set up their own “without needing permission.” The foundation’s account rules have changed before; check

the current rules.

36. CodeAI (formerly Code.org), “Python Lab,” CodeAI, 2026, <https://code.org/en-US/tools/python-lab>, accessed 6 October 2026: “Designed for students aged 13+,” with ages listed as 13 to 18.

37. Trinket, “Trinket has shut down,” trinket.io, 2026, <https://trinket.io/>, accessed 6 October 2026: “Trinket shut down on August 31, 2026. The site is no longer operating, and trinkets, courses, assignments, and account data are no longer accessible.”

38. GitHub, “GitHub Terms of Service,” GitHub Docs, 2026, <https://docs.github.com/en/site-policy/github-terms/github-terms-of-service>, accessed 6 October 2026: “A User must be at least 13 years of age.”

39. Replit, “Terms of Service,” Replit, 2026, <https://replit.com/terms-of-service>, accessed 6 October 2026: “you must be at least 13 years of age, and if you are under 18, you need permission from a parent or guardian.”

40. Andrew Hunt and David Thomas, *The Pragmatic Programmer* (Addison-Wesley), section “Debugging,” excerpt at <https://courses.cs.duke.edu/fall21/compsci307d/readings/pragmatic-programmer-debugging.pdf>, accessed 6 October 2026. The excerpt says, “No one writes perfect software, so it’s a given that debugging will take up a major portion of your day”; advises testing boundary conditions; notes that if you “changed only one thing” and the system stopped working, “that one thing was likely to be responsible”; and, in “Rubber Ducking,” says that “the simple act of explaining, step by step, what the code is supposed to do often causes the problem to leap off the screen and announce itself.” Its footnote tells of a colleague who carried a small yellow rubber duck and placed it on his terminal while coding, and the text adds that if no person is around, “a rubber duck, or teddy bear, or potted plant will do.” This is practitioner advice, not a study.

41. National Museum of American History, Smithsonian Institution, “Log Book With Computer Bug,” object ID 1994.0191.01, https://americanhistory.si.edu/collections/object/nmah_334663, accessed 6 October 2026. The museum’s description says Thomas Edison talked about bugs in electrical circuits in the 1870s; that in 1947 Mark II engineers at Harvard found a moth in a component, taped it into their logbook, and labeled it “first actual case of bug being found”; and that “this log book was probably not Hopper’s, but she and the rest of the Mark II team helped popularize the use of the term computer bug and the related phrase ‘debug.’” The Computer History Museum’s “This Day in History” entry dates the logbook page to 9 September 1947 (<https://www.computerhistory.org/t dih/september/9/>, accessed 6 October 2026).

42. Python Software Foundation, “8. Errors and Exceptions,” *The Python Tutorial*, Python 3.14 documentation, <https://docs.python.org/3/tutorial/errors.html>, accessed 6 October 2026. The tutorial distinguishes syntax errors from exceptions; says the error arrow points to where the error was detected and that “this is not always the place that needs to be fixed”; and says “the last line of the error message indicates what happened.” The error messages quoted in this book were produced by running each sample in Python 3.13; wording can differ slightly between versions, including the Python 3.14 that Thonny installs.

43. Python Software Foundation, “5. Data Structures,” *The Python Tutorial*, Python 3.14 documentation, <https://docs.python.org/3/tutorial/datastructures.html>, accessed 6 October 2026. Describes append (“Add an item to the end of the list”), remove (“Remove the first item from the

list whose value is equal to value. It raises a `ValueError` if there is no such item”), and `index` (“Return zero-based index”).

44. Python Software Foundation, “Built-in Functions,” *The Python Standard Library*, Python 3.14 documentation, <https://docs.python.org/3/library/functions.html>, accessed 6 October 2026. Describes `ord()` (returns the Unicode code point of a character; `ord('a')` returns 97) and `chr()` (the reverse; `chr(97)` returns 'a'), along with `len()`, `max()`, and `sum()`. Every Python sample in Chapters 4 and 6 was run in Python 3.13 to confirm the stated output; the micro:bit samples in Chapter 6 use functions from the micro:bit MicroPython documentation and were not run on a board.

45. Consumer Reports, “What You Need to Know About Exif Data,” <https://www.consumerreports.org/electronics-computers/privacy/what-can-you-tell-from-photo-exif-data-a2386546443/>, accessed 6 October 2026. Explains that smartphones and digital cameras automatically capture details of when, where, and how a photo was taken; that this data travels with the photo file; and that users can turn off the camera’s access to location.

46. CS Unplugged, “Searching Algorithms,” <https://www.csunplugged.org/en/topics/searching-algorithms/>, accessed 6 October 2026. Free unplugged lessons on sequential searching (checking items one at a time) and binary searching (halving a sorted list), and on choosing between algorithms for the same purpose.

47. MDN Web Docs (Mozilla), “Learn web development,” https://developer.mozilla.org/en-US/docs/Learn_web_development, accessed 6 October 2026. A free course and reference for HTML, CSS, and JavaScript, designed to take learners “from ‘beginner’ to ‘comfortable.’”

48. MDN Web Docs (Mozilla), “`rgb()`,” CSS reference, https://developer.mozilla.org/en-US/docs/Web/CSS/Reference/Values/color_value/rgb, accessed 6 October 2026. Describes colors written as red, green, and blue values, each a number from 0 to 255. The space-separated form used in this book, such as `rgb(20 80 140)`, is the current syntax; the comma-separated form is described as legacy.

49. MDN Web Docs (Mozilla), “HTML: A good basis for accessibility,” https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Accessibility/HTML, accessed 6 October 2026. Notes that many screen readers let users jump between headings or list them as a table of contents; that alt text “should always provide a direct representation of the image and what it conveys visually”; that decorative images can use an empty `alt=""`; that link text should make sense out of context (contrasting “Find out more about whales” with “click here”); that text color “has to be significantly different from the background color (a 4.5:1 contrast)”; and that real buttons and links should be used for controls.

50. 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 6 October 2026. Gives the keyboard shortcuts and the right-click “Inspect” route for opening developer tools, notes that Safari’s tools must be enabled first, and says the Console reports the errors the browser meets as it tries to run your code.

51. MDN Web Docs (Mozilla), “How does the Internet work?,” https://developer.mozilla.org/en-US/docs/Learn_web_development/Howto/Web_mechanics/How_does_the_Internet_work,

accessed 6 October 2026. Explains routers and IP addresses (compared to street addresses); notes that a home router often combines a switch, a router, and a modem; says the Internet is designed so that “whatever happens, they find a way to stay connected”; and distinguishes the Internet (the infrastructure) from the Web (a service built on top of it).

52. MDN Web Docs (Mozilla), “How the web works,” https://developer.mozilla.org/en-US/docs/Learn_web_development/Getting_started/Web_standards/How_the_web_works, accessed 6 October 2026. Describes a browser requesting files from a server; DNS as “like an address book for websites”; data sent in “small chunks called packets,” each with a header carrying addresses and its place in the sequence; packets taking different routes and arriving out of order, with missing packets requested again; and HTTPS.

53. 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 6 October 2026. The button example in Chapter 5 was run in a current desktop browser to confirm that clicking fills in the paragraph.

54. CodeAI (formerly Code.org), “Computer Science Discoveries,” CodeAI, 2026, <https://code.org/en-US/curriculum/computer-science-discoveries>, accessed 6 October 2026. A free curriculum listed for grades 6–10, including web design, that has students “building their own websites, apps, animations, games, and physical computing systems.”

55. CodeAI (formerly Code.org), “Artificial Intelligence Discoveries,” CodeAI, 2026, <https://code.org/en-US/curriculum/artificial-intelligence-discoveries>, and “CodeAI Launches AI Discoveries & AI Foundations,” <https://code.org/en-US/codeai-launches-ai-foundations-and-ai-discoveries>, both accessed 6 October 2026. The course page lists Web Lab among its tools, says its Web Development unit uses “a new, AI-assisted version of Web Lab,” and states that “Web Lab and AI tools are built for students ages 11 and up”; the launch page says “a free CodeAI account is required to access.” This book’s rule (AI helpers off or parent-run at 11–12) is stricter than the course’s own description.

56. CS Unplugged, “Squeezing pictures into less space,” <https://www.csunplugged.org/en/topics/image-representation/squeezing-pictures-into-less-space/>, accessed 6 October 2026. Explains run-length encoding with each row starting with white (a row beginning with black starts with 0); notes that fax machines “relied heavily on run-length encoding”; and explains that the simple version works well for black-and-white images but not photographs, while JPEG finds patterns in photos and then compresses them using RLE.

57. CS50, Harvard University, “How Computers Work,” CS50 AP curriculum reference, https://cs50.harvard.edu/ap/2021/curriculum/technology/references/how_computers_work.pdf, accessed 6 October 2026. Describes the CPU as “responsible for performing calculations and following instructions”; RAM as “short-term memory that software can use to store data quickly and temporarily”; drives as storing data “more permanently”; and the operating system as “the computer’s manager; it’s in charge of translating input from your keyboard and mouse, displaying information on the screen, and moving things around in memory,” deciding “how to delegate hardware resources for different software applications.”

58. Stanford Institute for Human-Centered Artificial Intelligence (HAI), “What is a Graphics Processing Unit (GPU)?,” AI definitions, <https://hai.stanford.edu/ai-definitions/what-is-a-gpu>, accessed 6 October 2026. Defines a graphics processor as a chip “originally designed to rapidly render graphics and images by performing many mathematical calculations simultaneously in parallel,” containing “thousands of smaller cores,” which makes it suited to the calculations used in AI.
59. International Energy Agency (IEA), *Energy and AI*, executive summary, IEA, April 2025, <https://www.iea.org/reports/energy-and-ai/executive-summary>, accessed 6 October 2026: “Data centres accounted for around 1.5% of the world’s electricity consumption in 2024, or 415 terawatt-hours (TWh),” and consumption “is set to more than double to around 945 TWh by 2030”; “AI-focused data centres can draw as much electricity as power-intensive factories such as aluminium smelters.” The IEA describes data centers’ global share as small but their local effects as “far more pronounced.” These are estimates and projections that change from year to year; the book uses them only to make the idea concrete.
60. MDN Web Docs (Mozilla), “Public-key cryptography,” Glossary, https://developer.mozilla.org/en-US/docs/Glossary/Public-key_cryptography, accessed 6 October 2026. “Keys come in pairs”; one is kept private and one is made public; “Anyone can encrypt a message to the owner of the private key, but only the owner of the private key can decrypt it”; and “it’s common for an encryption system to use a symmetric algorithm to encrypt the messages, then a public-key system to encrypt the symmetric key.”
61. MDN Web Docs (Mozilla), “HTTPS,” Glossary, <https://developer.mozilla.org/en-US/docs/Glossary/HTTPS>, accessed 6 October 2026. HTTPS “is an encrypted version of the HTTP protocol. It uses TLS to encrypt all communication between a client and a server.”
62. Federal Bureau of Investigation, Internet Crime Complaint Center (IC3), “Cyber Actors Exploit ‘Secure’ Websites in Phishing Campaigns,” Public Service Announcement I-061019-PSA, 10 June 2019, <https://www.ic3.gov/PSA/2019/PSA190610>, accessed 6 October 2026. Notes that https and the lock icon “are supposed to indicate the web traffic is encrypted,” that criminals use them on phishing sites, and advises: “Do not trust a website just because it has a lock icon or ‘https’ in the browser address bar.”
63. Micro:bit Educational Foundation, “Features,” <https://microbit.org/get-started/features/overview/>, accessed 6 October 2026. Lists buttons A and B; 25 LEDs in a 5x5 grid that also sense light; a temperature sensor in the processor; accelerometer; compass; radio and Bluetooth; a battery socket for two AAA batteries; and, on V2, a touch logo, microphone, and speaker.
64. Micro:bit Educational Foundation, “Python Editor” user guide, <https://microbit.org/get-started/user-guide/python-editor/>, and “New micro:bit Python Editor,” <https://microbit.org/new-microbit-python-editor/>, both accessed 6 October 2026; editor at <https://python.microbit.org/>. The guide says the editor supports “students aged 11-14,” is “totally free of charge and no logins or passwords are needed,” and keeps working offline once loaded. Its simulator supports “buttons, gestures, light levels, temperature, compass, sound, radio, serial pins and data logging,” so students can work on projects without a board; it shows a V2 board, so some code may not work on a V1.
65. Micro:bit MicroPython documentation, “Buttons” tutorial, <https://microbit-micropython.re>

adthedocs.io/en/v2-docs/tutorials/buttons.html; “microbit” module reference, <https://microbit-micropython.readthedocs.io/en/v2-docs/microbit.html>; and “random” module, <https://microbit-micropython.readthedocs.io/en/v2-docs/random.html>; all accessed 6 October 2026. Documents `button_a.is_pressed()`, `display.scroll()`, `display.show(Image.HAPPY)`, `sleep()` in milliseconds, `running_time()`, `temperature()` in degrees Celsius, and `random.randint()`.

66. Micro:bit MicroPython documentation, “Data Logging V2,” <https://microbit-micropython.readthedocs.io/en/v2-docs/log.html>, accessed 6 October 2026. The log module saves data to a MY_DATA file on a micro:bit V2 that “can be viewed and plotted with a browser”; documents `log.set_labels()` and `log.add()`, with an example logging `temperature()` and `display.read_light_level()`.

67. Cybersecurity and Infrastructure Security Agency (CISA), “Secure Our World,” CISA, 2026, <https://www.cisa.gov/secure-our-world>, accessed 6 October 2026. The four easy ways: “Recognize and Report Phishing”; “Stop Online Crime with Strong Passwords”; “Make Your Accounts Safer with Multifactor Authentication (MFA)”; “Update Software for Safety.” The page adds that “using strong passwords and a password manager are some easy ways to protect ourselves.”

68. Federal Trade Commission, “How To Recognize and Avoid Phishing Scams,” FTC Consumer Advice, <https://consumer.ftc.gov/articles/how-recognize-avoid-phishing-scams>, accessed 6 October 2026. The sample scam email shows a generic greeting, an account “on hold because of a billing problem,” and an invitation to click a link to update payment details. Multi-factor credentials “fall into three categories: something you know ... something you have ... something you are.” Phishing can be reported at ReportFraud.ftc.gov; phishing texts can be forwarded to SPAM (7726).

69. National Institute of Standards and Technology (NIST), *Digital Identity Guidelines: Authentication and Authenticator Management*, NIST Special Publication 800-63B-4, 2025, <https://pages.nist.gov/800-63-4/sp800-63b.html>, accessed 6 October 2026. Verifiers “SHALL require passwords that are used as a single-factor authentication mechanism to be a minimum of 15 characters in length”; “SHALL NOT impose other composition rules (e.g., requiring mixtures of different character types)”; and “SHALL NOT require subscribers to change passwords periodically,” though they must force a change if there is evidence of compromise. NIST writes rules for services and agencies, not families; this book calls them “the federal password guidelines” and translates them into household habits.

70. Federal Trade Commission, “Children’s Online Privacy Protection Rule,” final rule, *Federal Register* 90 FR 16918, FR Doc. 2025-05904, 22 April 2025, <https://www.federalregister.gov/documents/2025/04/22/2025-05904/childrens-online-privacy-protection-rule>, accessed 6 October 2026. The amended rule took effect 23 June 2025; with listed exceptions, regulated companies had until 22 April 2026 to comply.

71. Office of the U.S. Surgeon General, *Social Media and Youth Mental Health: The U.S. Surgeon General’s Advisory*, U.S. Department of Health and Human Services, 2023, <https://www.hhs.gov/surgeongeneral/reports-and-publications/youth-mental-health/social-media/index.html>, accessed 6 October 2026. The advisory page states that “we cannot conclude social media is sufficiently safe for children and adolescents.” The advisory’s full text notes: “Although age 13 is commonly the required minimum age used by social media platforms in the U.S., nearly 40% of children ages

8–12 use social media.” An advisory summarizes evidence and suggests actions; it is not a law. A January 2023 television remark by the Surgeon General about age 13 was an interview, not part of the advisory, and is not used here.

72. American Academy of Pediatrics, Center of Excellence on Social Media and Youth Mental Health, “Appropriate Age to Introduce a Mobile Device,” Q&A Portal, 2026, <https://www.aap.org/en/patient-care/media-and-children/center-of-excellence-on-social-media-and-youth-mental-health/qa-portal/qa-portal-library/qa-portal-library-questions/appropriate-age-to-introduce-a-mobile-device/>, accessed 6 October 2026. The page says research “doesn’t point to one perfect age” and points families to the Phone Ready quiz, whose signs of readiness include: “Is your child responsible about things like homework? Do they say sorry if they hurt a friend’s feelings? How well do they follow other tech rules? Are you as a parent ready to help support them?”

73. Michelle Faverio and Olivia Sidoti, “Teens, Social Media and AI Chatbots 2025,” Pew Research Center, 9 December 2025, <https://www.pewresearch.org/internet/2025/12/09/teens-social-media-and-ai-chatbots-2025/>, accessed 6 October 2026. An online survey of 1,458 U.S. teens ages 13 to 17, conducted 25 September to 9 October 2025. “Roughly two-thirds of teens (64%) say they ever use an AI chatbot,” including about three in ten who do so daily; among 13- and 14-year-olds, 57 percent use chatbots. Of YouTube: “Roughly three-quarters of teens say they use it every day”; “three-quarters of 15- to 17-year-olds say they use Instagram, compared with 44% of 13- to 14-year-olds.” The survey did not include children under 13. Pew is a nonpartisan research organization that takes no policy positions.

74. Federal Bureau of Investigation, “The Financially Motivated Sextortion Threat,” FBI News, 16 January 2024, <https://www.fbi.gov/news/stories/the-financially-motivated-sex-tortion-threat>, accessed 6 October 2026; see also FBI, “Sextortion,” <https://www.fbi.gov/how-we-can-help-you/scams-and-safety/common-scams-and-crimes/sex-tortion>. The FBI describes victims as “mostly boys” and says offenders “typically pretend to be alluring young girls” to befriend teens, often on social media and gaming platforms. Its advice to anyone targeted: “Understand you are not at fault and you are not alone”; “Ask for help from a trusted adult or law enforcement before sending money or more images. Cooperating with the predator rarely stops the blackmail and harassment”; report the account through the platform and block it; “Save all interactions”; and report to the FBI at tips.fbi.gov or 1-800-CALL-FBI (1-800-225-5324), or to local law enforcement.

75. National Center for Missing & Exploited Children (NCMEC), “Take It Down,” 2026, <https://takeitdown.ncmec.org/>, accessed 6 October 2026: “Take It Down is a free service that can help you remove or stop the online sharing of nude, partially nude, or sexually explicit images or videos taken of you when you were under 18 years old. You can remain anonymous while using the service and you won’t have to send your images or videos to anyone.” It works on “public or unencrypted online platforms that have agreed to participate.” NCMEC’s CyberTipline (<https://report.cybertip.org/>) takes reports of online exploitation.

76. College Board, “Guidance for Artificial Intelligence Tools and Other Services,” AP Central, 2026, <https://apcentral.collegeboard.org/exam-administration-ordering-scores/administering-exams/exam-policies/artificial-intelligence-tools>, accessed 6 October 2026. On AP Computer Science Principles: “Students should be aware that generative AI tools can produce incomplete code, code that creates or introduces biases, code with errors, inefficiencies in how the

code executes, or code complexities that make it difficult to understand and therefore explain the code.”

77. CodeAI (formerly Code.org), “Hour of AI,” CodeAI, 2026, <https://code.org/en-US/hour-of-ai>, accessed 6 October 2026. AI for Oceans is listed for grades 3–12: “you train a real machine learning model to identify sea creatures and trash in the ocean.” At 11–12 this book has the parent run it on the parent’s device.

78. Five state examples, read on official pages and accessed 6 October 2026. Laws change; read your own state’s current rules. This is not legal advice. New York, 8 NYCRR § 100.10 (Home instruction), via Cornell Legal Information Institute, <https://www.law.cornell.edu/regulations/new-york/8-NYCRR-100.10> (for grades seven and eight: “English (two units); history and geography (two units); science (two units); mathematics (two units); physical education (on a regular basis); health education (on a regular basis); art (one-half unit); music (one-half unit); practical arts (on a regular basis); and library skills (on a regular basis)”). Pennsylvania, 24 P.S. § 13-1327.1 (Home education program), <https://www.legis.state.pa.us/WU01/LI/LI/US/HTM/1949/0/0014.013.000.027.001..HTM> (lists required elementary and secondary courses; the portfolio “shall consist of a log, made contemporaneously with the instruction,” samples of the student’s work, and, in grades three, five, and eight, standardized test results in reading/language arts and mathematics). Texas Education Agency, “Home Schooling,” <https://tea.texas.gov/families-and-students/finding-school-your-child/home-schooling> (parents “are required to follow a course of study that includes good citizenship”; under *Leeper*, the written curriculum covers reading, spelling, grammar, math, and good citizenship). North Carolina Division of Non-Public Education, “Requirements and Recommendations,” <https://www.doa.nc.gov/divisions/non-public-education/home-schools/requirements-recommendations> (an annual nationally standardized achievement test that “must measure English grammar, reading, spelling, and mathematics”). Virginia, Va. Code § 22.1-254.1, <https://law.lis.virginia.gov/vacode/title22.1/chapter14/section22.1-254.1/> (a “description of the curriculum, limited to a list of subjects to be studied during the coming year”). None of the five names computer science or computing among required subjects for these grades.

79. HSLDA, “Homeschool Laws by State,” <https://hsllda.org/legal>, accessed 6 October 2026. Collects state summaries and advises that “You should follow the law of the state in which you are physically present.” HSLDA is an advocacy organization; use its pages as a starting point and read your state’s official text.

80. 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 6 October 2026. Among homeschooled students with a grade equivalent of 6th to 8th 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, and it is from 2016.

81. HSLDA, “3 Ways to Determine My Teen’s High School Course Credit,” HSLDA, 15 January 2020, <https://hsllda.org/post/3-ways-to-determine-my-teens-high-school-course-credit>, accessed 6 October 2026: “For a core course (English, science, history, math, or foreign language), you will want your teen to log at least 150 hours for 1.0 credit (roughly five hours a week for 30 weeks)”;

for an elective course, “log 120+ hours for 1.0 credit and 60+ hours for 0.5 credit”; a lab science, 180 hours. These are guidelines from an advocacy organization, not law; states and schools set their own rules for high-school credit.

82. Bebras USA, “Bebras Computing Challenge,” <https://bebraschallenge.org/>, accessed 6 October 2026. “Typically, Bebras USA is held in November”; the 2026 U.S. challenge was scheduled for 9–20 November. It is free; students have “45 minutes to complete as many tasks as they can”; age groups include Benjamins (10–12), Cadets (12–14), Juniors (14–16), and Seniors (16–18). A coordinator registers students; check the site for how a family can take part, and check the current dates and rules.

83. Python Software Foundation, “Python Documentation,” <https://www.python.org/doc/> and <https://docs.python.org/3/>, accessed 6 October 2026. Python and its documentation, including “The Python Tutorial,” are free.

84. Scratch Team, “Under 16,” Scratch Help Center, MIT, <https://mitscratch.freshdesk.com/en/support/solutions/articles/4000228296-under-16>, accessed 6 October 2026: “anyone under 16 who registers a new Scratch account will need permission from a parent or guardian”; unverified accounts “can still make and save Scratch projects, but they cannot comment, share, or participate in the online community.” Scratch itself is at <https://scratch.mit.edu/>.

85. Arduino Help Center, “About Arduino accounts for minors,” <https://support.arduino.cc/hc/en-us/articles/4839080453148-About-Arduino-accounts-for-minors>, accessed 6 October 2026. Arduino offers “Junior accounts” for users below 14 and “Restricted accounts” for ages 14 to 17; both must be linked to a parent. “A junior account can be created by a parent, or at the request of the minor (with parental consent),” and a parent must approve a restricted account. Check the current rules before creating an account.

86. American Computer Science League, “Divisions,” <https://www.acsl.org/get-started/divisions>, and “Registration,” <https://www.acsl.org/Registration>, both accessed 6 October 2026. Junior Division: “Best for junior high and middle school students who are learning programming or who have coding experience. No student beyond grade 9 may compete in the Junior Division.” Each contest combines a 30-minute short-answer test with an online programming problem to solve in 72 hours. For the 2026–27 season, “The deadline for registering for the season is December 31, 2026.” Advisors register teams; fees apply and are not listed here.

87. Air & Space Forces Association, *CyberPatriot XIX Rules Book*, 2 September 2026, https://www.uscyberpatriot.org/app/uploads/2026/09/CP19_RulesBook_Final_-_2-Sep-2026.pdf, accessed 6 October 2026: “All middle school and junior high-level students may compete on a Middle School Division team, provided that team is fielded by one of the approved organization types,” a list that includes “Homeschool*”; section 5, “Homeschools,” sets verification steps (for an independent homeschool, a signed Parent/Guardian Affidavit, or registering as an “other organization” with a local library or similar group). “A CyberPatriot team consists of two to six competitors.” For the 2026–27 season, team registration was extended to 8 October 2026. Check with the program at <https://www.uscyberpatriot.org/> before planning.

88. Congressional App Challenge, “Rules,” <https://www.congressionalappchallenge.us/students/rules/>, accessed 6 October 2026. Students “must be a middle or high school student at the time of

app submission”; teams may have up to four students; students compete in the district where they live or attend school; “All competing students must be U.S. residents at the time of submission.” The 2026 submission deadline was 26 October.

89. Carnegie Mellon University, “picoCTF.org is now CyLab Security Academy,” <https://picoctf.org/>, accessed 6 October 2026: “CyLab Security Academy is Carnegie Mellon University’s free cybersecurity learning platform”; logins “created before May 8, 2026” still work. Check the current terms for your student’s age.

90. USA Computing Olympiad, “USACO,” <https://usaco.org/>, accessed 6 October 2026. Its mission is “identifying, motivating, and training high-school computing students at all levels”; it offers free “On-line programming contests (four per year) for students at all levels.”

91. OECD and European Commission, *Empowering Learners for the Age of AI: An AI Literacy Framework for Primary and Secondary Education*, OECD Publishing, Paris, 18 June 2026, <https://doi.org/10.1787/65cd27d4-en>, accessed 6 October 2026. From the publication’s abstract: “AI literacy, as a set of knowledge, skills and attitudes, equips learners to understand how AI systems work, critically evaluate their outputs and use them ethically and creatively.”

92. American Academy of Pediatrics, “Family Media Plan,” [HealthyChildren.org](https://www.healthychildren.org/English/fmp/Pages/MediaPlan.aspx), <https://www.healthychildren.org/English/fmp/Pages/MediaPlan.aspx>, accessed 6 October 2026. A free online tool for writing down a family’s own choices about screens, places, and times.

93. Jenny Radesky, “Your Child’s First Phone: Are They Ready?,” [HealthyChildren.org](https://www.healthychildren.org/English/family-life/Media/Pages/Cell-Phones-Whats-the-Right-Age-to-Start.aspx), American Academy of Pediatrics, 3 June 2026, <https://www.healthychildren.org/English/family-life/Media/Pages/Cell-Phones-Whats-the-Right-Age-to-Start.aspx>, accessed 6 October 2026.

94. College Board, “About AP Computer Science,” AP Central, <https://apcentral.collegeboard.org/courses/about-ap-computer-science>, accessed 6 October 2026: the AP computer science courses “Require familiarity with concepts from Algebra 1. Do not require any computer science or coding experience.”

95. U.S. Census Bureau, “Computer and Internet Use in the United States: 2021,” press release CB24-TPS.61, 18 June 2024, <https://www.census.gov/newsroom/press-releases/2024/computer-internet-use-2021.html>, accessed 6 October 2026: “Most U.S. households had at least one type of computer (95%) and had a broadband internet subscription (90%) in 2021.” The Census counts smartphones as a type of computer.