INSTRUCTOR GUIDE

Underneath the Syntax

A Primer on Programming Fundamentals

By Edison Mooers

Instructor guide for introductory programming and code literacy

High school · college · workforce training · QA and technical-adjacent courses

Companion to the free interactive resource at gibiddapress.com/syntax/educators

About This Guide

This guide is written for instructors introducing programming concepts at the high school, introductory college, or workforce-training level. It follows the structure of Underneath the Syntax chapter by chapter, and for each chapter provides a learning objective, key vocabulary, discussion questions, a time-boxed classroom activity, and, where useful, an instructor note flagging a common misconception or a language-specific caveat.

The book itself is deliberately language-neutral. All of its examples use plain, readable pseudocode rather than the syntax of any single language, so students build a mental model of how programming works before they commit to Python, Java, JavaScript, C#, or anything else. That design choice is also what makes the book usable across very different courses: an AP Computer Science Principles class, a CS0 or intro-to-programming college course, a bootcamp-prep workshop, or even a course for testers and QA staff who need to reason about code without necessarily writing it themselves.

Nothing in this guide requires students to have a computer in front of them. Every suggested activity can run on paper or on a whiteboard within the time given, which makes the book usable as reading homework paired with discussion, even in a classroom without daily lab access. Where a class does have lab time, the free companion described below gives students runnable practice.

Course-Level Learning Outcomes

By the end of a course built around this book, students should be able to:

- Trace the execution of a short program, predicting its output or final state.
- Select an appropriate control structure or collection for a given problem.
- Explain how data, memory, scope, and references interact underneath a program's surface.
- Identify common logic errors before running a program.
- Translate language-neutral concepts into the syntax of a specific programming language.
- Evaluate the structure and behavior of AI-generated code, including its assumptions and gaps.

Who This Book Fits

- High school introductory computer science, as a first read before or alongside a language-specific course (e.g., before Python or Java in an intro unit, or as a supplement to AP Computer Science Principles).
- College CS0 / "intro to programming" courses, particularly for students arriving without prior exposure, or as a leveling text in the first two weeks of a CS1 course.
- Coding bootcamp pre-work and short workforce-training programs, to build durable vocabulary before students are asked to move quickly through one specific language and framework.
- QA, testing, and technical-adjacent courses, since the book explicitly addresses readers who need to reason about code and evaluate whether it behaves correctly without necessarily writing it themselves.

Chapter Tiers

With 26 chapters, the full book is a real commitment. Each chapter below is labeled Core, Recommended, or Extension so instructors can scope a course to the time they actually have:

Core: essential to any course built on this book; the load-bearing material everything else depends on. Chapters 1–11 and 25–26.

Recommended: valuable in most courses and worth including when time allows, but reasonable to trim under real time pressure. Chapters 12–16 and 20–24.

Extension: genuinely useful depth for students continuing toward software engineering, but not necessary for AP CSP, a brief workforce program, or a non-programmer code-literacy course. Chapters 17–19 (pointers, closures, generics).

Suggested Pacing

A few pacing options, from a full semester down to a short workforce workshop:

Full semester (14–16 weeks)

One chapter per class session, roughly two chapters per week, leaving the final one to two weeks for the capstone project described below. Part I (Chapters 1–11) fits comfortably in the first five to six weeks; Part II (Chapters 12–24) is the deepest stretch and benefits from the most time, particularly Chapter 13 on value vs. reference types, which is the most commonly misunderstood chapter; Part III (Chapters 25–26) wraps the course.

Condensed module (6–8 weeks): sample 8-week map

This map covers Part I in full plus the highest-value Recommended chapters (12, 13, 15, 20, 24), the new AI-code-literacy unit below, and the Part III capstone. It still gives students a coherent foundation without the full 26-chapter arc.

Week	Chapters	Focus	Assessment
1	1–3	History, what a program is, I/O	Reading check
2	4–6	Variables, operators, conditionals	Reading check
3	7–9	Loops, collections, strings	Concept quiz 1
4	10–11	Functions, error handling	Concept quiz 1 (cont.)
5	12–13	Binary/representation, value vs. reference	Trace-and-predict exercise
6	15, 20	Recursion, objects & classes	Concept quiz 2
7	24 + AI-Code-Literacy	Algorithmic thinking, evaluating AI-generated	AI-code review checklist

Week	Chapters	Focus	Assessment
	unit	code	
8	25–26	Capstone build and presentations	Capstone rubric

Self-paced or flipped classroom

Assign chapters as reading homework using the discussion questions below as a reading check, and use class time entirely for the suggested activities, the free companion exercises, and the AI-code-literacy activities below.

Using the Free Companion in the Classroom

Underneath the Syntax: Try It in QF Code is a free companion available at gibiddapress.com/books/underneath-the-syntax/companion/ Where the book is deliberately language-neutral, the companion gives students runnable, guided exercises in QF Code, a browser-based teaching language built for this purpose. For any class with lab access, the natural rhythm is to read a chapter, discuss it using the questions below, then move to the corresponding companion exercises for hands-on practice before advancing.

PART I

The Basics

Chapters 1–11 — what a program actually is, and the tools every program is built from.

Chapter 1: A Brief History of Programming [Core]

Book pages: 17–20

Learning objective: Explain why programming languages evolved the way they did, from machine code to today's multi-paradigm languages, and why that history still shapes how languages are designed.

Key terms

assembly language · punch cards · batch processing · high-level language · structured programming · object-oriented programming

Discussion questions

- Why did assembly language count as a genuine abstraction, even though it still required extremely detailed instructions?
- What problem was structured programming trying to solve, and do you see that same problem in any code you've looked at?

Suggested activity

Have students build a one-page timeline placing five languages they've heard of (e.g., Python, JavaScript, C, COBOL, Java) onto the eras described in the chapter, and justify each placement in one sentence.

Time: Prep 5 min · Activity 20 min · Discussion 10 min · **Materials:** paper, a printed list of 8–10 language names to choose from

Common misconception to watch for: *Students often assume older means worse. Emphasize that COBOL and FORTRAN are still running critical systems today; the chapter is about context, not a hierarchy of progress.*

Chapter 2: What a Program Actually Is [Core]

Book pages: 21–24

Learning objective: Describe a program as an ordered sequence of instructions operating on data, and explain why a computer must be told exactly what to do.

Key terms

instruction · sequential execution · literal execution

Discussion questions

- Why does the book compare a computer to a 'very literal-minded assistant'? Where has an overly literal instruction caused a real problem you've seen (in software or otherwise)?
- How is a recipe a useful, if imperfect, model for a program? Where does the comparison break down?

Suggested activity

In pairs, have students write step-by-step 'literal-minded assistant' instructions for a simple task (making a sandwich, tying a shoe) and swap with another pair to follow exactly as written, no assumed steps allowed.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

---

Chapter 3: Input & Output Basics [Core]

Book pages: 25–28

Learning objective: Distinguish output from input and explain why this book introduces them before any other concept.

Key terms

output · input · console

Discussion questions

- Why does the book introduce input and output before variables, types, or logic?
- What is the 'console' standing in for, and where else might input and output happen beyond it?

Suggested activity

Ask students to diagram (arrows and boxes) the flow of input and output for something familiar, like an ATM or a vending machine, without writing any code.

Time: Prep 5 min · Activity 10 min · Discussion 10 min · **Materials:** paper

---

Chapter 4: Variables & Types [Core]

Book pages: 29–32

Learning objective: Explain what a variable is, why types exist, and the difference between a variable's name, its value, and its type.

Key terms

variable · type · constant · mutability · type conversion · casting

Discussion questions

- Why does the book insist 'variable' is the right word, rather than something like 'box' or 'label'?
- What is the practical difference between a constant and a variable, and why would a programmer deliberately choose a constant?

Suggested activity

Give students five real-world values (a name, an age, a price, a yes/no answer, a list of favorite foods) and have them assign a type to each and justify it.

Time: Prep 5 min · Activity 10 min · Discussion 10 min · **Materials:** paper

Common misconception to watch for: *Learners often conflate a variable's name with its value. Reinforce that the name is just a label pointing at a value that can change.*

Chapter 5: Operators & Expressions [Core]

Book pages: 33–36

Learning objective: Use arithmetic, comparison, and boolean operators to build expressions, and evaluate nested expressions correctly.

Key terms

arithmetic operator · comparison operator · boolean operator · expression

Discussion questions

- What is the difference between an operator and an expression?
- Why do comparison operators always produce a boolean result, no matter what they compare?

Suggested activity

Give students a nested expression on the board and have them evaluate it step by step, innermost operation first, narrating each step out loud.

Time: Prep 5 min · Activity 10 min · Discussion 5 min · **Materials:** whiteboard

---

Chapter 6: Conditional Logic [Core]

Book pages: 37–40

Learning objective: Use if, else, and else-if to let a program choose between outcomes, and recognize when nested conditionals become hard to follow.

Key terms

if statement · else · else if · nesting

Discussion questions

- What real-world decision (outside of code) could be modeled with if / else if / else? Walk through the branches.
- At what point does nesting conditionals start to hurt readability, and what does the book suggest instead?

Suggested activity

Have students convert a short written policy (e.g., a library's late-fee rule, a theme park's height requirement) into a chain of conditionals on paper.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper, a printed sample policy

---

Chapter 7: Loops & Iteration [Core]

Book pages: 41–44

Learning objective: Distinguish while loops from for loops, and identify how a loop can accidentally become infinite.

Key terms

while loop · for loop · iteration · infinite loop

Discussion questions

- When would you reach for a while loop instead of a for loop, and vice versa?
- What are the ingredients of an infinite loop, and how would you notice one was happening?

Suggested activity

Give students a for loop and a while loop that do the same thing and ask them to identify what has to be true for both to produce identical output.

Time: Prep 5 min · Activity 10 min · Discussion 10 min · **Materials:** paper

Common misconception to watch for: *New learners often trust that a loop will simply 'know' when to stop. Stress that the stopping condition must be created and maintained deliberately.*

Instructor note: *For loops fit naturally when the number of iterations is known ahead of time (a fixed count, or looping over a collection). While loops fit when the stopping condition depends on something not known in advance (waiting for input, polling until a state changes). Many languages let either loop do the other's job, so accept any answer backed by sound reasoning rather than one fixed rule.*

Chapter 8: Arrays & Collections [Core]

Book pages: 45–50

Learning objective: Use arrays, maps/dictionaries, and sets appropriately, and choose the right collection type for a given problem.

Key terms

array · index · map/dictionary · key-value pair · set

Discussion questions

- Why does an array's index typically start at 0 rather than 1, and why does that trip up beginners?
- Given a phone contact list, why is a map a better fit than a plain array?

Suggested activity

Present three small data problems (a class roster, a phone book, a list of unique tags on a blog post) and have students choose array, map, or set for each, and defend the choice.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

---

Chapter 9: Strings, Deep Dive [Core]

Book pages: 51–54

Learning objective: Treat a string as a sequence of characters, index and combine strings, and explain why strings are immutable in most languages.

Key terms

string · immutability · concatenation · interpolation

Discussion questions

- What does it mean, practically, for a string to be immutable? What actually happens when you 'change' one character of a string?
- Why might a language's designers choose to make strings immutable rather than allow in-place edits?

Suggested activity

Have students trace, line by line, what a short sequence of string operations (slicing, concatenation, interpolation) produces, without running any code.

Time: Prep 5 min · Activity 10 min · Discussion 5 min · **Materials:** paper

---

Chapter 10: Functions [Core]

Book pages: 55–58

Learning objective: Package instructions into reusable functions with parameters, arguments, and return values, and explain why functions matter beyond avoiding repetition.

Key terms

function · parameter · argument · return value · abstraction

Discussion questions

- What is the difference between a parameter and an argument?
- Beyond avoiding copy-and-paste, what does the book say functions actually give a program?

Suggested activity

Have students name, describe the inputs/outputs of, and write a one-line description for three functions they'd need to build a simple grade calculator, without writing any code inside them yet.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

---

Chapter 11: Error Handling [Core]

Book pages: 59–62

Learning objective: Explain what happens when errors go unhandled, use try/catch-style exception handling, and identify when cleanup code (finally) is needed.

Key terms

exception · try/catch · finally · graceful failure

Discussion questions

- Why does the book argue that not every error should be caught? Give an example of an error that probably should crash the program.
- What kinds of cleanup work belong in a 'finally' block, and why does that code need to run no matter what happened?

Suggested activity

Give students a short scenario (a program reading a file that might not exist) and have them write, in plain English, what should happen in the try, catch, and finally parts.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

Instructor note: *A reasonable example of an error that probably shouldn't be caught: a null reference or an out-of-bounds index caused by a genuine bug in the program's own logic. Silently catching that kind of error hides a defect that needs to be fixed, not handled at runtime. As a rule of thumb, errors worth catching tend to come from the outside world (a missing file, bad user input, a failed network call); errors caused by the program's own logic being wrong are usually better left to crash loudly during development.*

PART II

Going Deeper

Chapters 12–24 — how those fundamentals work underneath the surface.

Chapter 12: Under the Hood: Binary & Representation [Recommended]

Book pages: 65–68

Learning objective: Explain that all data is ultimately binary, convert simple values between representations, and describe what bitwise operators actually do.

Key terms

bit · byte · two's complement · bitwise operator

Discussion questions

- Why do computers use two's complement to represent negative numbers instead of just adding a minus sign in front of a binary number?
- Name one practical use of a bitwise operator mentioned in the chapter.

Suggested activity

Use 8-bit examples throughout for consistency. Work through +5 and -5 in 8-bit two's complement together as a class on the board first (00000101, then invert and add 1 to get -5). Only after that walkthrough, have students convert 2–3 additional small positive and negative integers independently, checking their work against an online converter. For non-technical or younger students, drop the two's-complement portion entirely and keep the activity to converting positive integers only; that alone satisfies the chapter's core goal.

Time: Prep 10 min (worked example) · Activity 20 min · Discussion 10 min · **Materials:** paper, board for the worked example, optional online binary converter

Instructor note: *Two's complement is genuinely difficult for true beginners the first time through. Don't skip the shared worked example, and don't hesitate to use the simplified positive-only version for a non-technical or high-school audience. On the discussion question: two's complement lets the same addition circuitry handle both addition and subtraction without special-casing the sign, and it gives zero a single representation instead of two (a real problem in the simpler 'sign bit' schemes it replaced).*

Chapter 13: Value Types, Reference Types & Typing Systems [Recommended]

Book pages: 69–74

Learning objective: Distinguish value types from reference types, and compare static vs. dynamic typing and strong vs. weak typing across languages.

Key terms

value type · reference type · boxing/unboxing · static typing · dynamic typing · null

Discussion questions

- If you copy a value type and then change the copy, what happens to the original? What about a reference type?
- Why does the book call null 'the billion dollar mistake'? Do you agree with the framing?

Suggested activity

Give students two short code snippets side by side, one editing a value type, one editing a reference type, and have them predict and explain the difference in outcome.

Time: Prep 5 min · Activity 15 min · Discussion 15 min · **Materials:** printed handout with the two code snippets

Common misconception to watch for: *This is one of the most common sources of confusion in the book. Consider spending extra time here with a physical demonstration (index cards for values, sticky notes with arrows for references).*

Instructor note: *This chapter teaches a conceptual model, not one language's exact rules. Value/reference behavior differs in the specifics from language to language (e.g., Python's treatment of small integers and strings, Java's primitives vs. objects). Once your course moves to a specific language, connect this chapter's model to that language's actual semantics explicitly, rather than assuming it transfers automatically.*

Chapter 14: Scope & Lifetime [Recommended]

Book pages: 75–78

Learning objective: Explain local, global, and block scope, and describe the difference between the stack and the heap at a conceptual level.

Key terms

local scope · global scope · block scope · stack · heap

Discussion questions

- Why do most style guides discourage heavy use of global variables?
- Why does understanding the stack and heap matter, even for a beginner who will never manage memory manually?

Suggested activity

Have students trace a short function with a local variable and a global variable and identify, at each line, what's visible and what isn't.

Time: Prep 5 min · Activity 10 min · Discussion 10 min · **Materials:** paper

---

Chapter 15: Recursion [Recommended]

Book pages: 79–82

Learning objective: Identify the base case and recursive case in a recursive function, and compare recursion to an equivalent loop.

Key terms

recursion · base case · recursive case · call stack

Discussion questions

- What happens if a recursive function is missing a base case?
- For the factorial example, why might a loop be a more practical choice in production code, even though the recursive version is elegant?

Suggested activity

Have students hand-trace the call stack for a small recursive factorial call (e.g., factorial(4)), writing out each call and its return value.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

Instructor note: *Recursion vs. loops is a genuine tradeoff, not a right-or-wrong pick. Recursion can read more naturally for problems that are inherently recursive (tree traversal, divide-and-conquer), but each call consumes stack space, so very deep recursion can overflow the stack in a way an equivalent loop wouldn't. Some languages optimize certain recursive calls to avoid this (tail-call optimization); many common teaching languages, including Python and JavaScript, do not, so treat that optimization as language-dependent rather than guaranteed.*

Chapter 16: Maps, Sets & Nested Collections [Recommended]

Book pages: 83–86

Learning objective: Work with nested collections (a map of lists, a list of maps, etc.) and loop through them correctly.

Key terms

nested collection · map · set

Discussion questions

- What real data would naturally be modeled as a map where each value is itself a list?
- Why do nested loops over nested collections become harder to read quickly, and what does the book suggest to manage that?

Suggested activity

Give students a small nested structure (e.g., a map of students to a list of their grades) and have them write, in plain English, the steps to compute each student's average.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

---

Chapter 17: Pointers & References [Extension]

Book pages: 87–90

Learning objective: Explain what a pointer is, how it differs from a reference type, and why pointers matter even in languages that hide them.

Key terms

pointer · dereference · dangling pointer

Discussion questions

- The book says pointers matter 'even if a language hides them.' What does that mean for a language like Python or JavaScript?
- What is a dangling pointer, and why is it dangerous?

Suggested activity

Draw boxes-and-arrows diagrams as a class showing a pointer pointing to a value, then being reassigned, then the original value being freed.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** whiteboard

Instructor note: *Even in a language with no explicit pointers, passing a mutable object (a list, a dictionary) into a function still passes a reference to the same underlying data, not a copy. If the function mutates it, the caller sees the change. That's exactly what a pointer makes explicit; a garbage-collected language just manages the bookkeeping automatically instead of asking the programmer to.*

Chapter 18: Closures & Anonymous Functions [Extension]

Book pages: 91–94

Learning objective: Treat functions as values, write simple anonymous functions (lambdas), and explain what a closure remembers.

Key terms

anonymous function · lambda · closure

Discussion questions

- What does it mean to say 'functions are values' in a language that supports this? Where have you already seen this idea, even informally?

- What exactly does a closure 'close over,' and why does that matter for the function's later behavior?

Suggested activity

Give students a short closure example and ask them to predict what value it returns after the surrounding variable changes, then check the answer as a class.

Time: Prep 5 min · Activity 10 min · Discussion 10 min · **Materials:** printed handout with the closure example

---

Chapter 19: Generics & Templates [Extension]

Book pages: 95–98

Learning objective: Explain the problem generics solve, write a simple generic function conceptually, and describe what a constraint does.

Key terms

generic · type parameter · constraint

Discussion questions

- What problem is being solved by writing one generic function instead of several nearly-identical ones?
- Why not just skip type-checking entirely and let any type through? What would be lost?

Suggested activity

Have students identify two or three functions from earlier in the book (e.g., 'find the maximum value') that could be rewritten generically, and describe what a constraint on the type would need to guarantee.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

Instructor note: *Skipping type-checking entirely would let a function accept literally anything, including inputs that make no sense for it, with any resulting error surfacing later and further from its actual cause. Generics keep the safety of type-checking while still letting one function work across many types, which is the balance the chapter is pointing at.*

Chapter 20: Objects, Classes & Structures [Recommended]

Book pages: 99–102

Learning objective: Distinguish structs from classes, explain the relationship between a class and its instances, and describe encapsulation and enumerations.

Key terms

struct · class · instance · encapsulation · enum · inheritance

Discussion questions

- What is the difference between a class and an instance of that class?
- Why does encapsulation matter for keeping a program's data consistent and predictable?

Suggested activity

Have students design a simple class on paper (fields and a couple of behaviors) for something familiar, like a LibraryBook or a Playlist, and identify which fields should be encapsulated.

Time: Prep 5 min · Activity 20 min · Discussion 10 min · **Materials:** paper

---

Chapter 21: How Code Becomes Execution [Recommended]

Book pages: 103–106

Learning objective: Compare compiled, interpreted, and hybrid (bytecode/JIT) execution models, and describe manual memory management vs. garbage collection.

Key terms

compiler · interpreter · bytecode · JIT compilation · transpiler · garbage collection

Discussion questions

- What tradeoff does a compiled language accept in exchange for faster execution?
- Why do languages like Java and C# use an intermediate bytecode instead of compiling straight to machine code?

Suggested activity

Have students sort a list of languages (Python, C, Java, JavaScript, Rust, C#) into compiled, interpreted, and hybrid, and defend any language they're unsure about using the chapter's definitions.

Time: Prep 5 min · Activity 15 min · Discussion 15 min · **Materials:** printed list of the six languages

Instructor note: *Accept nuance in the sorting activity. Real implementations don't always sit neatly in one bucket, Python is typically called interpreted but CPython compiles to bytecode first, and JavaScript engines use JIT compilation. A language may be commonly described one way while a particular implementation uses bytecode, JIT compilation, transpilation, or several stages at once. Grade for sound reasoning, not for matching one official answer.*

Chapter 22: Writing Readable Code [Recommended]

Book pages: 107–110

Learning objective: Apply naming conventions, comment for 'why' rather than 'what,' and keep functions small and single-purpose.

Key terms

comment · naming convention · style guide · linter

Discussion questions

- Why does the book argue that comments should explain 'why,' not 'what'? Find an example of each kind.
- What's the practical cost of a function name like validateAndSaveAndNotify, according to the chapter?

Suggested activity

Give students a short, poorly-named and poorly-commented snippet (in plain pseudocode) and have them rename variables/functions and rewrite the comments to follow the chapter's guidance.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** printed handout with the snippet

---

Chapter 23: Modules & Imports [Recommended]

Book pages: 111–114

Learning objective: Explain what a module is, why code is split across files, and how namespaces prevent naming collisions.

Key terms

module · import · namespace · dependency

Discussion questions

- Why not just put all of a program's code in one giant file?
- What problem does a namespace solve when two modules happen to define something with the same name?

Suggested activity

Have students sketch a simple module map for a small app (e.g., a to-do list app split into a data module, a display module, and a main file) showing what imports from what.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** paper

---

Chapter 24: A Gentle Taste of Algorithmic Thinking [Recommended]

Book pages: 115–118

Learning objective: Explain what Big O notation communicates, and use binary search as a concrete example of O(log n) efficiency.

Key terms

Big O notation · binary search · O(log n)

Discussion questions

- What is Big O notation actually describing, and what is it deliberately not describing?
- Why is binary search so much faster than checking every item one at a time, once a list gets large?

Suggested activity

Have students play 'guess the number' (1–100) using only yes/no higher-or-lower answers, counting their guesses, then connect the strategy explicitly to binary search.

Time: Prep 5 min · Activity 15 min · Discussion 10 min · **Materials:** none required; paper to tally guesses

Instructor note: *Big O describes how an algorithm's cost grows as input size grows, not its exact runtime. It deliberately leaves out constant factors, the specific hardware, and behavior at small input sizes, where a 'worse' Big O algorithm can sometimes run faster in practice. It's a tool for comparing scaling behavior, not a stopwatch.*

PART III

Putting It Together

Chapters 25–26 — a complete working example and a tour of real languages.

Chapter 25: Putting It Together [Core]

Book pages: 121–126

Learning objective: Trace a complete, working program (a simple task manager) and identify where each fundamental from Parts I and II actually shows up.

Key terms

end-to-end example · integration

Discussion questions

- Pick any two fundamentals from earlier chapters (e.g., error handling and collections) and point to exactly where they appear in this chapter's program.
- What would break first if the error handling were removed from this example, and why?

Suggested activity

This chapter works well as the basis for a capstone assignment (see Capstone Project Ideas below) rather than a single-class activity.

Time: See Capstone Project Ideas for time and scope · **Materials:** see Capstone Project Ideas

---

Chapter 26: A Tour of the Languages [Core]

Book pages: 127–130

Learning objective: Recognize how the book's fundamentals surface, with different syntax, across a range of real languages, and identify a reasonable first language to learn next.

Key terms

paradigm · syntax vs. concept

Discussion questions

- Pick one 'classic' and one 'modern' language from the chapter. What's genuinely different about them, and what's just different syntax for the same underlying idea?
- Based on this chapter and your own goals, which language would you choose to learn next, and why?

Suggested activity

Have each student pick one language from the chapter, find one short real code sample online in that language, and annotate it, labeling each part with the fundamental from this book it corresponds to (variable, loop, function, etc.).

Time: Prep 5 min · Activity 20 min · Discussion 10 min · **Materials:** internet access or printed code samples

---

Using the Book for AI-Code Literacy

A growing share of this book's readers will never write a program from a blank page; they'll direct an AI tool to generate code and need to judge whether the result is actually correct. The book's vocabulary, variables, conditionals, loops, functions, parameters, return values, error handling, is exactly what that judgment requires. This unit works well after Chapter 11 (once students have the core vocabulary) and again after Chapter 25, as a capstone-adjacent exercise applied to a real generated example instead of the book's own.

Suggested activities

- Ask an AI system to generate a small function from a plain-English prompt. Have students identify its variables, conditionals, loops, parameters, and return value using the book's own vocabulary.
- Generate two different solutions to the same prompt and have students compare their structure and approach, not just whether both happen to work.
- Have students identify the assumptions the generated code makes about its input (What if the input is empty? Negative? The wrong type?).
- Using Chapter 11's vocabulary, have students identify what error handling is missing and what should be added.
- Ask students to explain the code in their own words without relying on the AI tool's own explanation of it.
- Have students test whether the generated code actually satisfies the original prompt, including edge cases the prompt didn't spell out.

Time: Prep 10 min (generate or prepare a code sample) · Activity 25 min · Discussion 15 min · **Materials:** access to an AI coding assistant, or pre-printed generated code samples for classrooms without AI access

For classrooms without access to an AI tool, bring in 2–3 pre-generated code samples (including at least one with a real, findable flaw) and run the same activities as a shared exercise rather than an individual one.

Capstone Project Ideas

Chapter 25 walks through a complete, working task-manager program built entirely from fundamentals introduced earlier in the book. It's designed to double as a capstone model. A few ways to use it as a summative assessment:

Option A: Extend the example

Have students take the task manager from Chapter 25 and add one new feature in pseudocode (due dates, priority levels, or categories), explicitly identifying which fundamentals from the book the new feature relies on.

Option B: Design a parallel system

Have students design (in pseudocode, on paper) an equivalent end-to-end program for a different small domain — a lending library, a simple inventory tracker, a grade book — that uses at minimum: variables and types, a collection, a function, a conditional, a loop, and error handling. Require a short written explanation of each design choice.

Option C: Translate to a real language

For classes that have also covered a specific language, have students implement the Chapter 25 example (or their own parallel design from Option B) in that language, and pair it with the corresponding QF Code companion exercises for scaffolding.

Capstone rubric (language-neutral)

Criterion	Weight	What to look for
Correct use of core concepts	30%	Variables, control flow, and collections are used correctly and appropriately for the problem
Logical organization	20%	Work is broken into sensible functions/modules rather than one undifferentiated block
Explanation of design choices	20%	Student can say why they chose a given structure, not just that it works
Error handling and edge cases	15%	Reasonable failure cases are anticipated and handled, not just the happy path
Readability and clarity	15%	Naming, comments, and structure follow Chapter 22's guidance

This rubric is intentionally language-neutral. For Option C submissions in a specific language, add a separate syntax-correctness criterion scored by your own course standard rather than folding it into the categories above.

General Assessment Approach

The discussion questions throughout this guide work well as short-answer reading checks or bell-ringer prompts. For more formal assessment, consider:

- Concept quizzes built from the Key Terms lists, asking students to explain a term in their own words rather than only define it.
- "Spot the bug" exercises using short pseudocode snippets with a single deliberate error tied to that chapter's concept (an off-by-one loop, a missing base case, a variable used out of scope).
- Trace-and-predict exercises, where students hand-trace a short piece of pseudocode and predict its output or final state, particularly effective for Chapters 5, 9, 13, and 15.

Avoid grading purely on syntax correctness in any single language during this unit. The book's entire premise is that the underlying concept is what should be assessed; requiring exact syntax defeats the purpose of using a language-neutral text.

Printable Student Worksheets

A companion set of printable, copy-ready student worksheets, a program-tracing sheet, a boxes-and-arrows reference diagram, a conditionals worksheet, a loop-tracing table, a recursion call-stack template, a collection-selection worksheet, a capstone planning sheet, and an AI-generated-code review checklist, is provided as a separate file (Underneath the Syntax: Printable Worksheets) rather than bundled into this guide. Keeping it separate lets you copy and distribute the worksheets to students directly, without also handing out the rubric, instructor notes, and answer guidance in this document.

Appendix: Using the Glossary

The book's glossary (page 133) defines every term used in the book in plain language, with a pointer back to the chapter where it's introduced in full. It's built as a quick-reference tool, not a replacement for the chapters, and works well as a standing classroom reference or a quiz-prep handout on its own.

A Note on Standards Alignment

This book was not written against a specific standards framework, and this guide does not claim formal alignment with any particular state, CSTA, or College Board standard. In practice, the concepts in Parts I and II (sequencing, variables, control flow, data structures, functions, and a first exposure to algorithmic thinking) overlap substantially with the core ideas found in most introductory computer science standards. Instructors should cross-reference this guide against their own required standards documents rather than treat any mapping here as official.

Evaluation Materials and Classroom Use

This guide and the printable worksheet set are public educator resources. The complete Quiz and Exercise Bank, separate instructor Answer Key, and a complimentary digital evaluation copy are available by request to educators and program leaders considering adoption. Requests are reviewed individually; print copies may be available selectively. Limited virtual author Q&A sessions may be available for selected pilots or classroom adoptions, subject to scheduling.

Educators may reproduce the printable worksheets for learners in educational settings they directly lead, including schools, tutoring and library programs, workforce programs, paid bootcamps, and employer-run training. The worksheets may not be sold separately, reposted as a complete public collection, incorporated into materials offered for resale, or redistributed as editable source files.

About the Author

Edison Mooers is a software engineering leader, author, and independent publisher with more than three decades of experience, from COBOL to modern enterprise architecture and AI-enabled software. Self-taught on a hand-me-down TRS-80, he later earned bachelor's and master's degrees in Information Technology while working full-time. He created QF Code and wrote Underneath the Syntax to make durable programming concepts easier to see.