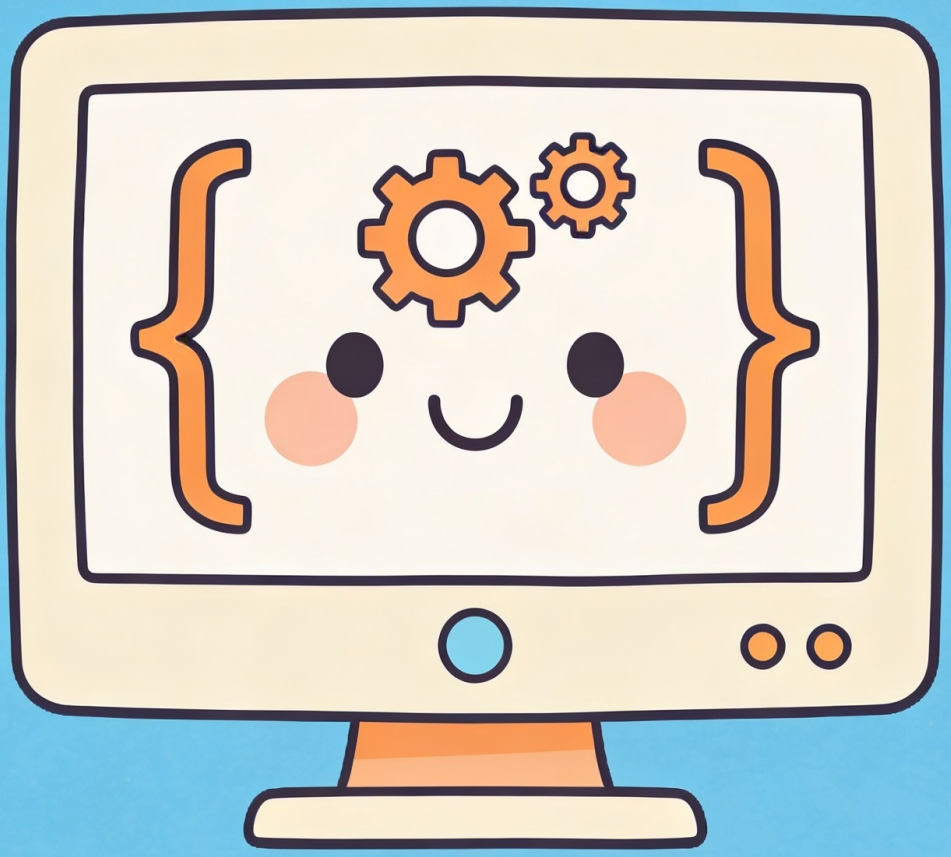
Coding for Kids



A Friendly Introduction to How Computers Think

Sarah Whitman

Coding for Kids

A Friendly Introduction to How Computers Think

by Sarah Whitman

CONTENTS

I What Computers Actually Do

II One Step at a Time: The Power of Sequences

III Again and Again: How Loops Save the Day

IV Making Choices: If This, Then That

V Giving Your Recipe a Name: Functions

VI Bugs, Patience, and Your First Real Project

CHAPTER I

What Computers Actually Do

Imagine you could ask a genie for anything. Not a movie genie with opinions and a smirk — a genie that does exactly what you say. Every word. Every time. No shortcuts, no assumptions, no reading between the lines.

That genie is a computer.

People talk about computers like they're smart. Incredibly smart. And in some ways that's fair — a modern laptop can do more than three billion calculations per second. Your brain, working flat out, manages somewhere around a hundred calculations per second on a good day. So yes, computers are fast in a way that's almost impossible to picture. But fast is not the same as smart. A calculator is fast. A stopwatch is fast. Speed doesn't mean understanding.

Here is the thing nobody tells you at the start: computers are genuinely, profoundly, almost comically obedient. They have no common sense. No hunches. No instincts. They cannot look at a messy instruction and figure out what you probably meant. If you give a computer a bad instruction, it doesn't shrug and make its best guess. It either crashes, freezes, or does something baffling — like printing the word "sandwich" 40,000 times at 3 a.m.

That's not a made-up example. Programmers everywhere have horror stories like it.

The Genie Problem

Here's a story worth sitting with. A child wishes to a magical genie: "Give me a million dollars." The genie does. It also disappears. The child forgot to say "and don't take it back," so an hour later the genie returns and reclaims every penny. The instructions were followed perfectly. The result was still a disaster.

Computers work exactly like this. They are extraordinarily good at following instructions. They are completely helpless at interpreting them.

Mitchel Resnick, who leads the Lifelong Kindergarten research group at MIT — the team that built Scratch, the coding platform used by more than 100 million young people worldwide — puts it simply: learning to code is learning to express your ideas clearly enough for a machine to act on them. Not almost clearly. Not mostly clearly. Precisely clearly.

That's the whole game. Coding is not magic. It's not even really about computers, not at its heart. It's about thinking carefully enough that you could explain something to a machine with zero imagination and have it come out right.

What a Computer Is Actually Doing

Open any app on your phone. Let's say you tap the camera. In the fraction of a second before the screen lights up, your phone's processor has already executed millions of tiny instructions: check that the camera hardware is connected, request permission to use it, load the software that handles the image sensor, prepare a frame buffer to store the picture, display a preview on the screen — and on and on. Each instruction is a single, tiny command. None of them require the phone to understand what a photo is. The phone doesn't know what a photo is. It just follows the list.

Every program ever written — every game, every app, every website — is a list of instructions. The programmer's job is to write a list that's correct enough, and specific enough, that the machine can run it without needing to guess.

This is great news. It means you don't need to be a genius to code. You need to be precise.

The Sandwich Robot

Let's do an experiment. Forget code for a moment. Pretend you're writing instructions for a robot that will make you a peanut butter sandwich. The robot is capable but literal — it has no idea what a sandwich is, what hunger is, or why anyone would want one. It just follows steps.

You might write:

```
• Get the peanut butter.
• Get the bread.
• Make the sandwich.
```

And the robot would stop at step 3. Make the sandwich how, exactly? With what? In what order? The robot isn't being difficult. It genuinely has no idea. That instruction is as clear to a computer as the phrase "do the thing" is to you.

Let's try again, this time more carefully:

```
• Go to the kitchen cupboard.
• Open the cupboard door.
• Pick up the loaf of bread.
• Close the cupboard door.
• Carry the bread to the counter.
• Open the bread bag.
• Take out two slices of bread.
```

- Place both slices flat on the counter.
- Go to the kitchen cupboard.
- Open the cupboard door.
- Pick up the jar of peanut butter.
- Close the cupboard door.
- Carry the jar to the counter.
- Unscrew the lid of the jar.
- Pick up a butter knife.
- Dip the knife into the peanut butter.
- Spread the peanut butter across one side of the first slice of bread.
- Place the second slice of bread on top of the first slice, flat sides touching.
- The sandwich is ready.

```

That's nineteen steps to make a sandwich that takes you about forty-five seconds. Welcome to programming.

This kind of plain-language instruction list — not written in any real coding language, just ordinary words — is called pseudo-code. "Pseudo" means fake, or pretend. Pseudo-code is the skeleton of a program. It lets you plan your logic before you worry about the exact spelling and punctuation a real programming language demands. Code.org, one of the largest coding education organisations in the world, uses pseudo-code activities right from day one — because thinking through the steps carefully is the hardest part, and you can do it entirely on paper.

Notice how step 17 says "spread the peanut butter across one side of the first slice." That word "one" matters. Without it, maybe the robot covers both sides. Maybe it spreads peanut butter on the bottom. You'd be surprised how often a missing word causes a program to go sideways.

## You Already Think in Code

Here's the secret. You've been doing this your whole life.

Think about the last time you explained something to a younger sibling or a friend who'd never done it before. How to tie shoelaces. How to play a card game. How to get to your house from school. You broke it into steps. You put the steps in order. You probably had to go back and add a step you forgot ("oh, and first you have to shuffle the cards"). You caught yourself being vague — "then you just sort of loop it around" — and found a clearer way to say it.

That is exactly the thinking process behind writing code. Sequence. Precision. Revision.

Resnick's team at MIT designed Scratch specifically to make this feel natural — more like snapping toy bricks together than writing an essay. But even in Scratch, the underlying challenge is the same: you have to
```

decide what happens first, what happens second, what happens when things go wrong. The blocks make the typing easier. The thinking is still yours.

Clearing Up the Magic

One more thing worth saying out loud: computers are not impressive because they think. They're impressive because they're fast and tireless. They'll run your sandwich instructions ten million times without getting bored, without skipping a step, without deciding to make toast instead. Humans are the opposite — we get bored quickly, we take shortcuts, we use intuition. That's not a flaw. That intuition is what lets you understand that "make the sandwich" is a goal, not a single step.

Computers handle the repetition. You handle the ideas.

Between the two of you, you can build almost anything.

And you start — every single time, whether you're nine or ninety — by asking the same question: What, exactly, needs to happen? Not roughly. Not sort of. Exactly.

Get comfortable with that question, and you've already started coding. Even without a screen, even without a keyboard, even sitting right here reading this page.

The genie is waiting. Let's figure out what to ask for.

CHAPTER II

One Step at a Time: The Power of Sequences

Imagine this. You wake up, grab your shoes, put them on — and then try to pull on your socks. It doesn't work. Obviously. You'd have to take the shoes off first, do the whole thing again, and now you're late for school.

Ridiculous, right? Nobody does it that way. But here's the thing: if you gave a computer instructions in that order, it would try anyway. No hesitation, no raised eyebrow, no common sense stepping in to save you. It would just do exactly what you said, in exactly the order you said it — and end up with shoes on the wrong feet and socks dangling uselessly from your hands.

That's sequences. And they're one of the most powerful ideas in all of coding.

What a Sequence Actually Is

A sequence is just a list of instructions carried out one at a time, in a specific order. Step one happens first. Then step two. Then step three. The computer doesn't skip ahead, doesn't guess what you probably meant, and doesn't swap things around because they'd work better a different way.

It follows the sequence. Exactly. Every time.

Computer scientists at Code.org describe a sequence as "an ordered list of instructions" — and that last word, ordered, is doing a lot of work in that sentence. The order isn't decoration. It's the logic. Change the order, and you've written a completely different program, even if every single instruction stays the same.

Think about that for a second. Same words, different order, totally different result. You already know this is true from real life. "Eat lunch, then brush teeth" leads to clean teeth. "Brush teeth, then eat lunch" leads to a weird-tasting sandwich and the whole brushing-teeth thing was pointless anyway.

Sequences are everywhere. You just haven't been calling them that.

The Morning Routine

Let's map out a morning. A real one — messy, rushed, the kind where you can't find one of your shoes and your cereal goes soggy while you're looking for it.

Here's a working sequence for getting to school:

```

- Wake up
```

- Get out of bed
- Go to the bathroom
- Brush teeth
- Get dressed (socks THEN shoes)
- Go downstairs
- Eat breakfast
- Pack your bag
- Leave the house

```

Read through it. Does it work? Mostly, yes. You arrive at school clothed, fed, and with your bag. Good.

Now try swapping just two steps. Move step 8 — "Pack your bag" — to after step 9 — "Leave the house."

``` ...

- Leave the house
- Pack your bag

```

Disaster. You're standing on the pavement trying to stuff your homework into a bag you haven't packed yet, inside a house you've already left. In code, this isn't just inconvenient. The program is broken. It does the wrong thing, and it will keep doing the wrong thing every single time you run it, because computers don't improvise.

That's the whole point of sequences: order is logic.

## Writing Your First Pseudo-Code

Here's your first real coding task. Don't panic — you don't need a computer for this one. Just paper and a pen.

Pseudo-code is instructions written in plain English (or whatever language you're comfortable with) that follow the same strict one-step-at-a-time structure as real code. Mitchel Resnick at MIT, who created Scratch, talks about coding as a new kind of literacy — a way of expressing ideas precisely and making things happen. Pseudo-code is where that literacy starts.

Pick one of these:

- Making a bowl of cereal
- Tying your shoes
- Sending a text message

Write every step. Every single one. Don't assume anything is obvious — remember, your reader is a computer with zero common sense and zero experience of being a person. "Get the cereal" isn't enough. Which
```

cupboard? Open the cupboard first, presumably? Pick up the box? Check there's actually cereal in it?

Here's what a first attempt might look like for making cereal:

```

- Open cupboard
- Take out cereal box
- Open fridge
- Take out milk
- Close fridge
- Get a bowl from the cupboard
- Get a spoon from the drawer
- Pour cereal into bowl
- Pour milk onto cereal
- Pick up spoon
- Eat

```

Looks good. But read it again. Did you close the cereal box before putting it back? Did you put anything away? If this were a real kitchen robot, it would leave everything out on the counter. Forever. Because you never told it to tidy up.

That's the game. You write the steps. You find the gaps. You fix them. That loop of writing, testing, and fixing is the heartbeat of coding — and you're already doing it.

Try It in Scratch

If you want to see your sequence actually run, Scratch is the place to start. It's free at scratch.mit.edu and was built specifically for learners your age by Resnick's Lifelong Kindergarten group at the MIT Media Lab. The whole idea behind Scratch — as Resnick explains in his 2017 book Lifelong Kindergarten — is that kids learn best when they're making things they care about, not just absorbing facts.

In Scratch, sequences are stacks of coloured blocks. You drag a block down, then another, then another. The program runs from top to bottom — first block first, last block last.

Try this:

`` when [green flag clicked] move 10 steps turn 90 degrees move 10 steps say "I made an L shape!" ``

Run it. Your sprite draws an L-shape and announces what it did. Now swap the "turn 90 degrees" block so it comes before the first "move 10 steps." Run it again. Different shape. Same blocks. Different order. Different result.

That's it. That's sequences. You just felt it.

Order Is Thinking

Here's something worth sitting with: the skill you're building isn't just a coding skill.

Every good recipe is a sequence. Not because chefs are secretly programmers, but because when you're teaching someone to bake bread, the order of steps is the knowledge. Knead the dough before it rises. Let it rise before you bake it. Bake it before you eat it. You could list every ingredient perfectly and still produce something inedible if you did those steps in the wrong order.

Sport is full of sequences, too. A tennis serve: feet position, toss the ball, backswing, contact, follow-through. Miss a step or scramble the order and the ball goes into the net. Or the crowd. Ask any coach and they'll tell you that beginners rush. They skip steps. They don't trust the sequence. Learning to slow down and do things in order, one step at a time, is genuinely hard — and genuinely worth learning.

Storytelling works the same way. Every story has a sequence: something happens, then something else happens as a result, then something changes because of that. Cause, then effect. Setup, then punchline. Tension, then release. Swap those around and the story stops making sense. You've probably read books where the ending came too early or a twist didn't land because the setup wasn't there yet. Broken sequence.

Careful, linear thinking — this happened, so then that happened — is one of the most useful things a human brain can practise. Coding just makes it visible. When your program doesn't work, you can point to the line where the order went wrong. You can see the mistake. That's actually a gift.

The Patience Part

Getting sequences right takes patience. That's not a warning — it's the whole lesson.

One of the most common beginner mistakes, whether you're coding in Scratch or writing pseudo-code on paper, is assuming the computer will fill in the gaps. It won't. If you write "make sandwich" as a single step, nothing happens — or rather, something happens that you didn't expect, because "make sandwich" isn't an instruction, it's a wish.

Break it down. Break it down again. Keep going until every step is so small and so clear that even a very obedient, very literal, not-at-all-clever machine could follow it without asking a single question.

That's the job. Not brilliance. Clarity.

And here's the good news: you can practise this anywhere. Next time you're following a recipe, read each step before you do it. Notice whether the order matters. Next time you're explaining something to a friend, pay attention to whether you're giving them the steps in the right sequence for them to actually understand.

Sequencing is a habit of mind. And once you start seeing it, you see it everywhere.

In the next chapter, you'll meet something that makes sequences even more powerful: the loop. Because sometimes, writing the same step fifty times in a row is exactly as annoying as it sounds — and there's a much better way.

CHAPTER III

Again and Again: How Loops Save the Day

Imagine you had to write "I will brush my teeth" one hundred times on a piece of paper. Not because you were in trouble — just because someone asked you to. Your hand would cramp by line twelve. You'd be bored by line three. By the end, you'd probably be writing "I wll brsh my tth" and hoping nobody noticed.

Computers don't get bored. They don't get tired. But here's the thing — if you had to write out every single instruction for a computer to do something one hundred times, you would get exhausted. And that's exactly the problem loops solve.

The Playlist That Never Stops

You already use loops every day. You just don't call them that.

Think about a music playlist set to repeat. You tap shuffle, hit play, and your ten favourite songs cycle through — over and over — until you turn it off or your battery dies. You didn't have to press play 200 times. You set up the rule once: keep playing songs until I say stop. The playlist handled the rest.

That's a loop. A rule that says: do this thing again, and again, following the same pattern, until some condition is met (or until you've done it a set number of times).

Mitchel Resnick, the MIT professor who created Scratch, talks about how children learn best when they build things that repeat and grow — not just things that run once and stop. A spinning sprite in Scratch, a song on repeat, a bouncing ball that never quite settles. Repetition, Resnick argues in Lifelong Kindergarten (2017), is how both people and programs find their rhythm.

Loops are that rhythm, written down as code.

Writing the Same Thing Fifty Times (Or Not)

Here's a real problem. Say you want a program to print a row of ten stars across the screen — like decorating a banner. No loops? Here's what your pseudo-code looks like:

`` print "" print "" print "" print "" print "" print "" print "" print "" print "" print "" ``

Ten lines. Fine. But what if you want fifty stars? A hundred? What if the number might change depending on what the user types? Suddenly you're in trouble.

With a loop, the same job takes three lines:

`` repeat 10 times: print "*" ``

Done. Clean. Easy to change — just swap the 10 for 50, or 100, or 1,000,000. The computer does the repetitive work so you don't have to.

This kind of loop is called a counted loop (sometimes you'll see it called a "for loop" in languages like Python or JavaScript). You know exactly how many times you want something to happen. You tell the computer the number. It counts, it repeats, it stops.

Two Kinds of Loops

Here's where it gets interesting. Not every loop knows in advance how many times it needs to run.

Think about filling a bucket from a garden hose. You don't stand there counting seconds — "one Mississippi, two Mississippi..." You watch the bucket. You keep the hose running until the bucket is full. Then you stop.

That's the second kind of loop: a condition-based loop (also called a "while loop" or "repeat until" loop). Instead of counting, it checks a condition. It keeps going as long as that condition is true — or keeps going until a condition becomes true, depending on how you write it.

In pseudo-code, the bucket scenario looks like this:

`` repeat until bucket is full: run the hose ``

Simple. But what happens if the bucket has a hole? The condition "bucket is full" might never become true, and the loop runs forever. Programmers call this an infinite loop — and it's usually a bug, not a feature. (Though some programs, like a playlist that genuinely should run until you stop it, use intentional infinite loops with a manual stop built in.)

In Scratch, you can see both types right in the block palette. The "repeat (10)" block is your counted loop — drag it out, type a number, drop your instructions inside. The "repeat until" block is your condition-based loop — you attach a condition shape (like "touching the edge?" or "score = 10?") to tell it when to stop. Scratch makes loops visual. Literally. The instructions sit inside the loop block, wrapped like a little sandwich.

Try It: The Repetitive Recipe

Here's your hands-on challenge. Read this recipe for making a stack of pancakes:

``` Pour batter into pan. Wait 2 minutes. Flip pancake. Wait 1 minute. Remove pancake. Put it on the plate.

Pour batter into pan. Wait 2 minutes. Flip pancake. Wait 1 minute. Remove pancake. Put it on the plate.

Pour batter into pan. Wait 2 minutes. Flip pancake. Wait 1 minute. Remove pancake. Put it on the plate. ```

That's three pancakes. Fifteen lines of code. Feel the repetition? It's almost insulting to write it out that way. Your brain spotted the pattern instantly — because human brains are very good at spotting patterns. That's one of our genuine superpowers.
```

Now rewrite it using a loop. Grab a piece of paper and try it before you read ahead.

Here's one version:

`` repeat 3 times: pour batter into pan wait 2 minutes flip pancake wait 1 minute remove pancake, put it on the plate ``

Five lines. Does the same job. And if you want eight pancakes instead of three? Change one number. That's it.

What if you don't know how many pancakes you need — you're just cooking until the batter runs out?

`` repeat until batter bowl is empty: pour batter into pan wait 2 minutes flip pancake wait 1 minute remove pancake, put it on the plate ``

Now the recipe adapts. Ten pancakes, two pancakes, twenty — it doesn't matter. The loop handles it.

This kind of thinking shows up everywhere in Code.org's CS Fundamentals courses, where students as young as eight start spotting repetition in real tasks and replacing it with loop structures. The curriculum doesn't start with code. It starts with exactly this: notice the pattern, then write a rule for it.

Spotting Patterns Is the Real Skill

Here's a thought worth sitting with.

The most important thing you learned in this chapter isn't the syntax of a loop. It's not "repeat 10 times" versus "repeat until." Those are just ways of writing down a skill you were already practising.

The real skill is pattern recognition — seeing repetition in the world and asking, "Could I write one rule that covers all of this?"

Athletes do it. A tennis coach watches a player's serve and says: "You're dropping your elbow at the same point every time." That's a loop — the same mistake, repeating. Fix the pattern, fix the serve. Chefs do it too. A good pastry recipe is full of loops: fold the dough, rest, fold again, rest again — because the baker learned that the same small action, repeated at the right intervals, builds something layered and light.

Authors do it in storytelling. The Rule of Three — three wishes, three little pigs, three chances for the hero to fail before succeeding — is a loop. It creates rhythm. It feels satisfying because repetition with variation is one of the oldest patterns in human storytelling.

When you practise writing loops in code, you're training your brain to see these patterns faster. You start asking "where is the repetition here?" as a reflex. That question is useful in maths, in music, in sport, in science. It's useful in every subject you'll study and most jobs that exist right now or will exist when you grow up.

One More Thing Before You Go

Loops are powerful. But they can also go wrong.

If you forget to include a condition that actually becomes true — if your "repeat until the bucket is full" loop is watching a bucket that can't fill — your program freezes. The computer isn't being stubborn. It's just doing exactly what you told it: keep going, keep going, keep going.

Always ask yourself: will my loop ever stop? What makes it stop? Can I be sure that stopping condition will actually happen?

That habit of asking "what ends this?" is good debugging instinct. You'll need it in the next few chapters. But more importantly, it's good life instinct. Plans that run indefinitely without a stopping condition — those are worth questioning too.

For now, grab that pancake recipe and try building it in Scratch. Use the "repeat" block. Then try swapping it for "repeat until" with a pretend "batter empty?" condition. Watch what changes and what stays the same.

Loops are one of coding's great shortcuts. And shortcuts, when they're honest about what they're doing, are just cleverness written down.

CHAPTER IV

Making Choices: If This, Then That

Every single day, without thinking about it, you make hundreds of choices.

Do I need a jacket? Is the cereal bowl empty yet? Should I cross the street now or wait? Your brain runs through these questions so fast it feels automatic. But it isn't. Hidden inside each decision is a little logical machine humming along: check a condition, pick a path, act on it.

Computers do exactly the same thing. They just need you to write the logic out loud.

The Light That Decides

Think about a traffic light. Not a fancy one. The basic three-colour kind you see at any intersection.

The light doesn't think. It has no feelings about your schedule, no idea you're running late for football practice. But it controls thousands of cars every day without making a single mistake — because the rule built into it is perfectly clear:

- If the light is red, stop.
- If the light is yellow, slow down.
- If the light is green, go.

That's it. Three conditions. Three responses. And because the rule never has exceptions, never depends on mood or guesswork, it works every time.

This is conditional logic. And it is one of the most powerful ideas in all of coding.

In Scratch — the visual coding language built at MIT's Media Lab — you'll find a block that looks exactly like this idea. It says "if \<condition\> then" with a space for instructions underneath. Simple. Clean. Unambiguous. Mitchel Resnick, the MIT professor who led the team that created Scratch, has argued that this kind of structured creative thinking — making clear decisions about when and what — is central to learning how to design anything, not just programs. In his 2017 book Lifelong Kindergarten, he describes coding as "a new type of literacy," one that lets young people express ideas with precision.

Making choices precisely. That's what this chapter is about.

If This, Then That

Let's start small.

Imagine you wake up in the morning and you're not sure whether to bring an umbrella. You look out the window. Here's the decision, written as pseudo-code:

`` if it is raining: take the umbrella ``

Read that once. Make sure you feel it. The computer checks one condition — is it raining? — and if the answer is yes (or TRUE, in computer terms), it does the thing inside the block. If it isn't raining, the whole block gets skipped. Nothing happens. You just walk out the door without thinking about it.

But what if you want the code to do something else when it's not raining? That's where else comes in.

`` if it is raining: take the umbrella else: leave the umbrella at home ``

Now you have two paths. Every time the code reaches this fork in the road, it asks the question, picks one path, ignores the other, and moves on. Clean. Decisive. No hesitation.

This is an if/else statement. It's one of the first real pieces of logic you'll write in almost any programming language on Earth — Python, JavaScript, Java, Scratch, all of them. Code.org's Computer Science Fundamentals curriculum introduces conditionals at Course B, designed for students roughly your age, using almost exactly this kind of real-world scenario. Weather decisions. Hungry or not hungry. Light or dark. The examples feel obvious on purpose, because the structure is what you're learning. Once you have the structure, you can apply it to anything.

When There Are More Than Two Choices

Here's where it gets interesting.

What if there are more than two possibilities? Go back to the traffic light. We only had three colours. Let's write that out properly:

`` if the light is red: stop the car else if the light is yellow: slow down else if the light is green: go ``

That's an if/else-if chain. The computer works down the list one condition at a time. The moment it finds a condition that's true, it does that action and skips everything below. If red is true — done. Never even checks yellow or green. Efficient, the same way you don't keep reading a menu after you've decided what you want.

Let's try one more example, something you might actually care about. Suppose you're writing a game in Scratch where the player collects points and you want to give them a rating at the end:

`` if score is greater than 90: display "Amazing!" else if score is greater than 70: display "Great job!" else if score is greater than 50: display "Not bad!" else: display "Keep practising!" ``

Notice the else at the bottom with no condition attached. That's a catch-all. If the score doesn't match any of the conditions above — if it's 30, or 10, or zero — the final else catches it. No case gets left unhandled. Nothing falls through the cracks.

That last else is a safety net. Good coders always ask: what if none of my conditions are true? Asking that question before you write a single line of code saves you hours of confusion later.

Activity: Draw Your Decision Tree

Here's something you can do right now with just paper and a pen. No computer needed.

Think of any real-life decision — something that has more than one possible outcome. Picking what to have for lunch. Choosing which route to take to a friend's house. Deciding whether to go outside.

Now draw it like a tree. Start at the top with a diamond shape (that's the traditional symbol for a decision in diagrams). Write your question inside it: Is it raining? Draw two lines coming down — one labelled YES, one labelled NO. At the end of each line, draw a rectangle and write the action: Take umbrella or Leave umbrella.

If either action needs another decision after it — say, if you take the umbrella, you then need to decide which bag to carry it in — draw another diamond hanging below that rectangle and repeat.

Keep going until every path ends in a clear action with no more branching.

What you've just drawn is a flowchart. Engineers, nurses, pilots, and software developers all use them. Serious ones. At NASA, mission controllers use decision trees to run through contingency plans during launches — each "if this fails, then do that" step mapped out before a single rocket fires. The logic isn't different from what you just sketched. It's just higher stakes.

When you've drawn your tree, try translating one branch into pseudo-code. You'll probably find it's easier than you expected, because you've already done the hard thinking on paper.

Asking the Right Question First

Here's the secret that separates good code from messy code: the question matters more than the answer.

Look at this attempt at pseudo-code for a vending machine:

`` if the user presses a button: give them the snack ``

That seems fine. But what question are you not asking? Did they put money in? Is the snack actually in stock? Is the machine plugged in?

Bad conditional logic usually isn't wrong because the code is hard to write. It's wrong because someone forgot to ask an important question at the start. They jumped straight to what do I do without fully thinking through what conditions are possible.

Before you write an if-statement, take thirty seconds. List every state the situation could be in. A traffic light has three colours. A quiz answer is either right or wrong. A character in your game can be standing still, walking, jumping, or falling. Write those states down first. Then the code almost writes itself.

Mitchel Resnick describes this kind of thinking as "thinking like a designer" — asking what are all the possibilities before committing to a solution. That habit is useful whether you're coding a game, planning a birthday party, or building a robot out of LEGO. The skill isn't really about computers. The skill is about

clarity.

Conditions Are Questions in Disguise

Every if-statement is really just a yes-or-no question. Is it raining? Is the score above 90? Is the light red? Your code waits for the answer, then acts.

What makes conditional logic feel powerful — almost magical — is how fast it runs. A computer can check ten thousand conditions in less than a second. But none of that speed matters if the conditions themselves are badly formed. Fast bad thinking is still bad thinking.

So slow down before you write. Ask your question precisely. Draw your tree if it helps. Then let the code carry it out.

You've now added one of the most fundamental tools in programming to your toolkit. Sequences told you order matters. Loops taught you how to repeat. Conditionals teach you how to choose.

And choosing well — in code, in decisions, in life — always starts with asking exactly the right question.

CHAPTER V

Giving Your Recipe a Name: Functions

Imagine you are making lunch for yourself. Easy enough. You grab the bread, spread the peanut butter, add the jelly, press the slices together, done.

Now imagine your little brother wants one too. You do the whole thing again. Then your mum asks for one. Again. Then your friend shows up. Again. By the fourth sandwich you are so bored you could scream, and you have not even eaten yet.

Here is the thing: a computer feels exactly this way about repeated instructions. Not bored, obviously — computers feel nothing — but it is wasteful. Writing the same ten steps over and over is how code gets messy, hard to read, and full of mistakes. There is a better way. It is called a function, and once you understand it, your code will never be the same.

The Recipe Box Idea

Think about how a professional chef works. Gordon Ramsay does not recite his beef Wellington recipe from memory every single time he needs it. He wrote it down once, named it, and now he can summon it whenever he wants just by saying the name. The instructions live in one place. He can use them a thousand times without rewriting a single word.

Functions work exactly like that.

A function is a named, reusable bundle of instructions. You write the bundle once. You give it a name. Then, whenever you need those instructions to run, you just say the name — and off the computer goes, following every step inside.

Mitchel Resnick, the MIT professor who helped build the Scratch programming language, talks about this idea in his book Lifelong Kindergarten (MIT Press, 2017). He describes coding as a form of creative building — snapping together pieces you have already made to create something bigger. Functions are those pieces. They are your building blocks.

Writing Your First Function

Let's go back to the sandwich. In pseudo-code — that loose, plain-language style of writing instructions you have been using throughout this book — a basic function looks like this:

`` FUNCTION makeSandwich(): get two slices of bread spread peanut butter on the first slice spread jelly on the second slice press the slices together put the sandwich on a plate END FUNCTION ``

That is the whole thing. You have defined the function. You told the computer: here is a bundle of steps, and I am calling this bundle makeSandwich.

But here is the crucial part — defining a function and calling a function are two different things. Right now, nothing has happened. The recipe is written down, but nobody made a sandwich yet.

To actually run those instructions, you call the function by writing its name:

`` makeSandwich() ``

Two words and a pair of brackets. The computer reads that line, jumps straight to your makeSandwich definition, runs every step inside it, and comes back. Just like that, a sandwich exists.

Want to make four sandwiches? Four lines:

`` makeSandwich() makeSandwich() makeSandwich() makeSandwich() ``

Yes, you could also use a loop here — you learned those in the last chapter. But notice how readable this is. Even someone who has never seen code before can look at makeSandwich() and understand what is happening. That clarity is one of the biggest gifts functions give you.

Changing the Order — Parameters

Here is where it gets really interesting.

Peanut butter and jelly is one sandwich. What about cheese? What about turkey? What if your brother wants sourdough bread and your mum wants wholemeal? Right now, your function only knows how to make one specific sandwich. It is not very flexible.

The solution is called a parameter. A parameter is a piece of information you pass into a function when you call it. It lets the function behave differently depending on what you give it.

Watch what happens when you add parameters to makeSandwich:

`` FUNCTION makeSandwich(breadType, filling): get two slices of breadType spread filling on the first slice press the slices together put the sandwich on a plate END FUNCTION ``

Now the function has two parameters: breadType and filling. They are like blank spaces in the recipe — gaps you fill in each time you call it.

Calling it looks like this:

`` makeSandwich("sourdough", "peanut butter and jelly") makeSandwich("wholemeal", "cheese") makeSandwich("white bread", "turkey") ``

Three different sandwiches. Same function. The instructions inside never changed — but the inputs did. That is the power of parameters. One recipe, infinite variations.

Code.org's curriculum for young learners describes parameters almost exactly this way: as values that let a single function handle many different situations, the same way a vending machine button works for every customer, not just one.

Getting Something Back — Return Values

So far, makeSandwich does its work quietly. It makes the sandwich and stops. But sometimes you want a function to give you something back — an answer, a result, a value you can use later.

That is called a return value.

Say you want the function to tell you how many calories are in the sandwich so you can write it in a food diary. The function needs to calculate that number and hand it back to you.

`` FUNCTION makeSandwich(breadType, filling): get two slices of breadType spread filling on the first slice press the slices together put the sandwich on a plate calories = 350 RETURN calories END FUNCTION ``

Now when you call the function, you can catch what it returns:

`` myCalories = makeSandwich("sourdough", "peanut butter and jelly") PRINT "This sandwich has " + myCalories + " calories." ``

The screen would show: This sandwich has 350 calories.

The function ran its steps, calculated a value, and handed it back. You stored it in a variable called myCalories and used it. Parameters go in, return values come out. Input, process, output. Sound familiar? It is the same basic idea computers have been doing since Chapter 1 — just wrapped up neatly with a bow.

Functions in Scratch: "My Blocks"

If you are using Scratch, you can build your own functions using a feature called My Blocks. You will find it in the block palette on the left side of the screen. Click "Make a Block," give it a name, and drag the steps you want inside it. From that moment on, your new block appears in the palette just like any other — pink, clickable, ready to use anywhere in your project.

Scratch 3.0, released in January 2019, made My Blocks even more useful by letting you add inputs to them — which is exactly the same idea as parameters. Name your block drawSquare, add an input called size, and suddenly your block can draw a tiny square or a giant one depending on what number you give it.

Try it. Make a Scratch project where one My Block draws a star, another plays a sound, and a third moves a sprite across the screen. Then build something by calling those blocks in different orders. You are not just coding. You are composing.

The Bigger Picture

Here is something worth sitting with for a moment.

Functions are not just a coding trick. They are how complex things get built in the real world.

An architect does not redesign a door from scratch for every room in a building. There is a standard door component — width, height, hinge placement — and she reuses it dozens of times, adjusting the measurements (parameters!) as needed. A novelist does not invent a new way to write dialogue for every chapter. A jazz musician does not compose a new scale for every solo. They all work from named, reusable building blocks they trust.

The whole of modern software runs this way. When you search Google, thousands of small functions fire in sequence — one fetches your query, one checks the spelling, one ranks results, one formats the page. None of those teams wrote everything from scratch. They called functions that other teams had already built, tested, and named. The same way a chef builds a five-course meal from mastered components rather than inventing cooking itself each evening.

Small, named, reusable. That is the secret.

Your Turn: Name That Function

Grab a piece of paper. Pick something you do regularly — brushing your teeth, feeding a pet, sending a text message. Write out all the steps. Now wrap them in a function:

`` FUNCTION [yourFunctionName](): step one step two step three ... END FUNCTION ``

Now ask yourself: does this function always do the exact same thing, or does it change depending on some input? If your tooth-brushing function changes based on which toothbrush you use, add a parameter: brushTeeth(toothbrushType).

Does it produce a result you want to know about? Add a return value.

You are not just playing pretend. This is exactly how professional programmers think before they write a single real line of code. They ask: what does this bundle need to know, and what does it need to give back?

Ask those two questions. Write the function. Call it by name.

That is the whole art of it.

CHAPTER VI

Bugs, Patience, and Your First Real Project

Something went wrong the moment you ran your code. The screen shows a red error message. Or worse — no error at all, just the wrong answer, staring back at you.

Take a breath. This is not a sign that you're bad at coding. It's a sign that you are coding.

Every single programmer alive — from a nine-year-old with a new Scratch account to the engineers who built the app on your parent's phone — makes mistakes. The word for those mistakes is bugs. And the story of where that word came from is one of the best stories in all of computing.

The Moth That Started Everything

On 9 September 1947, a team of engineers was working on a giant computer called the Harvard Mark II. The machine filled an entire room. It used thousands of mechanical switches called relays to crunch numbers, and on that September afternoon it kept producing wrong answers.

The team searched for the problem for hours.

Then someone found it. Literally inside the machine — stuck in relay number 70 — was a moth. A real, dead moth. It had flown in, gotten trapped between the relay contacts, and caused the computer to malfunction.

One of the engineers on the team was Grace Hopper, who would go on to become one of the most important figures in the history of computing. The team taped the moth into their logbook and wrote next to it: "First actual case of bug being found."

That logbook is now in the Smithsonian Institution in Washington, D.C. You can look it up.

The word "bug" had been used by engineers before that day, but Grace Hopper loved telling the moth story for the rest of her life. She used it to make a simple point: even the smartest people, working on expensive, important machines, run into problems they didn't expect. That is not failure. That is just how building things works.

Two Bugs That Changed History

Here's what's humbling about bugs: they don't only happen to beginners.

In 1999, NASA launched the Mars Climate Orbiter — a spacecraft that cost $327.6 million and took years to build. Its job was to orbit Mars and study the planet's weather. After a nine-month journey through space, the Orbiter arrived at Mars and began its entry manoeuvre.

Then it disappeared. Forever.

The investigation that followed found the cause: one team at Lockheed Martin had been sending navigation data in imperial units (pound-force seconds). NASA's own software expected metric units (newton-seconds). Two different measurement systems. Nobody caught it. One small mismatch in units sent a $327.6 million spacecraft into Mars's atmosphere at the wrong angle, where it burned up.

That's the kind of bug that keeps engineers awake at night.

Then there's Y2K. Back in the 1960s and 70s, computer memory was expensive — genuinely expensive, like a few thousand dollars per kilobyte. So programmers made a decision to save space: instead of storing a full four-digit year (like 1974), they stored just the last two digits (74). Sensible shortcut. At the time.

Fast-forward to the late 1990s. The year 2000 was coming. And suddenly everyone realised: what happens when a computer sees "00"? Does it think the year is 2000, or does it think it's 1900? Banks, hospitals, power grids — all of them ran old software built on that two-digit shortcut. The world spent an estimated $300 billion fixing it before the clock struck midnight on 31 December 1999.

A small decision. Enormous consequences. Years of work to repair.

If Grace Hopper's team found a bug in a room-sized computer, and NASA lost a spacecraft to a units mismatch, and the entire planet nearly tripped over a two-digit date shortcut — then you do not need to feel embarrassed when your Scratch project does something unexpected.

How to Actually Fix a Bug

Here's the thing about bugs: they are almost always fixable. The trick is having a method instead of just panicking and clicking things randomly.

Think of yourself as a detective. A detective doesn't just run around hoping to bump into the answer. A detective looks at the evidence, forms a theory, and tests it.

Step one: Read the error.

When something goes wrong, slow down and actually read the message. Error messages look scary. They use strange words. But they usually tell you where the problem is, even if they don't say exactly what you did wrong. "Line 7" means look at line 7. "Expected a number" means you probably put text somewhere a number should go.

Step two: Form a guess.

Say out loud — yes, out loud — what you think went wrong. "I think I forgot to set the score back to zero when the game restarts." Or "I think the loop is running one too many times." You don't have to be right. You just need a starting point.

Step three: Change one thing.

This is crucial. Change only one thing at a time, then test. If you change five things at once and the bug disappears, you have no idea which change fixed it. If you change five things and the bug gets worse, you have no idea what you broke. One change. Test. Repeat.

Step four: Test again.

Did it work? Great. Now test a few more cases to make sure you didn't accidentally create a new bug while fixing the old one. Did it not work? Go back to step two with a new guess.

That's the whole loop. Read. Guess. Change one thing. Test. Again.

Mitchel Resnick, the MIT professor who created Scratch, writes in his book Lifelong Kindergarten that the best learning happens when kids treat projects as something you tinker with — trying things, seeing what breaks, adjusting, and trying again. He calls it a "creative learning spiral." It looks a lot like debugging, because it basically is debugging. Messing around and refining is not a detour from learning. It is the learning.

Your First Real Project

Right. Let's build something.

You're going to make a number-guessing game. The computer picks a secret number between 1 and 20. You type a guess. The computer tells you if you're too high, too low, or exactly right. That's it. Simple, satisfying, and full of everything you've learned in this book — sequences, loops, and if/else decisions.

If you're using Scratch (scratch.mit.edu):

Start a new project. You'll need four pieces:

- A variable called secretNumber. When the green flag is clicked, set it to pick random 1 to 20.
- A variable called guess.
- A loop — use repeat until — that keeps asking for a guess until guess = secretNumber.
- Inside the loop, an if/else chain: if guess < secretNumber, say "Too low! Try higher." If guess > secretNumber, say "Too high! Try lower."
- After the loop ends, say "You got it!" with a congratulations message.

Run it. It probably won't work perfectly the first time. That's expected. Read what happens, form a guess about what went wrong, change one thing.

If you're using Replit (replit.com) and want to try Python:

``` import random

secret = random.randint(1, 20) guess = 0

while guess != secret: guess = int(input("Guess a number between 1 and 20: ")) if guess < secret: print("Too low!") elif guess > secret: print("Too high!")
```

print("You got it!") ```

Type that in. Run it. When it works, try changing 20 to 100. See what breaks. See what doesn't.

If you want to do this entirely on paper:

Draw the flowchart. A diamond shape for each decision (Is the guess too low? Is the guess too high?). Arrows pointing to "Too low!", "Too high!", or "You got it!" Play the game with a friend — one person secretly writes a number, the other follows the flowchart, asking questions until they get there.

Paper programming is real programming. The thinking is identical.

Once your game works, make it yours. Add a counter that tracks how many guesses it took. Tell the player at the end. Or add a silly message when they win. Or make it harder. Or easier. The game is a starting point, not a finish line.

One More Thing

You've reached the last chapter of this book.

Here's what I want you to carry with you — not the syntax, not the exact steps, but this: coding is a way of thinking clearly. It's about breaking big problems into small steps, spotting patterns, making decisions, and trying again when things go wrong.

Grace Hopper spent a lifetime building tools that let more people write code. Mitchel Resnick built Scratch because he believed that every kid deserves a chance to be a creator, not just a consumer. Neither of them thought the goal was to churn out programmers. The goal was to help people think.

You might grow up to write code every day. Or you might become a doctor, a musician, a chef, a historian, or something that doesn't exist yet. It doesn't matter. The habits you've practised here — patience, precision, creative problem-solving, the willingness to be wrong and try again — those travel with you everywhere.

Bugs are not proof that you're failing. They're proof that you're building something real.

Keep building.