Code Without Bars

A STEP-BY-STEP GUIDE TO CODING FOR BEGINNERS

Published by: Encrypter15 Press
First Edition: 2025

Chapter 1: The Power of Programming

Welcome to the world of coding. You might be thinking, “Programming? Isn’t that for math geniuses or people with fancy degrees?” Nope. Coding is for *you*—yes, you, reading this right now. It’s a skill that anyone with curiosity and a bit of persistence can learn. And trust me, you’ve already got what it takes to start.

Let’s break it down. Programming is just giving instructions to a computer to do something useful or cool. It’s like writing a recipe for your favorite dish or giving directions to a friend who’s lost. You tell the computer what to do, step by step, and it follows along. The best part? You don’t need to be a tech expert or have a computer in front of you to start learning. All you need is your brain, a pen, and some paper.

In this book, we’re going to learn Python, a programming language that’s simple to read, powerful to use, and one of the most popular in the world. But before we dive into code, let’s talk about *why* coding matters—especially for you.

What Can Coding Do?

Coding is a tool for solving problems, expressing creativity, and building a future. Here are a few things you can do with it:

- **Solve everyday problems.** Imagine creating a program to track your budget, organize a workout plan, or even calculate how much time you've got left to finish this book. Coding lets you build tools that make life easier.
- **Tell stories.** You can write programs that create interactive stories or games, like a quiz that tests your friends' knowledge or a choose-your-own-adventure game.
- **Open doors.** Coding is a skill that's in demand everywhere—businesses, schools, even creative fields like music and art. Learning to code is like learning to read and write: it gives you a way to communicate and create in a world that runs on technology.
- **Build confidence.** Every time you write a program that works, you prove to yourself that you can tackle tough challenges. It's a reminder that you're capable of more than you might think.

Here's the thing: coding isn't about being "smart" in the way people usually talk about. It's about thinking clearly, breaking problems into smaller pieces, and not giving up when things don't work the first time. You've already done that in your life—whether it's figuring out how to fix something that's broken, teaching someone a skill, or just getting through a tough day.

Why Python? Why Now?

We’re going to use Python because it’s like talking to a friend who gets you. The words and instructions in Python look a lot like regular English, so it’s easier to pick up than other programming languages. Plus, Python is used all over the place—by scientists, game developers, website builders, and even people automating boring tasks like sorting files.

You might be wondering, “Why should I learn this now?” Maybe you’re in a place where computers feel far away, or maybe you’ve never touched a keyboard. That’s okay. Coding is a way to take control of your future, no matter where you are. It’s a skill you can start learning with just your mind and some paper. And when you do get access to a computer, you’ll be ready to make things happen.

Think of this book as a guide to building something new. Each chapter is a step toward creating programs that *you* design—things that matter to you. By the end, you’ll have built a calculator, a quiz game, even a simple chatbot. More than that, you’ll have a way of thinking that lets you tackle any problem, big or small.

You’re Already a Programmer (Sort Of)

You might not realize it, but you’ve been solving problems like a programmer your whole life. Ever given someone directions to a place you know well? You had to break it down: “Go straight, turn left at the big tree, keep going until you see the red sign.” That’s what coding is—breaking a task into clear steps.

Or maybe you’ve followed a recipe. You gather ingredients, mix them in a certain order, and adjust if something’s off (like adding more salt). Programming is the same: you give instructions, test them, and tweak until it works.

Here’s a quick exercise to prove you’re already thinking like a coder. Grab a pen and paper (or just think it through). Imagine you’re explaining to someone how to make a peanut butter and jelly sandwich. Write down the steps, one by one. Be as clear as possible, like you’re talking to someone who’s never made a sandwich before.

Done? Look at your list. Those steps are like a program. You just wrote instructions that someone (or something, like a computer) could follow. If you missed a step—like “open the jar”—you’d see the problem when they tried it. That’s coding: writing instructions, testing them, and fixing what doesn’t work.

What’s Ahead in This Book

This book is built for *you*. It’s not about memorizing formulas or passing tests. It’s about giving you the tools to create, solve problems, and feel powerful. Here’s what we’ll do together:

- **Learn the basics.** We’ll start with the building blocks of coding—things like variables (ways to store information) and loops (ways to repeat tasks). Don’t worry, we’ll explain everything in plain language.
- **Write real programs.** You’ll create things like a calculator, a game, and a journal app. Each project will teach you new skills and let you show off what you can do.
- **Think like a coder.** Coding is more than writing lines on a screen. It’s a way of seeing problems as puzzles you can solve, one piece at a time.
- **Grow your future.** We’ll talk about how coding can fit into your life—whether you want to build apps, automate tasks, or just have fun creating.

You don’t need a computer to start. Many of the exercises in this book can be done with pen and paper. If you do have access to a computer later, you’ll be ready to type your ideas into Python and see them come to life.

Your First Challenge: Dream Big

Before we jump into the nitty-gritty, let’s get inspired. Take a moment to think: If you could create *anything* with code, what would it be? A game? A tool to help someone? A story that changes based on what the user picks? Write down one idea, or just hold it in your mind. There are no wrong answers.

That idea is your starting point. As you move through this book, you’ll gain the skills to turn ideas like that into reality. You’re not just learning to code—you’re learning to build something that’s *yours*.

Ready? Let’s take the first step.

Reflection Prompt
Write or think about one problem in your life that you'd love to solve with a program. It could be something practical (like tracking time) or fun (like a game). Why does it matter to you?

Paper Coding Exercise
Write instructions for a simple task, like tying a shoe or organizing a bookshelf. Try to list every single step, as if the person following them has no idea what to do. This is your first "program"!

Chapter 2: Your Brain Is Wired for Logic

You're here because you're ready to learn something new. That's awesome. Before we write our first lines of Python code, let's talk about something you're already good at: thinking logically. Programming is all about giving clear instructions to solve problems, and your brain is already wired to do that. You've been solving problems your whole life—coding just gives you a new way to do it.

In this chapter, we'll explore how your everyday thinking is like programming. We'll use examples from your world to show how logic works, and we'll do some fun exercises to get your brain warmed up. No computer needed—just your imagination, a pen, and some paper.

Logic Is Everywhere

Logic is just a fancy word for figuring things out step by step. You do it all the time without even realizing it. Let's look at a few examples:

Making a plan. Say you're getting ready for the day. You decide: "If it's cold, I'll wear a jacket. If it's not, I'll skip it." That's logic—making a decision based on a condition.

Following a routine. Maybe you work out every morning: stretch, do push-ups, then jog. That's a sequence, a set of steps in order, just like a program.

Fixing something. If your radio stops working, you might check the batteries, then the wires, then the power button. That's troubleshooting—testing and adjusting until you solve the problem.

Programming is the same. It's about breaking a task into steps, making decisions, and tweaking things when they don't work. You've already got the mindset for this—you just need to learn how to translate it into code.

Thinking Like a Programmer

Programmers solve problems by thinking in a clear, organized way. Here are three key ideas that make up a programmer's mindset. Don't worry if they sound new; we'll practice them together.

Break it down. Big problems are easier when you split them into smaller pieces. Imagine cleaning a messy room: instead of tackling it all at once, you start with the clothes, then the books, then the trash. Coding works the same way.

Follow the flow. Programs are like recipes—they have a start, a middle, and an end. You decide what happens first, what comes next, and when to stop.

Learn by doing. Programmers don't get it right the first time. They try, make mistakes, and try again. Each mistake teaches you something new.

Let's try an exercise to see these ideas in action. Grab your pen and paper (or just think it through).

Exercise: Plan a Day

Write down the steps you'd take to plan a perfect day, from waking up to going to bed. Be specific, like you're explaining it to someone who's never done it before. For example:
Wake up at 7 AM.
Brush teeth.
If I'm hungry, eat breakfast; otherwise, drink water.
Read for 30 minutes.

Try to list at least five steps. If you want, add a decision (like "if I'm hungry") or a repeat (like "exercise for 10 minutes, three times"). When you're done, look at your list. That's a program! You just wrote a sequence of steps with decisions and repeats, just like a coder would.

Mistakes Are Your Friend

Here's a secret about coding: mistakes are part of the process. When you write a program, it might not work the first time. Maybe you forget a step, or the computer misunderstands your instructions. That's okay! Programmers call this debugging—finding and fixing errors.
Think about learning to ride a bike. You probably fell a few times before you got the hang of it. Each fall taught you something: balance better, steer smoother. Coding is the same. Every error is a clue that helps you get closer to a working program.

Let's try another exercise to practice this.

Exercise: Spot the Mistake

Below is a set of instructions for making a cup of instant coffee. But there's a mistake in the steps. Can you spot it? Write down or think about what's wrong and how you'd fix it.

Boil water in a kettle.
Pour water into a cup.
Add sugar to the cup.
Stir the coffee with a spoon.
Drink the coffee.

Did you catch it? The instructions skip adding the coffee itself! You'd end up with a cup of hot water and sugar—not exactly coffee. To fix it, you'd add a step between 3 and 4, like: "Add instant coffee to the cup." This is debugging—finding what's missing or wrong and making it right.

Logic in Action: A Real-World Example

Let’s put it all together with a bigger example. Imagine you’re organizing a small library (maybe books in your cell or a community room). You want to make it easy to find a book. Here’s how you might think through it logically, like a programmer:

Goal: Find a specific book quickly.
Steps:
Sort the books by title (A to Z).
Check the first letter of the book you want.
Look at the books starting with that letter.
If the book is there, grab it. If not, check the next letter.
Decisions: If the book’s title starts with “B,” do I skip the “A” books?
Repeats: Keep checking letters until I find the book or run out of books.

This process—sorting, checking, deciding—is exactly what a computer program does. In later chapters, you’ll learn how to write code that does something like this, but for now, notice how your brain already works this way.

Why This Matters for You

Learning to think logically isn’t just about coding. It’s about taking control of how you solve problems. Whether you’re planning your day, fixing a mistake, or dreaming up a new idea, logic helps you stay focused and move forward. And when you start writing Python code in the next chapter, you’ll see how these skills turn into programs that do real things.
You don’t need to be a math whiz or a tech expert. You already have the tools: your ability to think, plan, and try again. Coding just gives you a new way to use them.

Reflection Prompt

Think about a time you solved a problem by breaking it into steps or trying again after a mistake. What did you learn from it? How could that skill help you with coding?

Paper Coding Exercise

Write instructions for a simple task you do often, like getting ready for bed or organizing your space. Include at least one decision (e.g., “If it’s late, skip this step”) and one repeat (e.g., “Do this three times”). Then, check your instructions for mistakes. If you find one, write down how you’d fix it.

Chapter 3: Tools You Already Have

You're ready to dive deeper into coding, and here's the good news: you don't need a computer, a fancy degree, or a high-tech setup to start. You already have everything you need to learn programming right now. Your brain, a pen, some paper, and a bit of grit—that's your toolkit. In this chapter, we'll show you how to use those tools to start thinking and working like a programmer, even without a screen in front of you.
This chapter is all about building your confidence and proving to yourself that coding is something you can do, no matter where you are or what you have access to. Let's get started.

Your Most Powerful Tool: Your Mind

Your brain is the ultimate coding machine. It's been solving problems, making decisions, and learning new things your whole life. Programming is just a new way to use those skills. Think about it: when you figure out how to stretch a small budget, organize your day, or teach someone a new skill, you're already thinking like a coder. You're planning, breaking things down, and adjusting when things don't go as expected.

Here's what your mind brings to the table:
Curiosity. You're reading this book, so you're already curious. That's the spark that drives every coder to learn and create.
Logic. Like we talked about in Chapter 2, you're already wired to think step-by-step and make decisions. Coding just gives you a way to write those steps down clearly.
Persistence. Learning anything new takes trial and error. You've pushed through challenges before, and that grit will carry you far in programming.

You don't need to be a genius or a math expert. Coding is about clear thinking and trying again when things don't work. You've got that covered.
Pen and Paper: Your First Coding Environment
You might be thinking, "How can I code without a computer?" Easy—programming starts with ideas, and you can write those ideas down anywhere. A pen and paper are all you need to practice the logic and structure of coding. In fact, many programmers plan their code on paper before they ever touch a keyboard. They sketch out steps, draw diagrams, or write "fake code" (called pseudocode) to map out their ideas.

Here's how you can use pen and paper to learn coding:
Write instructions. Just like in Chapters 1 and 2, you can write step-by-step plans for tasks, like recipes or directions. This is how you practice breaking problems into smaller pieces.
Test your logic. You can "run" your program by imagining what would happen if someone followed your instructions exactly. If something goes wrong, you debug by fixing the steps.
Plan programs. You can design what a program will do—say, a game or a calculator—before you ever need a computer. This helps you focus on the big picture.

Let's try it with an exercise.

Exercise: Paper-Based Program

Grab a pen and paper (or just think it through). Imagine you're creating a "program" to help someone decide what to wear based on the weather. Write down the steps, including at least one decision (like "if it's raining"). Here's an example to get you started:

1. Check the weather outside.
2. If it's raining, grab a raincoat and boots.
3. If it's sunny, wear a t-shirt and sneakers.
4. If it's cold, add a jacket.
5. Get dressed and go outside.

Now, "run" your program in your head. Pretend you're following the steps exactly. Is anything missing? For example, what if it's both raining and cold? Could you add a step to handle that? Write down your fixed version. This is how coders plan and test their ideas.

Imagination: Your Testing Ground

Your imagination is another tool you already have. When you write a program on paper, you can "run" it by picturing what would happen. This is called mental simulation, and it's something programmers do all the time. It's like playing a movie in your head to see if your steps make sense.

For example, let's say you wrote a program to organize a meal plan for the week. You can imagine going through each step: picking meals, checking ingredients, writing a grocery list. If you realize you forgot to check for allergies, you can go back and add that step. This process—imagining, testing, and fixing—is exactly what you'll do when you write real Python code later.

Here's a quick exercise to practice mental simulation.

Exercise: Imagine a Program

Think of a simple task, like planning a short workout routine. Write down five steps for it, like:

1. Stretch for 5 minutes.
2. Do 10 push-ups.
3. Rest for 1 minute.
4. Do 10 sit-ups.
5. Repeat steps 2–4 two more times.

Now, close your eyes and imagine someone following your steps exactly. Did they understand everything? Did anything go wrong? For example, did you forget to say when to stop resting? Write down one way you'd improve your "program."

Persistence: The Key to Success

The final tool you have is persistence. Coding isn't about getting it right the first time—it's about trying, messing up, and trying again. Every coder, from beginners to experts, makes mistakes.

What makes them successful is that they don’t give up. They look at errors as clues, not failures.

Think about a time you learned something new, like cooking a dish or fixing something broken. You probably didn’t nail it on the first try. Maybe the food was too salty, or the fix didn’t hold. But you tried again, learned from what went wrong, and got better. Coding works the same way.

Here’s a story to prove it. When I was learning to code, I spent hours trying to make a simple program that asked someone’s name and said “Hello, [name]!” I kept getting errors because I forgot a tiny piece of the code (you’ll learn about this in Chapter 8). Instead of quitting, I read the error message, tried a fix, and tested it again. When it finally worked, I felt like I’d just won a championship. That’s what persistence does—it turns frustration into victory.

Getting Ready for Python

In the next chapter, we’ll start exploring Python, the programming language you’ll use to bring your ideas to life. But before we get there, know this: you’re already equipped to succeed. Your mind, your ability to write and plan, your imagination, and your persistence—these are the tools that will carry you through this book and beyond.

If you don’t have a computer right now, that’s okay. The exercises in this book are designed so you can practice on paper and in your head. If you do get access to a computer later, you’ll be ready to jump in with confidence, because you’ve already practiced the hard part: thinking like a coder.

Reflection Prompt

Think about a time you used one of your “tools” (your mind, persistence, or imagination) to solve a problem or learn something new. How did it feel when you succeeded? How can that experience help you as you learn to code?

Paper Coding Exercise

Write a “program” (a set of instructions) for a task you do regularly, like making a sandwich, cleaning a space, or planning your study time. Include at least one decision (e.g., “If I have bread, use it; otherwise, use a tortilla”) and one repeat (e.g., “Check each item twice”). Then, “run” it in your head and fix any mistakes you find.

Below is a draft of **Chapter 4: What Is a Program?** from *Code Without Bars: Learning Python from the Ground Up*, tailored for incarcerated learners or those without formal education. This chapter builds on the foundation laid in Chapters 1–3, introducing the concept of a program in a clear, relatable way. It uses everyday analogies, simple language, and paper-based exercises to make the material accessible, empowering readers to understand programming as a structured set of instructions. The tone remains encouraging and non-technical, with a focus on building confidence and preparing learners for Python coding in later chapters.

Chapter 4: What Is a Program?

You’ve been building your coder’s mindset—thinking logically, using your imagination, and tapping into your persistence. Now it’s time to answer a big question: What exactly is a *program*? Don’t worry if the word sounds technical. By the end of this chapter, you’ll see that programs are something you already understand, because you’ve been creating them in your head your whole life.

In this chapter, we’ll break down what a program is, how it works, and why it’s so powerful. You’ll also try some exercises to practice writing your own “programs” on paper, setting you up for coding in Python soon. No computer needed—just your brain, a pen, and some paper. Let’s dive in!

A Program Is a Set of Instructions

A program is like a recipe for your favorite dish. It’s a list of steps that, when followed, does something useful or cool. Just like a recipe tells you how to make a cake (“mix flour, add eggs, bake at 350°F”), a program tells a computer how to do something, like show a message, calculate a number, or play a game.

Here’s the key: a program is just a clear, step-by-step guide. The computer follows it exactly, like a super obedient helper who does what you say—but only if your instructions are clear. If you miss a step or mix up the order, the result might not be what you expected. That’s okay, though, because fixing those mistakes is part of the fun.

Let’s look at an example from your world. Imagine you’re teaching someone how to fold a paper airplane. Your instructions might look like this:

1. Take a sheet of paper.
2. Fold it in half lengthwise.
3. Unfold it so it’s flat again.
4. Fold the top two corners to the center line.
5. Fold the plane in half again.
6. Fold each side down to make wings.
7. Throw the plane gently.

Those steps are a program. If the person follows them exactly, they’ll get a paper airplane. If they skip a step or fold the wrong way, the plane might not fly. Programming is the same: you write steps, test them, and fix them if needed.

The Parts of a Program

Every program has three main parts: **input**, **processing**, and **output**. Let's break them down with an example you can relate to.

- **Input:** This is the information a program starts with. Think of it like the ingredients you gather before cooking. For example, if you're making a sandwich, the input is the bread, peanut butter, and jelly.
- **Processing:** This is the work the program does—the steps it follows. For the sandwich, processing is spreading the peanut butter, adding jelly, and putting the bread together.
- **Output:** This is the result of the program. In the sandwich example, the output is the finished sandwich, ready to eat.

Let's try another example. Imagine a program that helps you decide whether to go outside. Here's how it might work:

- **Input:** You check the weather (e.g., "It's raining").
- **Processing:** You decide: "If it's raining, take an umbrella. If it's sunny, wear sunglasses."
- **Output:** You grab the umbrella or sunglasses and head out.

When you write a program in Python, you'll tell the computer what input to use, what steps to follow, and what to show as the output. But for now, let's practice this idea on paper.

Exercise: Write a Paper Program
Grab a pen and paper (or just think it through). Write a "program" for a simple task, like making a cup of tea or organizing a small stack of books. Your program should have:
- At least one input (e.g., "Check if there's hot water" or "Know how many books there are").
- At least three processing steps (e.g., "Put a tea bag in a cup, pour hot water, let it steep").
- One output (e.g., "A cup of tea" or "A neat stack of books").

Here's an example for making tea:
- **Input:** Hot water, tea bag, cup.
- **Processing:**
  1. Place tea bag in cup.
  2. Pour hot water over tea bag.
  3. Wait 3 minutes for tea to steep.
- **Output:** A hot cup of tea.

After writing your program, read through it. Could someone follow it exactly and get the right result? If not, what step would you add or fix?

Programs Have a Flow

Programs don't just list steps—they have a *flow*, like a story with a beginning, middle, and end. The computer starts at the first step, moves to the next, and keeps going until it's done. Sometimes, the flow includes:

- **Decisions:** Like choosing between paths. For example, "If it's cold, wear a jacket; otherwise, wear a t-shirt."
- **Repeats:** Doing something over and over. For example, "Keep stirring the soup until it's hot."

This flow is what makes programs powerful. You can tell the computer to make decisions or repeat tasks without you having to do it manually every time.

Let's try an exercise to practice flow.

Exercise: Create a Program with Flow
Write a program (on paper or in your head) for planning a short study session. Include:
- At least one decision (e.g., "If I'm tired, take a 5-minute break").
- At least one repeat (e.g., "Read one page, then repeat for 5 pages").
- A clear output (e.g., "Finish studying 5 pages").

Example:
- **Input:** A book, a notebook, a pen.
- **Processing:**
  1. Read one page of the book.
  2. Write down one key idea in the notebook.
  3. If I'm tired, take a 5-minute break; otherwise, keep going.
  4. Repeat steps 1–3 for 5 pages.
- **Output:** 5 pages read, with notes written.

Check your program. Does the flow make sense? Would someone know when to stop or what to do if they're tired? Fix any gaps you find.

Why Programs Matter

Programs are everywhere. The alarm clock that wakes you up, the calculator you use for math, the games you might play—they're all programs. Someone wrote a set of instructions to make those things work, and soon, you'll be able to do that too.

What makes programs so powerful is that they let you automate tasks, solve problems, and create things that matter to you. In this book, you'll write programs to do things like:

- Calculate a budget.
- Make a quiz game.
- Keep a journal.

Even better, you'll learn to create programs that are *yours*—tools and ideas that come from your mind. And you don't need a computer to start. By practicing on paper, you're building the skills to write real Python code when the time comes.

Getting Ready for the Next Step

You've just learned what a program is: a set of instructions with input, processing, and output, flowing from start to finish. You've also practiced writing your own programs on paper, which means you're already thinking like a coder.

In the next chapter, we'll dig into *variables*—a way to store and use information in your programs. You'll see how to save things like numbers or names and use them to make your programs smarter. For now, keep practicing your paper programs and trust that you're building skills that will make Python feel like a natural next step.

Reflection Prompt
Think about a task you do often that could be turned into a program. What would the input, processing, and output be? How would it feel to have a computer do that task for you?

Paper Coding Exercise
Write a program for a task you'd love to automate, like tracking your exercise or planning a meal. Include:
- At least one input.
- At least one decision or repeat.
- A clear output.

Then, "run" your program in your head and fix any mistakes you find. For example, if your program is for tracking push-ups, make sure you included how to count them!

Chapter 5: Variables & Data

You’re doing great! You’ve learned that a program is a set of instructions with a flow, like a recipe or a plan. Now, let’s make your programs smarter by adding a way to store and use information. That’s where *variables* come in. Think of variables as labeled containers, like jars in a kitchen, where you can store things like numbers, words, or lists to use later in your program. In this chapter, we’ll explore what variables are, how they work, and how to use them to make your programs more powerful. You’ll practice with paper exercises, so no computer is needed—just your brain, a pen, and some paper.

What Is a Variable?

A variable is a way to save information in a program so you can use it later. Imagine you’re keeping track of how many push-ups you did today. You could write “25” on a piece of paper and label it “push-ups.” That label—“push-ups”—is like a variable. It holds the value (25) and gives it a name so you can refer to it easily.

In programming, variables let you store all kinds of information:
- Numbers, like how many push-ups you did (25).
- Words, like someone’s name (“Alex”).
- Lists, like a set of chores (“wash dishes, sweep floor, fold clothes”).

You give each variable a name that makes sense, and then you can use that name in your program to work with the information.

Here’s a real-world example. Let’s say you’re planning a budget. You might have:
- A variable called “money” that holds the value 50 (for $50).
- A variable called “food_cost” that holds the value 20 (for $20 spent on food).
- Your program could use these to calculate how much money you have left: 50 - 20 = 30.

Variables are like your program’s memory—they hold onto information so you can do something with it.

Naming Your Variables

When you create a variable, you give it a name that describes what it's for. A good name is clear and simple, like "age" for someone's age or "score" for a game score. Here are a few tips for naming variables:
- Use names that make sense, like "total_money" instead of just "x."
- Keep it short but clear, like "name" instead of "the_name_of_the_person."
- Stick to letters, numbers, and underscores (like "my_score_1"), and avoid spaces or special symbols like ! or @.

Let's try an exercise to practice naming variables.

Exercise: Name That Variable
Grab a pen and paper (or think it through). For each item below, come up with a good variable name to store that information:
- The number of books you've read this month.
- Your favorite color.
- A list of three foods you ate today.

Example answers:
- Books read: "books_read"
- Favorite color: "fav_color"
- List of foods: "today_foods"

Write down your variable names. Then, check: Are they clear? Would someone else know what they mean? If not, try a different name.

Using Variables in a Program

Variables are powerful because you can use them in your program's steps. Let's go back to the budget example. Imagine a program that tracks your money:
- **Input:** You start with $50, and you spend $20 on food.
- **Processing:**
  1. Store 50 in a variable called "money."
  2. Store 20 in a variable called "food_cost."
  3. Subtract "food_cost" from "money" to get the money left.
- **Output:** The money you have left (30).

On paper, you could write this like:
- money = 50
- food_cost = 20
- money_left = money - food_cost
- Output: money_left (30)

Notice how the variable names make it easy to understand what's happening. When you write Python code later, you'll use variables the same way, but for now, let's practice with another paper exercise.

Exercise: Paper Program with Variables
Write a program (on paper or in your head) to track how many hours you study in a week. Use variables to store information. Include:
- A variable for the hours you study each day (e.g., "daily_hours").
- A variable for the number of days (e.g., "days").
- A step to calculate total hours (e.g., daily_hours × days).
- An output showing the total hours.

Example:
- daily_hours = 2
- days = 5
- total_hours = daily_hours × days
- Output: total_hours (10)

Write your program, then "run" it in your head. Check if the math works. If you made a mistake (like forgetting to multiply), fix it. This is how programmers test their code!

Types of Data

Variables can hold different kinds of information, called *data types*. Here are the main ones you'll use in Python:
- **Numbers:** Like 25 (for push-ups) or 3.5 (for hours). These can be whole numbers (called *integers*) or decimals (called *floats*).
- **Words:** Called *strings* in programming, like "Alex" or "Monday." You can think of them as text.
- **Lists:** A collection of items, like ["apple," "banana," "orange"] or [10, 20, 30].

Each type is used for different things. For example, you'd use numbers for calculations (like adding money) and strings for names or messages (like "Hello, Alex!").

Let's try an exercise to practice data types.

Exercise: Pick the Data Type
For each piece of information below, decide if it's a number, a string, or a list. Write down your answer.
- Your age.
- A list of your three favorite songs.

- The name of your hometown.
- The number of push-ups you did today.

Example answers:
- Your age: Number
- Favorite songs: List
- Hometown: String
- Push-ups: Number

Check your answers. Did you pick the right type for each? If you're not sure, think: Would I add it (number), say it (string), or group it with other things (list)?

Why Variables Matter

Variables are like the building blocks of your programs. They let you store information, use it in different ways, and make your programs flexible. For example, if you write a program to greet someone, you can use a variable to store their name. Then, the program can say "Hello, Alex!" or "Hello, Maria!" without changing the steps—just the variable's value.

In the next chapter, we'll dig into *decisions* and *loops*, which let your programs make choices and repeat tasks. Variables will play a big role there, holding the information your program needs to decide or repeat. For now, keep practicing with variables on paper—you're getting closer to writing real Python code.

Reflection Prompt
Think about something you'd like to keep track of in a program (like your exercise, budget, or favorite quotes). What variables would you use, and what data types would they be?

Paper Coding Exercise
Write a program for a simple task, like calculating the total cost of two items or listing three things you did today. Include:
- At least two variables (e.g., "item1_cost" and "item2_cost").
- At least one processing step using the variables (e.g., add them together).
- A clear output (e.g., "Total cost: 15").
Then, check your program by "running" it in your head. Fix any mistakes you find.

Chapter 6: Decisions & Loops

You've got the basics down: programs are instructions, and variables let you store information like numbers or words. Now, let's make your programs smarter by adding two powerful ideas: *decisions* and *loops*. Decisions let your program choose what to do based on certain conditions, like picking a jacket if it's cold. Loops let your program repeat tasks, like doing push-ups multiple times. These are the tools that make programs flexible and useful, and you're already familiar with them from everyday life. In this chapter, we'll explore how decisions and loops work, and you'll practice writing them on paper to get ready for Python coding. Grab a pen and paper (or just think it through)—no computer needed!

What Are Decisions?

A decision in a program is like choosing what to do based on the situation. In real life, you make decisions all the time. For example, if you're hungry, you eat; if not, you wait. In programming, decisions work the same way: your program checks a condition (like "Is it raining?") and chooses what to do next (like "Grab an umbrella").

In programming, decisions often use the word *if*. Here's how it looks in plain English:
- If it's raining, take an umbrella.
- Otherwise, take sunglasses.

You can also add more choices:
- If it's raining, take an umbrella.
- Else if it's cold, wear a jacket.
- Otherwise, wear a t-shirt.

These decisions use variables to check conditions. For example, you might have a variable called "temperature" set to 60. Your program could decide: "If temperature is less than 65, wear a jacket."

Let's try an exercise to practice decisions.

Exercise: Write a Decision Program
Write a paper program to decide what to do based on the time of day. Use a variable (like "hour") and include at least two decisions. Here's an example:
- hour = 7
- If hour is less than 12, output "Good morning, time for breakfast!"
- Else if hour is less than 18, output "Good afternoon, time to study!"
- Otherwise, output "Good evening, time to relax!"

Write your own program. For example, you could decide what to wear or eat based on the time. Check your program: Does it cover all possible times? Fix any gaps you find.

What Are Loops?

A loop is a way to repeat something in a program. In life, you use loops when you do a task over and over, like washing dishes until the sink is empty or doing 10 push-ups. In programming, loops let you repeat steps without writing them out every time.

There are two main types of loops:
- **Counted loops:** Repeat a specific number of times, like “Do 5 push-ups.”
- **Conditional loops:** Repeat until something happens, like “Keep washing dishes until they’re all clean.”

Here’s an example of a counted loop in plain English:

```
- For 3 times, do:
  - Output “Do a push-up.”
```

This would print “Do a push-up” three times. A conditional loop might look like:

```
- While there are dishes in the sink, do:
  - Wash one dish.
```

Let’s practice with a loop exercise.

Exercise: Write a Loop Program
Write a paper program that uses a loop to repeat a task. Pick something simple, like reading pages in a book or doing jumping jacks. Include a variable (like “pages” or “jumps”) and either a counted or conditional loop. Example:

```
- jumps = 5
- For jumps times, do:
  - Output “Do a jumping jack.”
- Output “Workout done!”
```

Write your program. Then, “run” it in your head: count each step of the loop. Did it repeat the right number of times? Fix any mistakes.

Combining Decisions and Loops

Decisions and loops are even more powerful when you use them together. Imagine a program that helps you plan a study session:
- If you’re tired, take a break.
- Repeat studying for 5 pages.
- After each page, check if you’re tired.

Here’s how it might look in plain English:

```
- pages = 5
- tired = “no”
- For pages times, do:
  - Read one page.
  - If tired is “yes,” output “Take a 5-minute break.”
  - Otherwise, output “Keep going!”
- Output “Study session done!”
```

This program uses a loop to repeat reading and a decision to check if you’re tired. You’ll see this pattern a lot in Python, and it’s how programs do complex things like games or trackers.

Let’s try a bigger exercise to combine both.

Exercise: Program with Decisions and Loops
Write a paper program to track a workout routine. Use:
- A variable for the number of exercises (e.g., “exercises = 3”).
- A loop to repeat the exercises.
- A decision to check something, like energy level (e.g., “energy = high”).
- An output at the end.

Example:

```
- exercises = 4
- energy = “high”
- For exercises times, do:
  - If energy is “high,” output “Do 10 reps of push-ups.”
  - Else if energy is “low,” output “Do 5 reps of push-ups.”
  - Output “One exercise done.”
- Output “Workout complete!”
```

Write your program, then “run” it in your head. Check: Does the loop repeat the right number of times? Does the decision make sense? Fix any errors.

Why Decisions and Loops Matter

Decisions and loops are what make programs feel alive. They let your program react to different situations (like the weather or a user's input) and repeat tasks without you doing the work manually. In later chapters, you'll use these ideas in Python to make programs that:
- Ask a user a question and respond differently based on their answer.
- Keep running a game until someone wins.
- Track numbers or words over time.

For now, you're practicing the logic behind these ideas. By writing paper programs with decisions and loops, you're building the skills to write real Python code soon.

Reflection Prompt
Think of a task you do that involves repeating steps or making choices (like cooking or planning your day). How could a program with decisions and loops make it easier?

Paper Coding Exercise
Write a program for a task you'd like to automate, like checking chores or planning meals. Include:
- At least one variable (e.g., "meals = 3").
- A loop to repeat something (e.g., "Plan 3 meals").
- A decision to choose something (e.g., "If I have chicken, make chicken; otherwise, make rice").
- A clear output (e.g., "Meal plan done").

Run it in your head and fix any mistakes.

Chapter 7: Functions & Reuse

You're making great progress! You've learned how to store information with variables, make choices with decisions, and repeat tasks with loops. Now, let's talk about *functions*—a way to organize your program so you can reuse pieces of it without rewriting everything. Think of functions like a shortcut or a reusable recipe: you write it once, give it a name, and use it whenever you need it. In this chapter, we'll explore what functions are, how they work, and why they're so useful. You'll practice creating them on paper, so no computer is needed—just your brain, a pen, and some paper.

What Is a Function?

A function is a chunk of code that does a specific job, like calculating a total or printing a greeting. You give it a name, and then you can "call" it whenever you want that job done. It's like having a favorite sandwich recipe: instead of writing out "spread peanut butter, add jelly, put bread together" every time, you just say "make a sandwich," and the steps happen.

Here's a real-life example. Imagine you always greet your friends the same way: "Hey, [name], good to see you!" Instead of repeating that phrase every time, you could create a "greet" function:
- Name the function "greet."
- Define the steps: Say "Hey," add the person's name, add "good to see you!"
- Use it by saying "greet Alex" or "greet Maria."

In programming, functions save you time and make your programs easier to read. You write the steps once, and then you can use them over and over.

Let's try an exercise to get the idea.

Exercise: Design a Function
On paper (or in your head), create a function for a simple task, like saying a motivational phrase. Give it a name and list the steps it does. Example:
- Function name: motivate
- Steps:
  1. Output "You've got this!"
  2. Output "Keep pushing forward!"

Write your own function. For example, it could be for packing a bag or checking the weather. Then, write how you'd "call" it (e.g., "motivate" to run the function). Check: Do the steps make sense? Would they work every time you call the function?

How Functions Work

A function has two parts: the *definition* (where you write the steps) and the *call* (where you use it). Here’s how it looks in plain English for a function that calculates the total hours you’ve worked:

- **Definition:**
  - Function name: calculate_hours
  - Input: days (how many days you worked), hours_per_day (hours each day)
  - Steps:
    1. Multiply days by hours_per_day.
    2. Output the total hours.
- **Call:** calculate_hours with days = 5, hours_per_day = 4
- **Result:** Outputs 20 (because 5 × 4 = 20)

The input (like “days” and “hours_per_day”) is called a *parameter*. It’s like giving the function the ingredients it needs to do its job. You can call the function with different parameters to get different results, like calculate_hours with 3 days and 6 hours (18 hours).

Functions are powerful because they let you reuse code. If you need to calculate hours again, you don’t rewrite the steps—just call the function with new numbers.

Let’s practice with an exercise.

Exercise: Write a Function with Parameters
Write a paper function that does a simple calculation or task. Include at least one parameter.
Example:
- Function name: add_items
- Parameter: item1_cost, item2_cost
- Steps:
  1. Add item1_cost and item2_cost.
  2. Output the total cost.
- Call: add_items with item1_cost = 10, item2_cost = 5
- Result: Outputs 15

Write your own function, like one to count total exercises or plan meals for a number of days. Call it with specific values, then check: Does it give the right output? Fix any mistakes.

Using Functions with Decisions and Loops

Functions get even better when you combine them with decisions and loops (from Chapter 6). For example, imagine a function that checks if you're on track with a goal, like reading books:
- Function name: check_reading
- Parameter: pages_read, goal_pages
- Steps:
  - If pages_read is greater than or equal to goal_pages, output "You hit your goal!"
  - Otherwise, output "Keep reading, you're almost there!"
- Call: check_reading with pages_read = 10, goal_pages = 15
- Result: Outputs "Keep reading, you're almost there!"

Or, you could use a loop in a function to repeat a task:
- Function name: do_workout
- Parameter: reps
- Steps:
  - For reps times, output "Do a push-up."
  - Output "Workout done!"
- Call: do_workout with reps = 3
- Result: Outputs "Do a push-up" three times, then "Workout done!"

Let's try combining these ideas.

Exercise: Function with Decisions or Loops
Write a paper function that uses a decision or a loop (or both). Example:
- Function name: plan_study
- Parameter: pages, tired
- Steps:
  - For pages times, do:
    - If tired is "yes," output "Take a 5-minute break."
    - Otherwise, output "Read a page."
  - Output "Study done!"
- Call: plan_study with pages = 3, tired = "no"
- Result: Outputs "Read a page" three times, then "Study done!"

Write your own function, like one for tracking chores or planning a workout. Call it with specific values, then "run" it in your head. Check: Does the decision or loop work as expected? Fix any errors.

Why Functions Matter

Functions make your programs organized and reusable. Instead of writing the same steps over and over, you create a function once and call it whenever you need it. This saves time and makes your programs easier to fix or change. In Python, functions will let you build complex programs—like games or trackers—without getting overwhelmed.

Functions also let you break big problems into smaller pieces. For example, a program to plan a week might have one function for meals, another for exercise, and another for study. Each function handles one part, making the whole program easier to manage.

In the next chapter, we'll finally dive into Python itself. You'll see how to write real code for variables, decisions, loops, and functions. For now, keep practicing on paper—you're building the logic that will make Python feel natural.

Reflection Prompt
Think of a task you do often that could be a function, like greeting someone or checking a schedule. What would you name the function, and what steps would it include?

Paper Coding Exercise
Write a function for a task you'd like to reuse, like calculating total money spent or listing daily goals. Include:
- A clear function name.
- At least one parameter.
- A decision or loop (or both).
- A call with specific values.

Run it in your head and fix any mistakes. For example, if your function tracks money, make sure the calculations add up!

Chapter 8: Meet Python

You've built a solid foundation—writing programs on paper, using variables, decisions, loops, and functions. Now, it's time to meet *Python*, the programming language you'll use to turn your ideas into real, working programs. Python is like a friendly tool: it's easy to understand, powerful, and used by millions of people to create everything from games to websites. In this chapter, we'll explore what Python is, why it's great for beginners, and how to write your first lines of code. You can still practice on paper, so no computer is needed yet—just your brain, a pen, and some paper. Let's get started!

What Is Python?

Python is a programming language—a way to give instructions to a computer. Think of it like a language you use to talk to a friend who only understands clear, simple sentences. Python is special because it's designed to be easy to read and write, almost like plain English. For example, to print the words "Hello, world!" on the screen, you just write:

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

That's it! When you "run" this code (tell the computer to follow it), it shows "Hello, world!" on the screen. Compare that to other languages, which might use more complicated symbols or words. Python keeps it simple, which is why it's perfect for beginners like you.

Python is also powerful. It's used by scientists to analyze data, by game developers to build apps, and by everyday people to automate tasks. Whether you want to create a budget tracker or a quiz game, Python can make it happen.

Why Learn Python?

You might be wondering, "Why Python and not another language?" Here are a few reasons:
- **It's beginner-friendly.** Python's words and structure are clear, so you can focus on learning to code instead of memorizing weird symbols.
- **It's versatile.** You can use Python for small projects (like a calculator) or big ones (like a website).

- **It’s popular.** Learning Python connects you to a huge community of coders who share ideas and tools.

Even if you don’t have a computer right now, learning Python’s ideas on paper will prepare you to jump in when you do get access. And when you’re ready, you can run Python on a computer, phone, or even a library terminal.

Let’s try a paper exercise to get a feel for Python.

Exercise: Your First Python Program
Write a simple Python program on paper (or in your head) that prints a message. Use the `print` command, which tells the computer to show something on the screen. Example:
- Code: `print("Keep coding, you’re doing great!")`
- Result: Shows “Keep coding, you’re doing great!” when run.

Write your own program. For example, print your name or a motivational phrase. Make sure to use quotes around the words (Python calls this a *string*). Check: Did you write `print` correctly? Did you use quotes? Fix any mistakes.

Writing Python Code

Python code is written line by line, and each line is an instruction. Here’s how a small program might look:

```
name = "Alex"
print("Hello,")
print(name)
```

This program:
- Creates a variable called `name` and stores the string “Alex.”
- Prints “Hello,” on the screen.
- Prints the value of `name` (Alex).

When run, it shows:

```
Hello,
Alex
```

Notice how the variable `name` lets you store and reuse information, just like you practiced in Chapter 5. Python uses an equals sign (`=`) to put a value into a variable, like putting food into a labeled jar.

Let’s practice writing a Python program with variables.

Exercise: Python Program with Variables
Write a paper program that uses a variable and prints something. Example:
- Code:

```
  score = 10
  print("Your score is")
  print(score)
```

- Result: Shows “Your score is” and then “10” when run.

Write your own program. For example, store a number (like hours studied) or a string (like a goal) in a variable, then print it. Check: Did you use `=` to set the variable? Did you use `print` with quotes for words? Fix any errors.

Python’s Rules

Python has a few rules to keep things organized. Don’t worry—they’re simple, and you’ll get used to them. Here are the basics:
- **Case matters.** Python treats `Print` and `print` as different things. Always use lowercase for commands like `print`.
- **Quotes for strings.** Words or phrases (strings) must be in quotes, like `"Hello"` or `'Alex'`. Numbers, like `10`, don’t need quotes.
- **Indentation.** Python uses spaces to group steps together (like in loops or functions). You’ll see this later, but for now, keep each line starting at the left.
- **Comments.** You can add notes in your code using a `#` symbol. For example: `# This is a comment` doesn’t run—it’s just for you to explain things.

Here’s an example with all these rules:

```
# This program greets someone
name = "Maria"  # Store the name
print("Hi,")    # Print greeting
print(name)     # Print the name
```

Let’s try an exercise to practice these rules.

Exercise: Write a Commented Program
Write a Python program on paper that uses a variable, a `print` command, and a comment.
Example:
- Code:

```
# Program to track study time
hours = 3
print("You studied for")
print(hours)
print("hours today!")
```

- Result: Shows “You studied for”, “3”, “hours today!” when run.

Write your own program, like one to show a budget or a favorite quote. Include a comment (starting with `#`) to explain what it does. Check: Did you use quotes for strings? Is the variable set with `=`? Fix any mistakes.

Why Python Matters for You

Python is your gateway to creating things that matter to you. With just a few lines, you can make a program that tracks your goals, plays a game, or tells a story. In the coming chapters, you’ll use Python to:
- Make programs interactive by asking for input.
- Build calculators and games.
- Save and read information, like a journal.

Even without a computer, practicing Python on paper teaches you how to think like a coder. You’re learning the language’s rules and structure, so when you get to a computer, you’ll be ready to type and run your code.

Reflection Prompt
Imagine a small Python program you’d like to create, like greeting a friend or tracking a score. What would it do, and why would it be useful to you?

Paper Coding Exercise
Write a Python program on paper for a simple task, like showing a message or tracking a number. Include:
- A comment (with `#`) explaining the program.

- At least one variable (e.g., `goal = "Learn Python"` or `points = 25`).
- At least one `print` command to show something.

Run it in your head: imagine what the screen would show. Fix any mistakes, like missing quotes or wrong capitalization.

Chapter 9: Printing & Input

You’re now familiar with Python and how to write simple programs using variables and the `print` command. It’s time to make your programs more interactive by learning how to *print* messages clearly and get *input* from a user. Printing is how your program talks to the world, showing messages or results on the screen. Input is how the world talks back, letting users give information to your program, like their name or a number. In this chapter, we’ll explore how to use printing and input in Python to create programs that feel alive. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s dive in!

Printing Messages Clearly

The `print` command is your way to show information on the screen. You’ve already used it to display simple messages or variable values, like `print("Hello!")` or `print(score)`. But you can make your output more interesting by combining words, variables, and even formatting to make it clear and useful.

For example, instead of printing a number by itself, you can combine it with words:

```
score = 10
print("Your score is", score)
```

This prints: `Your score is 10`. Notice the comma in `print("Your score is", score)`—it tells Python to print the string and the variable together, with a space between them.

You can also print multiple things in one line using a special trick called an *f-string*. It looks like this:

```
name = "Alex"
print(f"Welcome, {name}!")
```

This prints: `Welcome, Alex!` The `f` before the string and the curly braces `{}` let you put a variable directly into the message.

Let’s practice printing with an exercise.

Exercise: Write a Print Program
Write a Python program on paper that uses `print` to show a message with a variable. Example:
- Code:

```
  # Program to show study time
  hours = 4
  print(f"You studied {hours} hours today!")
```

- Result: Shows `You studied 4 hours today!`

Write your own program. For example, print a message about a budget, a goal, or a favorite thing, using a variable. Try using an f-string (with `f` and `{}`). Check: Did you use quotes for strings? Did you include the variable correctly? Fix any mistakes.

Getting Input from Users

To make your program interactive, you can ask the user for information using the `input` command. The `input` command shows a message, waits for the user to type something, and stores their answer in a variable. Here's an example:

```
name = input("What is your name? ")
print("Hello,", name)
```

When you run this:
- The screen shows: `What is your name?`
- The user types “Maria” and presses Enter.
- The program stores “Maria” in the `name` variable and prints: `Hello, Maria`.

The `input` command always gives you a string (text), even if the user types a number. If you want to use the input as a number (like for math), you need to convert it using `int()` for whole numbers or `float()` for decimals. For example:

```
age = int(input("How old are you? "))
print("Next year, you’ll be", age + 1)
```

If the user types `25`, it prints: `Next year, you’ll be 26`.

Let’s try an exercise to practice input.

Exercise: Write an Input Program
Write a Python program on paper that uses `input` to ask the user something and then prints a response. Example:
- Code:

```
  # Program to greet a user
  name = input("What’s your name? ")
  print(f"Nice to meet you, {name}!")
```

- Result: Asks `What’s your name?`, then prints `Nice to meet you, [whatever they typed]!`

Write your own program. For example, ask for a favorite color, a number of hours worked, or a goal, then print a response using the input. Check: Did you store the input in a variable? Did you use `print` to show something back? Fix any errors.

Combining Printing and Input

Printing and input work together to make programs interactive. You can ask for input, store it in a variable, use it in your program (like with decisions or calculations), and then print the result. Here’s an example that combines input, a decision, and printing:

```
# Program to check study progress
hours = int(input("How many hours did you study today? "))
if hours >= 3:
    print("Great job, you’re on track!")
else:
    print("Keep going, add some more study time!")
```

This program:
- Asks for hours studied and stores it as a number (using `int()`).
- Uses a decision (`if` and `else`) to check the hours.
- Prints a message based on the input.

Notice the `if` and `else`—they’re Python’s way of making decisions, like you practiced in Chapter 6. You’ll learn more about them in Chapter 12, but for now, see how they make the program respond differently based on input.

Let’s practice combining these ideas.

Exercise: Interactive Program
Write a Python program on paper that uses input, a decision, and printing. Example:
- Code:

```
  # Program to check exercise
  reps = int(input("How many push-ups did you do? "))
```

```
if reps >= 10:
    print(f"Wow, {reps} push-ups! You’re strong!")
else:
    print(f"Good effort with {reps} push-ups. Try for more!")
```

- Result: Asks `How many push-ups did you do?`, then prints a message based on the number.

Write your own program. For example, ask for a budget amount and check if it’s enough, or ask for a goal and give feedback. Include:
- An `input` to ask for something.
- An `if`/`else` decision.
- A `print` with an f-string to show the result.

Run it in your head: imagine typing different inputs and check what the output would be. Fix any mistakes, like missing quotes or forgetting `int()` for numbers.

Why Printing and Input Matter

Printing and input are how your programs connect with people. Printing lets you show results, messages, or feedback, making your program useful. Input lets users talk to your program, making it interactive and personal. Together, they let you create programs that feel like a conversation, like:
- A quiz that asks questions and gives feedback.
- A tracker that takes your daily progress and shows a summary.
- A game that responds to your choices.

In the next chapter, we’ll explore how to work with numbers and strings in Python to do calculations and manipulate text. For now, keep practicing printing and input on paper—you’re building the skills to make real, interactive Python programs.

Reflection Prompt
Think of a program you’d like to make interactive, like a greeting app or a goal tracker. What would you ask the user, and what would you print back?

Paper Coding Exercise
Write a Python program on paper that:
- Uses `input` to ask the user for something (like a name, number, or goal).
- Uses a variable to store the input.
- Uses an `if`/`else` decision to check the input.
- Uses `print` with an f-string to show a response.

Example: Ask for hours worked, check if it’s over 5, and print a message like `Great work for {hours} hours!` Run it in your head with different inputs and fix any errors.

Chapter 10: Working with Numbers & Strings

You're now comfortable with printing messages and getting input from users in Python. It's time to make your programs even more powerful by learning how to work with *numbers* and *strings*. Numbers let you do math, like calculating totals or tracking scores. Strings let you handle text, like names or messages. In this chapter, we'll explore how to use numbers and strings in Python, combine them, and format your output to look clean and clear. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let's get started!

Working with Numbers

Numbers are a big part of programming. You can use them to calculate, compare, or track things like money, hours, or points. In Python, there are two main types of numbers:
- **Integers**: Whole numbers, like `5`, `100`, or `0`.
- **Floats**: Numbers with decimals, like `3.14`, `0.5`, or `10.0`.

You can do math with numbers using operators:
- `+` for addition (e.g., `5 + 3` equals `8`).
- `-` for subtraction (e.g., `10 - 4` equals `6`).
- `` *for multiplication (e.g.,* `2  3` equals `6`).
- `/` for division (e.g., `10 / 2` equals `5.0`).
- **`` for exponents (e.g., `2** 3` equals `8`, because 2 × 2 × 2 = 8).

Here's an example program that uses numbers:

```
hours = 4
days = 5
total_hours = hours * days
print("You worked", total_hours, "hours this week!")
```

This prints: `You worked 20 hours this week!`

If you get input as a number, remember to convert it using `int()` or `float()`. For example:

```
hours = int(input("How many hours did you study? "))
double_hours = hours * 2
print("Double your study time is", double_hours)
```

Let's practice with numbers.

Exercise: Number Program
Write a Python program on paper that does a simple calculation. Example:
- Code:

```
# Calculate total cost
item1 = 10
item2 = 5
total = item1 + item2
print(f"Total cost is ${total}")
```

- Result: Shows `Total cost is $15`

Write your own program, like one to calculate exercise reps, a budget, or study time. Use at least one math operation (`+`, `-`, ``*, or `/`). Check: Did your math work? Did you use `print` with an f-string? Fix any mistakes.*

Working with Strings

Strings are text in Python, like names, messages, or words, and they're always in quotes (`"Hello"` or `'Alex'`). You can use strings to store and show information or combine them to make messages. For example:

```
name = "Maria"
greeting = "Hello, " + name + "!"
print(greeting)
```

This prints: `Hello, Maria!` The `+` operator combines strings, but you need to add spaces yourself (like the space after `Hello,`).

You can also use f-strings for cleaner string formatting, like you saw in Chapter 9:

```
name = "Maria"
print(f"Hello, {name}!")
```

Strings can do more than just combine. You can make them uppercase or lowercase with methods like `.upper()` or `.lower()`:

```
word = "hello"
print(word.upper())  # Prints: HELLO
print(word.lower())  # Prints: hello
```

Let's practice with strings.

Exercise: String Program
Write a Python program on paper that uses a string. Example:
- Code:

```
  # Greet with uppercase
  name = input("What’s your name? ")
  big_name = name.upper()
  print(f"Hi, {big_name}!")
```

- Result: If user types `maria`, shows `Hi, MARIA!`

Write your own program, like one to show a favorite quote, a goal, or a name in uppercase. Use a string and either `+` or an f-string. Check: Did you use quotes for strings? Did you format the output clearly? Fix any errors.

Mixing Numbers and Strings

To make your programs useful, you'll often mix numbers and strings in your output. For example, you might want to show a number inside a message. F-strings are perfect for this:

```
points = 25
print(f"You earned {points} points!")
```

This prints: `You earned 25 points!`

If you get a number from `input`, you need to convert it to use it in math:

```
cost = float(input("How much was the item? "))
tax = cost * 0.1  # 10% tax
total = cost + tax
print(f"With tax, you pay ${total}")
```

If the user enters `20`, it calculates `20 + (20 *0.1) = 22` and prints: `With tax, you pay $22`.*

Let's try mixing numbers and strings.

Exercise: Mixed Program
Write a Python program on paper that uses both a number and a string. Example:
- Code:

```
# Track exercise
name = input("What’s your name? ")
reps = int(input("How many reps did you do? "))
print(f"Great job, {name}! You did {reps} reps!")
```

- Result: If user types `Alex` and `15`, shows `Great job, Alex! You did 15 reps!`

Write your own program, like one to track a budget or study hours. Use `input` for a string and a number, and print a message with both. Check: Did you convert number inputs with `int()` or `float()`? Did you use an f-string? Fix any mistakes.

Formatting Output

To make your programs look professional, you can format your output to be clear and neat. Here are two tips:
- **Use f-strings** for clean messages, like `f"Your total is ${total}"`.
- **Control numbers** for floats. For example, to show only two decimal places (like for money), use `{:.2f}`:

```
price = 19.999
print(f"Cost: ${price:.2f}")
```

This prints: `Cost: $19.99`

Let’s practice formatting.

Exercise: Formatted Program
Write a Python program on paper that formats output neatly. Example:
- Code:

```
# Calculate meal cost
meal = float(input("How much was the meal? "))
tip = meal * 0.15  # 15% tip
total = meal + tip
print(f"Meal: ${meal:.2f}, Tip: ${tip:.2f}, Total: ${total:.2f}")
```

- Result: If user types `20`, shows `Meal: $20.00, Tip: $3.00, Total: $23.00`

Write your own program, like one for a budget or score. Use a float and format it with `{:.2f}`. Check: Does the output look clean? Did you convert inputs correctly? Fix any errors.

Why Numbers and Strings Matter

Numbers and strings are the building blocks of most programs. Numbers let you calculate and track things, like totals or progress. Strings let you communicate with users through messages or names. By combining them, you can create programs that:
- Calculate budgets or exercise goals.
- Show personalized messages or feedback.
- Track and display information in a clear way.

In the next chapter, we'll explore *lists* and loops to work with collections of data, like a list of chores or scores. For now, keep practicing numbers and strings on paper—you're getting closer to building real Python programs!

Reflection Prompt
Think of a program you'd like to make that uses numbers and strings, like a budget tracker or a greeting app. What would it calculate or show?

Paper Coding Exercise
Write a Python program on paper that:
- Uses `input` to get a string (like a name) and a number (like hours or cost).
- Does a calculation with the number (like multiply or add).
- Uses an f-string to print a formatted message with both.
Example: Ask for a name and hours worked, calculate double hours, and print `"{name}, double your hours is {double_hours}!"` Run it in your head with different inputs and fix any errors.

Chapter 11: Lists & Loops

You're making great progress with Python, using numbers and strings to create interactive programs. Now, let's level up by learning about *lists* and how to combine them with *loops*. Lists let you store multiple pieces of information, like a set of names or numbers, in one place. Loops let you work with each item in a list without writing code for each one. Together, they make your programs more powerful and efficient. In this chapter, we'll explore how to create and use lists, loop through them, and combine them with other ideas like decisions. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let's dive in!

What Is a List?

A list in Python is like a shopping list or a to-do list—it holds multiple items in one place. Each item can be a number, string, or even another list. For example:

```
chores = ["wash dishes", "sweep floor", "fold clothes"]
scores = [10, 20, 30]
```

In these examples:
- `chores` is a list of strings.
- `scores` is a list of numbers.

You create a list by putting items inside square brackets `[]`, separated by commas. You can access a specific item using its position (called an *index*), starting at 0. For example:

```
chores = ["wash dishes", "sweep floor", "fold clothes"]
print(chores[0])  # Prints: wash dishes
print(chores[1])  # Prints: sweep floor
```

Let's practice creating and using lists.

Exercise: Create a List Program
Write a Python program on paper that creates a list and prints one item. Example:
- Code:

```
  # List of favorite foods
  foods = ["pizza", "tacos", "salad"]
  print("My favorite food is", foods[0])
```

- Result: Shows `My favorite food is pizza`

Write your own program with a list of at least three items (e.g., goals, names, or numbers). Print one item using its index (like `list[2]`). Check: Did you use `[]` for the list? Is the index correct (starting at 0)? Fix any mistakes.

Looping Through Lists

Loops let you work with every item in a list without writing separate code for each. In Python, a *for* loop is perfect for lists. Here's an example:

```
chores = ["wash dishes", "sweep floor", "fold clothes"]
for chore in chores:
    print("Do this:", chore)
```

This prints:

```
Do this: wash dishes
Do this: sweep floor
Do this: fold clothes
```

The `for` loop takes each item in the list, one by one, and runs the code inside the loop (like `print`). Notice the colon `:` and the indentation (spaces) under the loop—Python uses these to know which code belongs to the loop.

You can also use a loop to do something with each item, like add numbers:

```
scores = [10, 20, 30]
total = 0
for score in scores:
    total = total + score
print("Total score:", total)
```

This adds `10 + 20 + 30` and prints: `Total score: 60`

Let's practice looping.

Exercise: Loop Program
Write a Python program on paper that loops through a list. Example:

- Code:

```
# List of goals
goals = ["study", "exercise", "read"]
for goal in goals:
    print(f"Today’s goal: {goal}")
```

- Result: Shows:

```
Today’s goal: study
Today’s goal: exercise
Today’s goal: read
```

Write your own program with a list of at least three items. Use a `for` loop to print each item. Check: Did you use `for`, a colon, and indentation? Does the loop cover all items? Fix any errors.

Combining Lists, Loops, and Decisions

Lists and loops get even more powerful when you add decisions (from Chapter 9). For example, you can check each item in a list and do something based on a condition:

```
scores = [10, 25, 15]
for score in scores:
    if score >= 20:
        print(f"{score} is a high score!")
    else:
        print(f"{score} is okay, keep practicing!")
```

This prints:

```
10 is okay, keep practicing!
25 is a high score!
15 is okay, keep practicing!
```

Here, the loop goes through each score, and the `if`/`else` decision checks if it’s high (20 or more). Notice the indentation: the `if`/`else` is indented under the loop, and the `print` statements are indented under the `if`/`else`.

Let’s try combining these ideas.

Exercise: List with Decisions
Write a Python program on paper that uses a list, a loop, and a decision. Example:
- Code:

```
  # Check exercise reps
  reps = [5, 12, 8]
  for rep in reps:
      if rep >= 10:
          print(f"{rep} reps is awesome!")
      else:
          print(f"{rep} reps is good, try for more!")
```

- Result: Shows:

```
  5 reps is good, try for more!
  12 reps is awesome!
  8 reps is good, try for more!
```

Write your own program with a list of at least three items (e.g., hours studied, costs, or tasks). Use a `for` loop and an `if`/`else` to check each item. Check: Is the loop correct? Is the decision indented properly? Fix any mistakes.

Adding to Lists

You can add items to a list using the `.append()` method. For example:

```
tasks = ["study", "exercise"]
tasks.append("read")
print(tasks)
```

This prints: `["study", "exercise", "read"]`

You can use `input` to let users add to a list:

```
tasks = []
new_task = input("Add a task: ")
tasks.append(new_task)
print("Your tasks:", tasks)
```

If the user types `write`, it prints: `Your tasks: ["write"]`

Let's practice adding to lists.

Exercise: Append Program
Write a Python program on paper that creates a list and adds an item. Example:
- Code:

```
# Build a shopping list
shopping = ["milk", "bread"]
new_item = input("Add an item: ")
shopping.append(new_item)
print(f"Your list: {shopping}")
```

- Result: If user types `eggs`, shows `Your list: ["milk", "bread", "eggs"]`

Write your own program that starts with a list, uses `input` to get a new item, and appends it. Print the list. Check: Did you use `.append()`? Did you print the updated list? Fix any errors.

Why Lists and Loops Matter

Lists and loops let you handle multiple pieces of information at once, like tracking tasks, scores, or goals. They make your programs efficient because you don't need to write separate code for each item. With loops, you can process entire lists, and with decisions, you can make your program react differently to each item. This is how you'll build things like:
- A to-do list app that shows and updates tasks.
- A game that tracks multiple scores.
- A tracker that checks progress on goals.

In the next chapter, we'll dive deeper into decisions with `if`, `elif`, and `else` to make your programs even smarter. For now, keep practicing lists and loops on paper—you're building skills for real Python programs!

Reflection Prompt
Think of a program you'd like to make with a list, like tracking chores or favorite quotes. What would the list hold, and how would a loop use it?

Paper Coding Exercise
Write a Python program on paper that:

- Creates a list with at least three items.
- Uses a `for` loop to process the list.
- Includes an `if`/`else` decision for each item.
- Uses `.append()` to add a new item (e.g., from `input`).

Example: A program that tracks study hours, checks if each is enough, and adds a new hour.
Run it in your head and fix any errors.

Chapter 12: Conditions & Logic

You’ve learned how to use lists and loops to handle multiple pieces of information, and you’ve started making your programs interactive with input and output. Now, let’s make your programs even smarter by diving deeper into *conditions*—the way Python makes decisions. Conditions let your program choose what to do based on information, like deciding to wear a jacket if it’s cold or suggesting a break if you’ve studied for hours. In this chapter, we’ll explore how to use `if`, `elif`, and `else` to create flexible logic, and we’ll combine them with variables, lists, and loops. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s get started!

Understanding Conditions

A condition in Python is a way to check something and decide what to do next. You’ve already seen `if` and `else` in earlier chapters, like checking if a score is high enough. Conditions use *comparisons* to test if something is true or false. Here are the main comparison operators:
- `==` (equals): Checks if two things are the same, like `score == 10`.
- `!=` (not equals): Checks if two things are different, like `score != 0`.
- `>` (greater than), `<` (less than): For numbers, like `age > 18`.
- `>=` (greater than or equal to), `<=` (less than or equal to): Like `hours <= 5`.

Here’s a simple example:

```
age = int(input("How old are you? "))
if age >= 18:
    print("You’re an adult!")
else:
    print("You’re not an adult yet.")
```

If the user enters `20`, it prints: `You’re an adult!` If they enter `16`, it prints: `You’re not an adult yet.`

Let’s practice writing conditions.

Exercise: Simple Condition Program
Write a Python program on paper that uses a condition. Example:
- Code:

```
  # Check study time
  hours = int(input("How many hours did you study? "))
  if hours >= 3:
      print("Great job studying!")
  else:
      print("Try to study a bit more!")
```

- Result: If user enters `4`, shows `Great job studying!`; if `2`, shows `Try to study a bit more!`

Write your own program, like one to check a score, budget, or exercise reps. Use `if` and `else` with a comparison (like `>=` or `==`). Check: Did you convert input with `int()` if needed? Does the condition make sense? Fix any errors.

Using `elif` for Multiple Choices

Sometimes, you need more than two options (if or else). That's where `elif` (short for "else if") comes in. It lets you check additional conditions. For example:

```
temperature = int(input("What’s the temperature? "))
if temperature > 80:
    print("It’s hot, wear shorts!")
elif temperature > 60:
    print("It’s nice, a t-shirt is fine.")
elif temperature > 40:
    print("It’s chilly, grab a jacket.")
else:
    print("It’s cold, bundle up!")
```

This checks conditions in order: if one is true, it runs that block and skips the rest. If none are true, it runs the `else`.

Let's practice using `elif`.

Exercise: Multi-Condition Program
Write a Python program on paper that uses `if`, `elif`, and `else`. Example:
- Code:

```
  # Check exercise reps
  reps = int(input("How many push-ups did you do? "))
  if reps >= 20:
      print("Wow, you’re super strong!")
  elif reps >= 10:
      print("Great effort, solid workout!")
  else:
      print("Good start, keep practicing!")
```

- Result: If user enters `25`, shows `Wow, you’re super strong!`; if `15`, shows `Great effort, solid workout!`; if `5`, shows `Good start, keep practicing!`

Write your own program, like one to check hours studied or money saved. Use at least one `elif`. Check: Are the conditions in the right order? Did you indent properly? Fix any mistakes.

Combining Conditions with Logic

You can make conditions more powerful by combining them with *logical operators*:
- `and`: Both conditions must be true, like `age > 18 and score >= 90`.
- `or`: At least one condition must be true, like `day == "Saturday" or day == "Sunday"`.
- `not`: Reverses a condition, like `not score == 0`.

Example:

```
hours = int(input("How many hours did you study? "))
mood = input("Are you tired? (yes/no) ")
if hours >= 3 and mood == "no":
    print("You’re killing it, keep going!")
else:
    print("Take a break and try again!")
```

This only prints “You’re killing it” if *both* conditions are true: hours are 3 or more *and* mood is “no.”

Let’s practice combining conditions.

Exercise: Logical Condition Program
Write a Python program on paper that uses `and` or `or`. Example:
- Code:

```
  # Check workout
  reps = int(input("How many reps did you do? "))
  energy = input("Energy level (high/low)? ")
  if reps >= 10 and energy == "high":
      print("Amazing workout!")
  else:
      print("Good effort, rest up!")
```

- Result: If user enters `12` and `high`, shows `Amazing workout!`; otherwise, shows `Good effort, rest up!`

Write your own program, like one to check study time or budget with two conditions (using `and` or `or`). Check: Are the logical operators correct? Does the logic make sense? Fix any errors.

Conditions with Lists and Loops

Conditions are even more powerful when combined with lists and loops (from Chapter 11). You can loop through a list and make decisions for each item. Example:

```
scores = [15, 25, 10]
for score in scores:
    if score >= 20:
        print(f"{score} is a high score!")
    else:
        print(f"{score} needs improvement.")
```

This prints:

```
15 needs improvement.
25 is a high score!
10 needs improvement.
```

You can also use input to build a list and check conditions:

```
numbers = []
for i in range(3):  # Loop 3 times
    num = int(input("Enter a number: "))
    numbers.append(num)
for num in numbers:
    if num > 10:
        print(f"{num} is big!")
    else:
        print(f"{num} is small.")
```

Let's combine these ideas.

Exercise: List and Conditions Program
Write a Python program on paper that uses a list, a loop, and conditions. Example:

- Code:

```
# Check study hours
hours_list = []
for i in range(3):
    hours = int(input("Hours studied today? "))
    hours_list.append(hours)
for hours in hours_list:
    if hours >= 2:
        print(f"{hours} hours is solid!")
    else:
        print(f"{hours} hours, try for more!")
```

- Result: If user enters `3`, `1`, `4`, shows:

```
3 hours is solid!
1 hours, try for more!
4 hours is solid!
```

Write your own program, like one to check reps or costs. Use a loop to build a list with `input` and another loop with conditions to check each item. Check: Is the loop correct? Are conditions indented? Fix any errors.

Why Conditions Matter

Conditions let your programs think and react, like a friend who gives advice based on the situation. They make your programs flexible, so they can handle different inputs and scenarios. With conditions, you can:
- Give personalized feedback in a quiz.
- Track goals and suggest improvements.
- Make games that respond to player choices.

In the next chapter, we'll explore *functions* in Python to organize your code and make it reusable. For now, keep practicing conditions on paper—you're building the logic for smarter Python programs!

Reflection Prompt
Think of a program you'd like to make that uses conditions, like a goal checker or a decision app. What conditions would you check, and what would the program do?

Paper Coding Exercise

Write a Python program on paper that:

- Uses `input` to get at least one piece of information.
- Uses `if`, `elif`, and `else` (or `and`/`or`) for conditions.
- Prints a response based on the conditions.

Example: Ask for hours and mood, check both with `and`, and print feedback. Run it in your head with different inputs and fix any errors.

Chapter 13: Functions & Modules

You’ve learned how to make your Python programs interactive with conditions, lists, loops, and input. Now, let’s organize your code and make it more powerful with *functions* and *modules*. Functions let you group code into reusable chunks, like a recipe you can use over and over. Modules are like toolkits, giving you extra features to make your programs do more. In this chapter, we’ll explore how to create and use functions in Python, introduce modules, and show how they save time and effort. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s dive in!

What Is a Function in Python?

A function is a named block of code that does a specific job, like calculating a total or printing a message. You define it once and then “call” it whenever you need it. This saves you from rewriting the same code multiple times. In Chapter 7, you practiced functions on paper; now, you’ll write them in Python.

Here’s a simple function:

```
def greet(name):
    print(f"Hello, {name}!")
```

- `def` tells Python you’re defining a function.
- `greet` is the function’s name.
- `name` is a parameter (input the function uses).
- The indented lines are the function’s steps.
- You call it with `greet("Alex")`, which prints: `Hello, Alex!`

Let’s practice writing a function.

Exercise: Simple Function Program
Write a Python program on paper that defines and calls a function. Example:
- Code:

```
# Define a function to show motivation
def motivate(goal):
    print(f"You can achieve your {goal}!")
motivate("Python coding")
```

- Result: Shows `You can achieve your Python coding!`

Write your own function, like one to print a message or calculate something (e.g., double a number). Include a parameter and call the function. Check: Did you use `def`, a colon, and indentation? Does the call match the parameter? Fix any errors.

Functions with Return Values

Functions can also give back a result using the `return` statement. This lets you use the function's output in other parts of your program. For example:

```
def add_numbers(num1, num2):
    total = num1 + num2
    return total

result = add_numbers(5, 3)
print("The sum is", result)
```

This prints: `The sum is 8`. The function adds `num1` and `num2`, returns the total, and stores it in `result`.

You can use `return` with conditions or loops:

```
def check_score(score):
    if score >= 20:
        return "High score!"
    else:
        return "Keep practicing!"

message = check_score(25)
print(message)
```

This prints: `High score!`

Let's practice functions with `return`.

Exercise: Return Function Program
Write a Python program on paper that defines a function with `return`. Example:
- Code:

```
  # Function to double reps
  def double_reps(reps):
      return reps * 2
  result = double_reps(10)
  print(f"Double your reps is {result}")
```

- Result: Shows `Double your reps is 20`

Write your own function that returns something, like a calculation (e.g., total hours) or a message based on a condition. Call it and print the result. Check: Does the function return a value? Is the call correct? Fix any mistakes.

Combining Functions with Lists and Loops

Functions work great with lists and loops to process multiple items. For example:

```
def check_goals(goals):
    for goal in goals:
        if len(goal) > 5:  # Check if goal name is long
            print(f"{goal} is a big goal!")
        else:
            print(f"{goal} is a good start!")

tasks = ["study", "exercise", "read book"]
check_goals(tasks)
```

This prints:

```
study is a good start!
exercise is a big goal!
read book is a big goal!
```

Here, the function loops through a list and uses a condition to check each item.

Let’s practice combining these ideas.

Exercise: Function with List Program
Write a Python program on paper that uses a function with a list and loop. Example:
- Code:

```
  # Function to check hours
  def check_hours(hours_list):
      for hours in hours_list:
          if hours >= 3:
              print(f"{hours} hours is solid!")
          else:
              print(f"{hours} hours, add more!")
  study_hours = [2, 4, 1]
  check_hours(study_hours)
```

- Result: Shows:

```
2 hours, add more!
4 hours is solid!
1 hours, add more!
```

Write your own function that takes a list (e.g., scores, tasks) and loops through it with a condition. Call it with a sample list. Check: Is the loop indented? Does the function work for all items? Fix any errors.

What Are Modules?

A module is like a toolbox that gives you extra functions to use in Python. Python comes with many built-in modules, like `math` for calculations or `random` for generating random numbers. To use a module, you *import* it at the start of your program.

For example, the `math` module has a function called `sqrt` to calculate the square root:

```
import math
number = 16
result = math.sqrt(number)
print(f"Square root of {number} is {result}")
```

This prints: `Square root of 16 is 4.0`

The `random` module can pick a random item from a list:

```
import random
choices = ["rock", "paper", "scissors"]
pick = random.choice(choices)
print(f"Random pick: {pick}")
```

This prints a random choice, like `Random pick: paper`.

Let's practice using a module.

Exercise: Module Program
Write a Python program on paper that uses a module like `math` or `random`. Example:
- Code:

```
# Use math module
import math
number = 25
root = math.sqrt(number)
print(f"The square root of {number} is {root}")
```

- Result: Shows `The square root of 25 is 5.0`

Write your own program using `math` (e.g., `math.floor(3.7)` rounds down to `3`) or `random` (e.g., `random.choice(["yes", "no"])`). Check: Did you import the module? Did you use the dot (`.`) to call the function? Fix any errors.

Why Functions and Modules Matter

Functions make your code organized and reusable, so you don't repeat yourself. Modules give you powerful tools without having to write them from scratch. Together, they let you:
- Build complex programs, like games or trackers, by breaking them into small functions.
- Use built-in tools to do math, pick random items, or handle dates.
- Keep your code clean and easy to fix.

In the next chapter, we'll tackle *errors* and *debugging* to help you fix problems when your code doesn't work. For now, keep practicing functions and modules on paper—you're building skills for real Python programs!

Reflection Prompt
Think of a program you'd like to make with a function, like calculating totals or picking a random task. What would the function do, and would you use a module?

Paper Coding Exercise
Write a Python program on paper that:
- Defines a function with a parameter and `return`.

- Uses a list and loop with a condition inside the function.
- Uses a module (like `math` or `random`) for one task.

Example: A function that takes a list of numbers, checks if they’re big, and uses `math.sqrt`. Run it in your head and fix any errors.

Chapter 14: Errors & Debugging

You’re making awesome progress with Python, building programs with functions, lists, conditions, and modules. But here’s a truth about coding: things don’t always work the first time. Errors happen to every programmer, from beginners to experts. The good news? Errors are just clues telling you what needs fixing, and *debugging* is the process of finding and solving those problems. In this chapter, we’ll explore common errors in Python, learn how to read error messages, and practice fixing them. You can do this on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s dive in!

What Are Errors?

An error is when your program doesn’t run as expected. Maybe it crashes, shows the wrong output, or does nothing at all. In Python, errors often come with a message that hints at what went wrong. Think of it like a puzzle: the error message is a clue, and debugging is solving it.

There are three main types of errors:
- **Syntax Errors**: Mistakes in the code’s structure, like forgetting a colon or quotes.
- **Runtime Errors**: Problems that happen when the program runs, like dividing by zero or using a variable that doesn’t exist.
- **Logic Errors**: When the program runs but gives the wrong result, like adding instead of subtracting.

Here’s an example of a syntax error:

```
print("Hello)  # Missing a quote
```

Python might say: `SyntaxError: unexpected EOF while parsing`. This means it expected a closing quote (`"`) but didn’t find one.

Let’s practice spotting errors.

Exercise: Spot the Syntax Error
Look at this program and find the mistake. Write the corrected version on paper.
- Code:

```
  # Print a greeting
  name = "Alex"
  print("Hi, name)  # What's wrong?
```

- Result: Won’t run due to a syntax error.

Answer: The missing quote after `name`. Corrected:

```
print("Hi, name")  # Fixed with closing quote
```

Write your own program with a deliberate syntax error (e.g., missing colon, wrong indentation). Then, fix it. Check: Does the fixed version look like valid Python? Fix any other mistakes you spot.

Reading Error Messages

When Python finds an error, it gives you a message with details. These messages can seem confusing, but they’re like a map to the problem. For example:

```
age = int(input("How old are you? "))
if age > 18
    print("Adult!")
```

This has a syntax error (missing colon after `if age > 18`). Python might say: `SyntaxError: invalid syntax at line 2`. The “line 2” part tells you where to look.

For a runtime error, consider:

```
number = int(input("Enter a number: "))
result = 10 / number
```

If the user enters `0`, Python says: `ZeroDivisionError: division by zero`. This means you tried to divide by zero, which isn’t allowed.

Let’s practice reading errors.

Exercise: Find the Error Type
Look at each program and decide if it’s a syntax, runtime, or logic error. Write the problem and how to fix it.
1.

```
score = 10
print(score + " points")  # What’s wrong?
```

2.

```
   total = 5 + 3
   print(total - 2)  # Should subtract, but adds
```

Answers:
1. Runtime error: Can't add a number and string. Fix: Use an f-string, like `print(f"{score} points")`.
2. Logic error: Adds instead of subtracts. Fix: Change to `print(total)` or correct the logic.

Write your own program with a deliberate error (syntax, runtime, or logic). Identify the error type and fix it. Check: Did you explain the error clearly? Does the fix work?

Debugging Step-by-Step

Debugging is like being a detective. Here's a simple process to find and fix errors:
1. **Read the error message**: Look for the type (e.g., `SyntaxError`) and line number.
2. **Check the code**: Look at the line mentioned and nearby lines for mistakes.
3. **Test with small changes**: Fix one thing at a time and imagine running the code.
4. **Use print for clues**: Add `print` statements to see what's happening in your program.

Example with a logic error:

```
# Should check if score is passing
score = int(input("Enter your score: "))
if score > 60:
    print("You failed!")
else:
    print("You passed!")
```

This prints the wrong message (logic error). Add `print` to debug:

```
score = int(input("Enter your score: "))
print("Score is", score)  # Check the value
if score > 60:
    print("You failed!")
else:
    print("You passed!")
```

If you enter `70`, it prints `Score is 70` and `You failed!`—clearly wrong. Fix by swapping the messages:

```
if score > 60:
    print("You passed!")
else:
    print("You failed!")
```

Let’s practice debugging.

Exercise: Debug a Program
Here’s a program with errors. Find and fix them on paper.
- Code:

```
  # Calculate total cost
  cost = input("Enter cost: ")
  tax = cost * 0.1
  print("Total with tax:", cost + tax)
```

- Problems:
  - `input` gives a string, but `cost` needs to be a number for math.
  - No f-string for clean output.

Fix it, then write your own program with at least one error (e.g., wrong operator, missing quotes). Debug it by adding a `print` statement and fixing the issue. Check: Does the fixed program work? Did `print` help find the error?

Common Errors to Watch For

Here are some mistakes you might see and how to fix them:
- **Missing colon (`:`)**: After `if`, `for`, or `def`. Fix: Add the colon.
- **Wrong indentation**: Lines under a loop or condition need 4 spaces. Fix: Align the indentation.
- **Using a string for math**: Like `cost * 0.1` when `cost` is a string. Fix: Use `int()` or `float()`.
- **Wrong variable name**: Like `print(scor)` instead of `score`. Fix: Check spelling.
- **Logic mistakes**: Like using `>` instead of `<`. Fix: Test the condition with sample values.

Let’s practice fixing common errors.

Exercise: Fix Common Errors

Write a program with at least two common errors (e.g., missing colon, wrong variable name).
Example:
- Code with errors:

```
# Check hours
hours = int(input("Hours studied? "))
if hours > 2  # Missing colon
    print("Good job, hours!")  # Wrong variable
```

- Fixed:

```
if hours > 2:
    print(f"Good job, {hours}!")
```

Write your own program with errors, then fix them. Check: Did you catch all mistakes? Does the fixed version make sense?

Why Debugging Matters

Errors are a normal part of coding—they're how you learn. Debugging teaches you to think critically, test ideas, and improve your programs. With practice, you'll spot errors faster and fix them like a pro. This skill will help you:
- Build reliable programs, like trackers or games.
- Understand how Python works by seeing what breaks it.
- Stay confident even when things go wrong.

In the next chapter, we'll start building real projects, like a calculator, using everything you've learned. For now, keep practicing debugging on paper—you're getting ready to create awesome Python programs!

Reflection Prompt
Think of a time you fixed something that wasn't working (like a plan or a task). How can that problem-solving help you debug code?

Paper Coding Exercise
Write a Python program on paper with at least two deliberate errors (e.g., syntax and logic). Example: A program to check a budget with a missing colon and wrong math. Debug it by:
- Writing the error message you'd expect.

- Adding a `print` statement to check values.
- Fixing the errors.

Run it in your head and confirm the fixed version works.

Chapter 15: Build a Calculator

You’ve learned the building blocks of Python—variables, conditions, loops, functions, and debugging. Now, it’s time to put those skills together to create something real: a *calculator* program. This project will let users input numbers and choose operations (like addition or subtraction), and the program will show the result. It’s a great way to practice everything you’ve learned while building something useful. You can write and test this on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s get started!

Planning the Calculator

Before writing code, let’s plan what the calculator will do:
- **Input**: Ask the user for two numbers and an operation (like add, subtract, multiply, or divide).
- **Processing**: Use conditions to check the operation and calculate the result.
- **Output**: Show the result in a clear, formatted way.
- **Features**: Handle basic math operations and check for errors (like dividing by zero).

Here’s a rough idea in plain English:
- Ask for the first number.
- Ask for the second number.
- Ask for the operation (+, -, _, /)._
- If the operation is +, add the numbers.
- If the operation is -, subtract them.
- If the operation is _, multiply them._
- If the operation is /, divide them (but check for zero).
- Print the result.

This uses variables, input, conditions, and math—perfect for practicing your skills.

Writing the Calculator Program

Let’s write the Python code for a simple calculator. Here’s a complete program:

```
# Simple Calculator
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
operation = input("Choose operation (+, -, *, /): ")
```

```
if operation == "+":
    result = num1 + num2
    print(f"{num1} + {num2} = {result}")
elif operation == "-":
    result = num1 - num2
    print(f"{num1} - {num2} = {result}")
elif operation == "*":
    result = num2 * num2
    print(f"{num1} * {num2} = {result}")
elif operation == "/":
    if num2 == 0:
        print("Error: Cannot divide by zero!")
    else:
        result = num1 / num2
        print(f"{num1} / {num2} = {result:.2f}")
else:
    print("Invalid operation! Use +, -, *, or /")
```

This program:
- Uses `float()` to handle decimals.
- Asks for two numbers and an operation.
- Uses `if`/`elif`/`else` to choose the right calculation.
- Checks for division by zero to avoid errors.
- Formats division output to two decimal places with `{:.2f}`.

Let’s test it in your head:
- Input: `num1 = 10`, `num2 = 5`, `operation = "+"`
- Output: `10 + 5 = 15`
- Input: `num1 = 10`, `num2 = 0`, `operation = "/"`
- Output: `Error: Cannot divide by zero!`

Let’s practice writing part of it.

Exercise: Basic Calculator Program
Write a Python program on paper for a calculator with just *addition* and *subtraction*. Example:
- Code:

```
  # Add or subtract calculator
  num1 = float(input("Enter first number: "))
  num2 = float(input("Enter second number: "))
  op = input("Choose + or -: ")
  if op == "+":
      print(f"Result: {num1 + num2}")
  elif op == "-":
      print(f"Result: {num1 - num2}")
  else:
      print("Use + or - only!")
```

- Result: If user enters `6`, `4`, `+`, shows `Result: 10`

Write your own program for addition and subtraction. Check: Did you use `float()` for numbers? Are conditions indented? Run it in your head with different inputs and fix any errors.

Adding a Function

To make the calculator reusable, let’s put the calculations in a function. This way, you can call it multiple times without rewriting the code. Here’s an example:

```
def calculate(num1, num2, operation):
    if operation == "+":
        return num1 + num2
    elif operation == "-":
        return num1 - num2
    elif operation == "*":
        return num1 * num2
    elif operation == "/":
        if num2 == 0:
            return "Error: Cannot divide by zero!"
        return num1 / num2
    else:
        return "Invalid operation!"

# Main program
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
operation = input("Choose operation (+, -, *, /): ")
result = calculate(num1, num2, operation)
print(f"Result: {result}")
```

This program:
- Defines a `calculate` function that takes three parameters: `num1`, `num2`, and `operation`.
- Returns the result or an error message.
- Calls the function and prints the result.

Let’s practice with a function.

Exercise: Calculator Function

Write a Python program on paper that uses a function for addition and multiplication. Example:
- Code:

```
# Add or multiply function
def calc(num1, num2, op):
    if op == "+":
        return num1 + num2
    elif op == "*":
        return num1 * num2
    else:
        return "Invalid operation!"
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
op = input("Choose + or *: ")
print(f"Result: {calc(num1, num2, op)}")
```

- Result: If user enters `3`, `4`, ``*, shows `Result: 12`*

Write your own program with a function for two operations (e.g., subtraction and division). Include a zero check for division. Check: Does the function return the right value? Is the call correct? Fix any errors.

Making It Interactive with a Loop

To make the calculator even better, let's add a loop so users can keep calculating until they choose to stop. Here's an example:

```
while True:
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))
    operation = input("Choose operation (+, -, *, /) or 'q' to quit: ")
    if operation == "q":
        print("Goodbye!")
        break
    result = calculate(num1, num2, operation)
    print(f"Result: {result}")
```

This program:
- Uses a `while True` loop to keep running.
- Lets users quit by typing `q` (using `break` to exit the loop).
- Calls the `calculate` function from above.

Let’s practice adding a loop.

Exercise: Looping Calculator
Write a Python program on paper that uses a loop to let users calculate multiple times.
Example:
- Code:

```
# Looping add/subtract calculator
def calc(num1, num2, op):
    if op == "+":
        return num1 + num2
    elif op == "-":
        return num1 - num2
    else:
        return "Use + or -!"
while True:
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))
    op = input("Choose + or - (or 'q' to quit): ")
    if op == "q":
        print("Done!")
        break
    print(f"Result: {calc(num1, num2, op)}")
```

- Result: Keeps asking for inputs until user types `q`.

Write your own program with a loop and a function for two operations. Include a way to quit.
Check: Does the loop stop with `break`? Is the function indented correctly? Fix any errors.

Why This Project Matters

Building a calculator shows you how to combine Python’s tools—input, conditions, functions, and loops—to create something useful. It’s a step toward real programs like budget trackers or games. You’ve also practiced debugging by checking for errors like division by zero. This project teaches you:
- How to structure a program with functions.
- How to handle user input and choices.
- How to make programs interactive with loops.

In the next chapter, we’ll build a quiz game, using lists and conditions to ask questions and track scores. For now, keep practicing your calculator on paper—you’re creating real Python programs!

Reflection Prompt
What other features could you add to your calculator, like new operations or a history of calculations? How would that make it more useful?

Paper Coding Exercise
Write a Python program on paper for a calculator that:
- Uses a function to handle at least three operations (e.g., +, -,).
- Includes a loop to keep calculating until the user quits.
- Checks for errors (like division by zero).
- Uses f-strings for clear output.
Run it in your head with different inputs (e.g., valid and invalid operations) and fix any errors.

Chapter 16: Create a Quiz Game

You’ve built a calculator and mastered combining Python’s tools like functions, loops, conditions, and lists. Now, let’s create something fun and interactive: a *quiz game*. This project will let users answer questions, track their score, and get feedback, using everything you’ve learned. It’s a great way to practice organizing code and handling user input. You can write and test this on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s get started!

Planning the Quiz Game

Before writing code, let’s plan what the quiz game will do:
- **Input**: Ask the user a series of questions and get their answers.
- **Processing**: Check if each answer is correct, keep track of the score, and provide feedback.
- **Output**: Show the questions, feedback for each answer, and the final score.
- **Features**: Use a list to store questions and answers, loops to ask them, and conditions to check correctness.

Here’s the idea in plain English:
- Store questions and correct answers in a list.
- Loop through each question, show it, and get the user’s answer.
- Check if the answer is correct, add to the score if it is.
- Show feedback for each question and the final score.

This will use lists, loops, conditions, input, and printing—perfect for practicing your skills.

Writing the Quiz Game Program

Let’s write a simple quiz game with a few questions. Here’s a complete program:

```
# Quiz Game
questions = [
    ["What is 2 + 2? ", "4"],
    ["What color is the sky? ", "blue"],
    ["How many days in a week? ", "7"]
]
score = 0

for question, correct_answer in questions:
    user_answer = input(question)
    if user_answer.lower() == correct_answer.lower():
        print("Correct!")
        score = score + 1
    else:
        print(f"Wrong! The answer is {correct_answer}")
print(f"Your final score: {score} out of {len(questions)}")
```

This program:
- Uses a list of lists, where each inner list has a question and its answer.
- Loops through the questions, asking each one with `input`.
- Checks answers with `if` (using `.lower()` to ignore case, like “Blue” vs. “blue”).
- Tracks the score and prints the final result.

Let’s test it in your head:
- For question “What is 2 + 2?”, user answers “4” → “Correct!”, score = 1.
- For “What color is the sky?”, user answers “red” → “Wrong! The answer is blue”, score stays 1.
- Final output: `Your final score: 1 out of 3`.

Let’s practice writing part of it.

Exercise: Basic Quiz Program
Write a Python program on paper for a quiz with two questions. Example:
- Code:

```
# Simple quiz
questions = [["What is 5 - 3? ", "2"], ["What is a cat? ", "animal"]]
score = 0
for q, a in questions:
    answer = input(q)
    if answer == a:
        print("Right!")
        score = score + 1
    else:
        print(f"No, it’s {a}")
print(f"Score: {score}/2")
```

- Result: If user answers “2” and “animal”, shows `Right!` twice and `Score: 2/2`.

Write your own quiz with two questions. Use a list of lists, a loop, and conditions. Check: Does the loop ask each question? Is the score tracked correctly? Fix any errors.

Adding a Function

To make the quiz reusable, let’s put the answer-checking in a function. This keeps the code organized and lets you reuse it for different quizzes. Here’s an example:

```
def check_answer(user_answer, correct_answer):
    if user_answer.lower() == correct_answer.lower():
        print("Correct!")
        return 1
    else:
        print(f"Wrong! The answer is {correct_answer}")
        return 0

# Main quiz program
questions = [
    ["What is 10 / 2? ", "5"],
    ["What’s the capital of France? ", "Paris"]
]
score = 0
for question, answer in questions:
    user_input = input(question)
    score = score + check_answer(user_input, answer)
print(f"Your score: {score} out of {len(questions)}")
```

This program:
- Defines a `check_answer` function that checks if the answer is correct and returns 1 (correct) or 0 (wrong).
- Uses the function in a loop to check answers and update the score.
- Prints the final score.

Let’s practice with a function.

Exercise: Quiz Function Program
Write a Python program on paper with a function to check answers. Example:
- Code:

```
  # Quiz with function
  def is_correct(user, correct):
      if user == correct:
          print("Good job!")
          return 1
      else:
          print(f"Sorry, it’s {correct}")
          return 0
  questions = [["What is 3 + 4? ", "7"], ["What’s a dog’s sound? ", "bark"]]
  score = 0
  for q, a in questions:
      ans = input(q)
      score = score + is_correct(ans, a)
  print(f"Score: {score}/2")
```

- Result: If user answers “7” and “bark”, shows `Good job!` twice and `Score: 2/2`.

Write your own quiz with a function to check answers. Use two questions and a loop. Check: Does the function return the right value? Is the score updated? Fix any errors.

Making It Interactive with a Loop

To make the quiz more engaging, let’s add a loop so users can play again. Here’s an example:

```
def check_answer(user_answer, correct_answer):
    if user_answer.lower() == correct_answer.lower():
        print("Correct!")
        return 1
    else:
        print(f"Wrong! The answer is {correct_answer}")
        return 0

while True:
    questions = [
        ["What is 2 * 3? ", "6"],
        ["What’s the opposite of up? ", "down"]
    ]
    score = 0
    for question, answer in questions:
        user_input = input(question)
        score = score + check_answer(user_input, answer)
    print(f"Your score: {score} out of {len(questions)}")
    play_again = input("Play again? (yes/no): ")
    if play_again.lower() == "no":
        print("Thanks for playing!")
        break
```

This program:
- Uses a `while True` loop to keep the quiz running.
- Asks if the user wants to play again and stops with `break` if they say “no.”
- Resets the score each time.

Let’s practice adding a loop.

Exercise: Looping Quiz Program
Write a Python program on paper for a quiz that lets users play again. Example:
- Code:

```
# Looping quiz
def check(ans, correct):
    if ans == correct:
        print("Nice!")
        return 1
    else:
        print(f"No, it’s {correct}")
        return 0
while True:
    questions = [["What is 1 + 1? ", "2"], ["What’s a bird? ", "animal"]]
    score = 0
    for q, a in questions:
        ans = input(q)
        score = score + check(ans, a)
    print(f"Score: {score}/2")
    again = input("Play again? (y/n): ")
    if again == "n":
        print("Bye!")
        break
```

- Result: Runs quiz, shows score, and repeats until user enters `n`.

Write your own quiz with a function, two questions, and a play-again loop. Check: Does the loop restart or stop correctly? Is the score reset? Fix any errors.

Why This Project Matters

The quiz game shows you how to combine Python’s tools—lists, loops, functions, and conditions—to create an interactive program. It’s a step toward real-world apps like educational tools or games. You’ve practiced:
- Storing data in lists.
- Using functions to organize code.
- Making programs interactive with loops and input.
- Debugging by checking answers and handling case.

In the next chapter, we’ll build a journal app to save and read text, introducing file handling. For now, keep practicing your quiz game on paper—you’re creating real Python programs!

Reflection Prompt
What kind of quiz would you make, like math, trivia, or personal goals? How would it help you or others?

Paper Coding Exercise
Write a Python quiz program on paper that:
- Uses a list with at least three questions.
- Has a function to check answers and return a score.
- Includes a loop to play again.
- Uses f-strings for clear feedback.

Run it in your head with different answers and fix any errors.

Chapter 17: Make a Journal App

You've built a calculator and a quiz game, combining Python's tools like functions, loops, lists, and conditions. Now, let's create something personal and practical: a *journal app*. This project will let users write entries, save them, and read them back later. It introduces *file handling*, which lets your program store information even after it stops running. You'll use input, lists, and loops to manage entries, and we'll keep it simple for paper-based practice. No computer is needed—just your brain, a pen, and some paper. Let's get started!

Planning the Journal App

Before writing code, let's plan what the journal app will do:
- **Input**: Ask the user for journal entries and commands (like write, read, or quit).
- **Processing**: Store entries in a list and save them to a file, or read entries from the file.
- **Output**: Show saved entries or confirm actions (like "Entry saved!").
- **Features**: Allow adding entries, viewing all entries, and quitting, with a menu to choose actions.

Here's the idea in plain English:
- Show a menu: "Write entry, read entries, or quit."
- If the user chooses to write, get their entry and save it.
- If they choose to read, show all saved entries.
- If they choose to quit, stop the program.
- Store entries in a list and save them to a file.

This uses input, loops, conditions, lists, and file handling—great for practicing your skills.

Writing the Journal App Program

Let's write a simple journal app that stores entries in a list and simulates saving them to a file. Since you're practicing on paper, we'll treat the list as the "file" for now. Here's a complete program:

```
# Journal App
entries = []

while True:
    print("1. Write entry")
    print("2. Read entries")
    print("3. Quit")
    choice = input("Choose an option (1-3): ")

    if choice == "1":
        entry = input("Write your journal entry: ")
        entries.append(entry)
        print("Entry saved!")
    elif choice == "2":
        if len(entries) == 0:
            print("No entries yet.")
        else:
            print("Your entries:")
            for i, entry in enumerate(entries, 1):
                print(f"{i}. {entry}")
    elif choice == "3":
        print("Journal closed. Goodbye!")
        break
    else:
        print("Invalid choice! Use 1, 2, or 3.")
```

This program:
- Uses a list (`entries`) to store journal entries.
- Runs a `while True` loop to show a menu and get user choices.
- For choice `1`, takes an entry with `input` and adds it to the list with `append`.
- For choice `2`, loops through the list and shows entries with numbers (using `enumerate` to count).
- For choice `3`, stops with `break`.
- Checks for invalid choices and empty lists.

Let's test it in your head:
- Choose `1`, enter "Today was good" → "Entry saved!"
- Choose `2` → Shows "Your entries: 1. Today was good"
- Choose `3` → "Journal closed. Goodbye!"

Let's practice writing part of it.

Exercise: Basic Journal Program
Write a Python program on paper for a journal with just *write* and *read* options. Example:
- Code:

```
# Simple journal
entries = []
while True:
    print("1. Write  2. Read")
```

```
      choice = input("Choose 1 or 2: ")
      if choice == "1":
          entry = input("Enter your thought: ")
          entries.append(entry)
          print("Saved!")
      elif choice == "2":
          if len(entries) == 0:
              print("Nothing saved yet.")
          else:
              for entry in entries:
                  print(entry)
      else:
          print("Choose 1 or 2!")
```

- Result: Saves entries and shows them, or shows “Nothing saved yet.”

Write your own program with write and read options. Use a list and loop. Check: Does the loop show the menu? Does `append` work? Fix any errors.

Adding File Handling (Simulated)

In a real Python program, you can save entries to a file using `open`, `write`, and `read`. Since you’re on paper, we’ll simulate file handling by assuming the `entries` list is the file. Here’s how it would look with real file handling:

```
# Journal with file (simulated)
entries = []

while True:
    print("1. Write entry")
    print("2. Read entries")
    print("3. Quit")
    choice = input("Choose an option (1-3): ")

    if choice == "1":
        entry = input("Write your journal entry: ")
        entries.append(entry)
        # Simulate saving to file: "write" to entries list
        print("Entry saved to journal!")
    elif choice == "2":
        if len(entries) == 0:
            print("No entries in journal.")
        else:
            print("Reading journal:")
            for i, entry in enumerate(entries, 1):
                print(f"{i}. {entry}")
    elif choice == "3":
        print("Saving and closing journal!")
        break
    else:
        print("Invalid choice!")
```

In real Python, you'd use:
- `with open("journal.txt", "a") as file: file.write(entry + "\n")` to save.
- `with open("journal.txt", "r") as file: print(file.read())` to read.

For now, the `entries` list acts like the file.

Let's practice this.

Exercise: Simulated File Journal
Write a Python program on paper that simulates saving and reading entries. Example:
- Code:

```
# Simulated journal
journal = []
while True:
    print("1. Write  2. Read  3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "1":
        entry = input("Your entry: ")
        journal.append(entry)
        print("Saved to journal!")
    elif choice == "2":
        if len(journal) == 0:
            print("Empty journal.")
        else:
            for i, entry in enumerate(journal, 1):
                print(f"Entry {i}: {entry}")
    elif choice == "3":
        print("Journal closed!")
        break
    else:
        print("Choose 1, 2, or 3!")
```

- Result: Saves entries in `journal` and shows them numbered.

Write your own program with a menu, list, and simulated file handling. Check: Does the menu loop? Are entries saved and shown correctly? Fix any errors.

Adding a Function

To make the code cleaner, let’s use a function to handle saving or reading. Here’s an example:

```
def manage_journal(entries, choice):
    if choice == "1":
        entry = input("Write your journal entry: ")
        entries.append(entry)
        return "Entry saved!"
    elif choice == "2":
        if len(entries) == 0:
            return "No entries yet."
        result = "Your entries:\n"
        for i, entry in enumerate(entries, 1):
            result = result + f"{i}. {entry}\n"
        return result
    else:
        return "Invalid choice!"

# Main program
entries = []
while True:
    print("1. Write entry")
    print("2. Read entries")
    print("3. Quit")
    choice = input("Choose an option (1-3): ")
    if choice == "3":
        print("Journal closed!")
        break
    print(manage_journal(entries, choice))
```

This program:
- Uses a `manage_journal` function to handle writing and reading.
- Returns messages or entry lists to print.
- Keeps the main loop simple.

Let’s practice with a function.

Exercise: Journal with Function
Write a Python program on paper with a function to handle journal actions. Example:
- Code:

```
  # Journal with function
```

```
def journal_action(entries, choice):
    if choice == "1":
        entry = input("Your entry: ")
        entries.append(entry)
        return "Saved!"
    elif choice == "2":
        if len(entries) == 0:
            return "Nothing saved."
        return "\n".join([f"{i}. {e}" for i, e in enumerate(entries, 1)])
    return "Invalid!"
journal = []
while True:
    print("1. Write  2. Read  3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "3":
        print("Done!")
        break
    print(journal_action(journal, choice))
```

- Result: Saves entries and shows them, or stops with `3`.

Write your own program with a function for journal actions. Check: Does the function handle both options? Is the loop correct? Fix any errors.

Why This Project Matters

The journal app shows you how to build a program that stores and retrieves data, a key skill for real-world apps like diaries or trackers. You've practiced:
- Using lists to store data.
- Creating a menu with loops and conditions.
- Organizing code with functions.
- Simulating file handling for persistence.

In the next chapter, we'll build a number guessing game, adding randomness and more interactivity. For now, keep practicing your journal app on paper—you're creating real Python programs!

Reflection Prompt
What would you use a journal app for, like tracking thoughts or goals? How would saving entries help you?

Paper Coding Exercise

Write a Python journal program on paper that:

- Uses a list to store entries.
- Has a menu with write, read, and quit options.
- Uses a function to handle at least one action.
- Shows numbered entries when reading.

Run it in your head with sample entries and fix any errors.

Chapter 18: Build a Number Guessing Game

You’ve created a calculator and a journal app, using Python’s tools like functions, loops, lists, and conditions. Now, let’s make something fun and challenging: a *number guessing game*. In this game, the computer picks a random number, the user tries to guess it, and the program gives hints to guide them. This project introduces the `random` module for generating random numbers and combines all your skills to create an engaging game. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s get started!

Planning the Number Guessing Game

Before writing code, let’s plan what the game will do:
- **Input**: The user guesses a number.
- **Processing**: The program picks a random number, checks the guess, and gives hints (like “too high” or “too low”).
- **Output**: Show hints for each guess and a win message when the user guesses correctly.
- **Features**: Use a loop to keep asking for guesses, track the number of attempts, and let the user play again.

Here’s the idea in plain English:
- Pick a random number between 1 and 100.
- Ask the user for a guess.
- If the guess is too high or too low, give a hint and ask again.
- If the guess is correct, show a win message with the number of attempts.
- Ask if they want to play again.

This uses input, loops, conditions, and the `random` module—perfect for practicing your skills.

Writing the Number Guessing Game

Let’s write a Python program for a simple guessing game. We’ll use the `random` module to pick a number. Here’s a complete program:

```
import random

while True:
    secret_number = random.randint(1, 100)
    attempts = 0
    print("I’m thinking of a number between 1 and 100!")

    while True:
        guess = int(input("Your guess: "))
        attempts = attempts + 1
        if guess == secret_number:
            print(f"You got it! It took {attempts} guesses.")
            break
        elif guess > secret_number:
            print("Too high! Try again.")
        else:
            print("Too low! Try again.")

    play_again = input("Play again? (yes/no): ")
    if play_again.lower() == "no":
        print("Thanks for playing!")
        break
```

This program:
- Imports the `random` module and uses `random.randint(1, 100)` to pick a number.
- Uses a `while True` loop to start new games.
- Tracks guesses with `attempts` and checks them with `if`/`elif`/`else`.
- Breaks the inner loop when the guess is correct and the outer loop if the user quits.

Let’s test it in your head:
- Secret number is 42. User guesses `50` → “Too high!” (attempts = 1).
- User guesses `30` → “Too low!” (attempts = 2).
- User guesses `42` → “You got it! It took 3 guesses.”
- User enters “no” to quit → “Thanks for playing!”

Let’s practice writing part of it.

Exercise: Basic Guessing Game
Write a Python program on paper for a guessing game with a fixed range (1–10). Example:
- Code:

```
# Simple guessing game
import random
secret = random.randint(1, 10)
attempts = 0
print("Guess a number between 1 and 10!")
while True:
    guess = int(input("Your guess: "))
    attempts = attempts + 1
    if guess == secret:
        print(f"Correct! Took {attempts} tries.")
        break
    elif guess > secret:
        print("Too high!")
    else:
        print("Too low!")
```

- Result: If secret is 7 and user guesses 8, 6, 7, shows “Too high!”, “Too low!”, “Correct! Took 3 tries.”

Write your own program for a 1–10 range. Include hints and track attempts. Check: Does the loop stop when correct? Is `random` imported? Fix any errors.

Adding a Function

To make the game cleaner, let’s use a function to check guesses. This keeps the code organized and reusable. Here’s an example:

```
import random

def check_guess(guess, secret):
    if guess == secret:
        return "correct"
    elif guess > secret:
        return "too high"
    else:
        return "too low"

while True:
    secret_number = random.randint(1, 20)
    attempts = 0
    print("Guess my number (1-20)!")
    while True:
        guess = int(input("Your guess: "))
        attempts = attempts + 1
        result = check_guess(guess, secret_number)
        if result == "correct":
            print(f"You win! It took {attempts} guesses.")
            break
        else:
            print(f"Hint: {result}")
    play_again = input("Play again? (y/n): ")
    if play_again.lower() == "n":
        print("Game over!")
        break
```

This program:
- Defines a `check_guess` function to compare the guess and secret number.
- Returns a string (“correct”, “too high”, or “too low”) to control the game.
- Uses two loops: one for guesses, one for new games.

Let’s practice with a function.

Exercise: Guessing Game with Function
Write a Python program on paper with a function to check guesses. Example:
- Code:

```
# Guessing game with function
import random
def check(guess, secret):
    if guess == secret:
        return "correct"
    elif guess > secret:
        return "high"
    else:
        return "low"
secret = random.randint(1, 10)
attempts = 0
print("Guess a number (1-10)!")
while True:
    guess = int(input("Guess: "))
    attempts = attempts + 1
    result = check(guess, secret)
    if result == "correct":
        print(f"Got it in {attempts} tries!")
        break
    print(f"Too {result}!")
```

- Result: If secret is 5 and user guesses 7, 3, 5, shows “Too high!”, “Too low!”, “Got it in 3 tries!”

Write your own program with a function for a 1–10 range. Check: Does the function return the right result? Is the loop correct? Fix any errors.

Adding Error Handling

To make the game robust, let’s handle bad inputs, like non-numbers. Here’s an example:

```
import random

def check_guess(guess, secret):
    if guess == secret:
        return "correct"
    elif guess > secret:
        return "too high"
    else:
        return "too low"

while True:
    secret_number = random.randint(1, 50)
    attempts = 0
    print("Guess my number (1-50)!")
    while True:
        try:
            guess = int(input("Your guess: "))
            if guess < 1 or guess > 50:
                print("Please enter a number between 1 and 50!")
                continue
            attempts = attempts + 1
            result = check_guess(guess, secret_number)
            if result == "correct":
                print(f"You win! It took {attempts} guesses.")
                break
            else:
                print(f"Hint: {result}")
        except ValueError:
            print("Enter a number, not text!")
    play_again = input("Play again? (y/n): ")
    if play_again.lower() == "n":
        print("Thanks for playing!")
        break
```

This program:
- Uses `try`/`except` to catch invalid inputs (like letters instead of numbers).
- Checks if guesses are in the valid range (1–50).
- Uses `continue` to ask again for bad inputs.

Let's practice error handling.

Exercise: Guessing Game with Error Handling
Write a Python program on paper with error handling. Example:
- Code:

```
  # Safe guessing game
```

```
import random
secret = random.randint(1, 10)
attempts = 0
print("Guess a number (1-10)!")
while True:
    try:
        guess = int(input("Guess: "))
        attempts = attempts + 1
        if guess == secret:
            print(f"Correct in {attempts} tries!")
            break
        elif guess > secret:
            print("Too high!")
        else:
            print("Too low!")
    except ValueError:
        print("Numbers only!")
```

- Result: If user enters “abc”, shows “Numbers only!”; if 7, 3, 5 (secret = 5), shows “Too high!”, “Too low!”, “Correct in 3 tries!”

Write your own program for a 1–10 range with `try`/`except`. Check: Does it handle bad inputs? Does the game work? Fix any errors.

Why This Project Matters

The number guessing game combines Python’s tools—random numbers, loops, functions, conditions, and error handling—to create a fun, interactive program. It’s a step toward real-world apps like games or quizzes. You’ve practiced:
- Using the `random` module for unpredictability.
- Handling user input and errors.
- Organizing code with functions and loops.
- Giving clear feedback with hints.

In the next chapter, we’ll explore *dictionaries* to store more complex data, like player stats or question banks. For now, keep practicing your game on paper—you’re building real Python programs!

Reflection Prompt
What would make your guessing game more fun, like a smaller range, more hints, or a score system? How would you add that?

Paper Coding Exercise

Write a Python guessing game on paper that:

- Uses `random.randint` for a 1–20 range.
- Has a function to check guesses.
- Includes `try`/`except` for error handling.
- Lets users play again with a loop.

Run it in your head with sample guesses and fix any errors.

Chapter 19: Dictionaries & Data

You've built a calculator, a quiz game, and a number guessing game, mastering Python's tools like loops, functions, and lists. Now, let's explore *dictionaries*, a powerful way to store and organize data. Dictionaries let you pair keys with values, like a phonebook linking names to numbers. They're perfect for tracking complex information, like player stats or question banks. In this chapter, we'll learn how to create and use dictionaries, combine them with loops and conditions, and apply them to a small project. You can practice on paper, so no computer is needed—just your brain, a pen, and some paper. Let's dive in!

What Is a Dictionary?

A dictionary in Python is like a real dictionary: it pairs a *key* (like a word) with a *value* (like its definition). Instead of numbered positions like lists, dictionaries use keys to find values. For example:

```
player = {"name": "Alex", "score": 50, "level": 3}
print(player["name"])  # Prints: Alex
print(player["score"])  # Prints: 50
```

Here:
- The dictionary is in curly braces `{}`.
- Keys (`name`, `score`, `level`) are paired with values (`"Alex"`, `50`, `3`) using colons.
- You access a value with square brackets, like `player["name"]`.

Dictionaries can hold strings, numbers, or even lists as values:

```
grades = {"math": 90, "english": 85, "tests": [88, 92]}
```

Let's practice creating a dictionary.

Exercise: Create a Dictionary Program
Write a Python program on paper that creates a dictionary and prints one value. Example:
- Code:

```
  # Player stats
  stats = {"name": "Maria", "points": 100}
  print(f"Player: {stats['name']}, Points: {stats['points']}")
```

- Result: Shows `Player: Maria, Points: 100`

Write your own program with a dictionary of at least two key-value pairs (e.g., goals, stats, or tasks). Print one value using a key. Check: Did you use `{}`, colons, and correct key names? Fix any errors.

Modifying Dictionaries

You can add, change, or remove items in a dictionary:
- **Add or update**: Assign a value to a key.

```
  player = {"name": "Alex"}
  player["score"] = 10  # Add new key-value
  player["name"] = "Sam"  # Update existing key
  print(player)  # Prints: {'name': 'Sam', 'score': 10}
```

- **Remove**: Use `pop()` or `del`.

```
  player = {"name": "Alex", "score": 10}
  player.pop("score")  # Removes score
  print(player)  # Prints: {'name': 'Alex'}
```

You can also get user input to update a dictionary:

```
stats = {}
name = input("Enter player name: ")
stats["name"] = name
print(f"Added: {stats}")
```

Let’s practice modifying dictionaries.

Exercise: Modify Dictionary Program
Write a Python program on paper that creates and modifies a dictionary. Example:
- Code:

```
  # Update stats
  stats = {"goals": 5}
  stats["wins"] = 2  # Add new key
  stats["goals"] = 6  # Update existing key
  print(f"Stats: {stats}")
```

- Result: Shows `Stats: {'goals': 6, 'wins': 2}`

Write your own program that creates a dictionary, adds one key-value pair, and updates another. Print the result. Check: Did you use correct syntax for adding/updating? Fix any errors.

Looping Through Dictionaries

You can loop through a dictionary to access its keys, values, or both. Here’s how:
- **Loop through keys**:

```
stats = {"name": "Alex", "score": 50}
for key in stats:
    print(f"{key}: {stats[key]}")
```

  Prints:

```
name: Alex
score: 50
```

- **Loop through values**: Use `.values()`.

```
for value in stats.values():
    print(value)
```

  Prints: `Alex`, `50`
- **Loop through key-value pairs**: Use `.items()`.

```
for key, value in stats.items():
    print(f"{key} is {value}")
```

  Prints:

```
name is Alex
score is 50
```

Let’s practice looping.

Exercise: Dictionary Loop Program
Write a Python program on paper that loops through a dictionary. Example:
- Code:

```
  # Show tasks
  tasks = {"morning": "study", "afternoon": "exercise"}
  for task, time in tasks.items():
      print(f"{time}: do {task}")
```

- Result: Shows:

```
  morning: do study
  afternoon: do exercise
```

Write your own program with a dictionary of at least two key-value pairs. Loop through it using `.items()`. Check: Does the loop show all pairs? Is the syntax correct? Fix any errors.

Dictionaries in a Project: Track Player Stats

Let's combine dictionaries with other skills to build a small program that tracks player stats for a game. The user can add stats, view them, or quit. Here's an example:

```
stats = {}
while True:
    print("1. Add stat")
    print("2. View stats")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    if choice == "1":
        key = input("Enter stat name (e.g., score): ")
        value = input("Enter value: ")
        try:
            stats[key] = int(value)  # Try to convert to number
        except ValueError:
            stats[key] = value  # Keep as string if not a number
        print(f"Added {key}: {stats[key]}")
    elif choice == "2":
        if len(stats) == 0:
            print("No stats yet.")
        else:
            print("Player stats:")
            for key, value in stats.items():
                print(f"{key}: {value}")
    elif choice == "3":
        print("Game stats saved. Goodbye!")
        break
    else:
        print("Choose 1, 2, or 3!")
```

This program:
- Uses a dictionary (`stats`) to store key-value pairs.
- Runs a menu loop to add, view, or quit.
- Handles errors with `try`/`except` for number inputs.
- Loops through the dictionary to show stats.

Let’s practice this project.

Exercise: Player Stats Program
Write a Python program on paper for a stats tracker with add and view options. Example:
- Code:

```
# Stats tracker
stats = {}
while True:
    print("1. Add stat  2. View stats  3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "1":
        stat = input("Stat name: ")
        value = input("Value: ")
        stats[stat] = value
        print(f"Saved {stat}")
    elif choice == "2":
        if len(stats) == 0:
            print("No stats.")
        else:
            for key, val in stats.items():
                print(f"{key}: {val}")
    elif choice == "3":
        print("Done!")
        break
    else:
        print("Invalid choice!")
```

- Result: Saves stats (e.g., “score: 10”) and shows them, or quits.

Write your own program with a dictionary, menu, and loop. Check: Does the dictionary store data? Does the loop work? Fix any errors.

Why Dictionaries Matter

Dictionaries let you organize complex data, like stats or settings, in a way that’s easy to access and update. They’re perfect for:
- Tracking player info in games.
- Storing settings or preferences.
- Managing question banks for quizzes.

Combined with loops, conditions, and functions, dictionaries make your programs flexible and powerful. In the next chapter, we’ll wrap up with a final project: a to-do list app, combining everything you’ve learned. For now, keep practicing dictionaries on paper—you’re building real Python programs!

Reflection Prompt
What would you use a dictionary for, like tracking goals or game stats? How would keys and values help?

Paper Coding Exercise
Write a Python program on paper that:
- Creates a dictionary with at least three key-value pairs.
- Uses a loop to show all pairs.
- Includes a menu to add a new pair or quit.
- Handles input errors with `try`/`except`.

Run it in your head with sample inputs and fix any errors.

Chapter 20: Build a To-Do List App

You’ve built a calculator, a quiz game, a number guessing game, and a journal app, mastering Python’s core tools—variables, loops, functions, dictionaries, and more. Now, let’s tie it all together with a final project: a *to-do list app*. This program will let users add tasks, mark them as done, view their list, and save it for later. It’s a practical way to practice everything you’ve learned while creating something you can use in real life. You can write and test this on paper, so no computer is needed—just your brain, a pen, and some paper. Let’s get started!

Planning the To-Do List App

Before writing code, let’s plan what the app will do:
- **Input**: Let users add tasks, mark tasks as done, view the list, or quit.
- **Processing**: Store tasks in a dictionary (with task names as keys and status like “done” or “pending” as values), update statuses, and simulate saving to a file.
- **Output**: Show the task list with statuses and confirm actions (like “Task added!”).
- **Features**: Use a menu for user choices, handle errors, and keep tasks organized.

Here’s the idea in plain English:
- Show a menu: “Add task, mark done, view tasks, or quit.”
- If adding a task, get the task name and store it as “pending.”
- If marking done, update the task’s status to “done.”
- If viewing, show all tasks with numbers and statuses.
- If quitting, stop and “save” the list.
- Use a dictionary to track tasks and statuses.

This uses dictionaries, loops, conditions, input, and error handling—perfect for showcasing your skills.

Writing the To-Do List App

Let’s write a Python program for a to-do list app. We’ll use a dictionary to store tasks and simulate saving them. Here’s a complete program:

```
# To-Do List App
tasks = {}
```

```
while True:
    print("1. Add task")
    print("2. Mark task done")
    print("3. View tasks")
    print("4. Quit")
    choice = input("Choose an option (1-4): ")

    if choice == "1":
        task = input("Enter task name: ")
        if task in tasks:
            print("Task already exists!")
        else:
            tasks[task] = "pending"
            print(f"Added: {task}")
    elif choice == "2":
        task = input("Enter task to mark done: ")
        if task in tasks:
            tasks[task] = "done"
            print(f"{task} marked as done!")
        else:
            print("Task not found!")
    elif choice == "3":
        if len(tasks) == 0:
            print("No tasks yet.")
        else:
            print("Your tasks:")
            for i, (task, status) in enumerate(tasks.items(), 1):
                print(f"{i}. {task} - {status}")
    elif choice == "4":
        print("Tasks saved. Goodbye!")
        break
    else:
        print("Choose 1, 2, 3, or 4!")
```

This program:
- Uses a dictionary (`tasks`) where keys are task names and values are statuses (“pending” or “done”).
- Runs a `while True` loop to show a menu and get choices.
- For choice `1`, adds a task if it doesn’t exist.
- For choice `2`, updates a task’s status to “done” if it exists.
- For choice `3`, shows tasks with numbers using `enumerate`.
- For choice `4`, stops with `break`.
- Checks for errors like duplicate tasks or invalid choices.

Let’s test it in your head:
- Choose `1`, enter “Study Python” → “Added: Study Python”
- Choose `1`, enter “Exercise” → “Added: Exercise”

- Choose `3` → Shows:

```
Your tasks:
1. Study Python - pending
2. Exercise - pending
```

- Choose `2`, enter “Study Python” → “Study Python marked as done!”
- Choose `3` → Shows:

```
Your tasks:
1. Study Python - done
2. Exercise - pending
```

- Choose `4` → “Tasks saved. Goodbye!”

Let’s practice writing part of it.

Exercise: Basic To-Do List Program
Write a Python program on paper for a to-do list with *add* and *view* options. Example:
- Code:

```
# Simple to-do list
tasks = {}
while True:
    print("1. Add task  2. View tasks")
    choice = input("Choose 1 or 2: ")
    if choice == "1":
        task = input("Task name: ")
        tasks[task] = "pending"
        print("Task added!")
    elif choice == "2":
        if len(tasks) == 0:
            print("No tasks.")
        else:
            for i, task in enumerate(tasks, 1):
                print(f"{i}. {task}")
    else:
        print("Choose 1 or 2!")
```

- Result: Adds tasks and shows them numbered, or shows “No tasks.”

Write your own program with add and view options. Use a dictionary and loop. Check: Does the dictionary store tasks? Does the view show all tasks? Fix any errors.

Adding a Function

To make the code cleaner, let’s use a function to handle adding and viewing tasks. Here’s an example:

```python
def manage_task(tasks, choice):
    if choice == "1":
        task = input("Enter task name: ")
        if task in tasks:
            return "Task already exists!"
        tasks[task] = "pending"
        return f"Added: {task}"
    elif choice == "2":
        if len(tasks) == 0:
            return "No tasks yet."
        result = "Your tasks:\n"
        for i, (task, status) in enumerate(tasks.items(), 1):
            result = result + f"{i}. {task} - {status}\n"
        return result
    return "Invalid choice!"

# Main program
tasks = {}
while True:
    print("1. Add task")
    print("2. View tasks")
    print("3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "3":
        print("Tasks saved. Goodbye!")
        break
    print(manage_task(tasks, choice))
```

This program:
- Defines a `manage_task` function to handle adding and viewing.
- Returns messages or the task list for printing.
- Keeps the main loop simple.

Let’s practice with a function.

Exercise: To-Do List with Function
Write a Python program on paper with a function for task actions. Example:
- Code:

```python
  # To-do with function
  def handle_task(tasks, choice):
      if choice == "1":
          task = input("Task: ")
          tasks[task] = "pending"
          return "Task added!"
      elif choice == "2":
          if len(tasks) == 0:
              return "No tasks."
          return "\n".join([f"{i}. {t}" for i, t in enumerate(tasks, 1)])
      return "Invalid!"
  tasks = {}
  while True:
      print("1. Add  2. View  3. Quit")
      choice = input("Choose 1-3: ")
      if choice == "3":
          print("Done!")
          break
      print(handle_task(tasks, choice))
```

- Result: Adds tasks and shows them, or quits.

Write your own program with a function for add and view. Check: Does the function handle both actions? Is the loop correct? Fix any errors.

Adding Simulated File Handling and Error Handling

To make the app more robust, let's simulate saving tasks to a file (using the dictionary as the "file") and handle errors. Here's an example:

```
tasks = {}
while True:
    print("1. Add task")
    print("2. Mark task done")
    print("3. View tasks")
    print("4. Quit")
    choice = input("Choose an option (1-4): ")

    try:
        if choice == "1":
            task = input("Enter task name: ")
            if task.strip() == "":
                print("Task cannot be empty!")
            elif task in tasks:
                print("Task already exists!")
            else:
                tasks[task] = "pending"
                print(f"Added: {task} (saved to file)")
        elif choice == "2":
            task = input("Enter task to mark done: ")
            if task in tasks:
                tasks[task] = "done"
                print(f"{task} marked done (saved to file)")
            else:
                print("Task not found!")
        elif choice == "3":
            if len(tasks) == 0:
                print("No tasks yet.")
            else:
                print("Your tasks:")
                for i, (task, status) in enumerate(tasks.items(), 1):
                    print(f"{i}. {task} - {status}")
        elif choice == "4":
            print("Tasks saved to file. Goodbye!")
            break
        else:
            print("Choose 1, 2, 3, or 4!")
    except Exception:
        print("An error occurred. Try again!")
```

This program:
- Simulates saving by updating the `tasks` dictionary.
- Checks for empty inputs with `task.strip() == ""`.
- Uses `try`/`except` to catch unexpected errors.
- Shows tasks with statuses and numbers.

Let’s practice this.

Exercise: Robust To-Do List
Write a Python program on paper with error handling and simulated file saving. Example:
- Code:

```
# Robust to-do list
tasks = {}
while True:
    print("1. Add  2. View  3. Quit")
    choice = input("Choose 1-3: ")
    try:
        if choice == "1":
            task = input("Task: ")
            if task.strip() == "":
                print("Enter a task!")
            else:
                tasks[task] = "pending"
                print("Task saved!")
        elif choice == "2":
            if len(tasks) == 0:
                print("No tasks.")
            else:
                for i, t in enumerate(tasks, 1):
                    print(f"{i}. {t}")
        elif choice == "3":
            print("Saved. Done!")
            break
        else:
            print("Choose 1-3!")
    except:
        print("Error, try again!")
```

- Result: Adds tasks, shows them, handles empty inputs, and quits.

Write your own program with add, view, and quit options, plus error handling. Check: Does it catch empty inputs? Does the dictionary work? Fix any errors.

Why This Project Matters

The to-do list app combines everything you've learned into a practical tool you could use daily. It shows you how to:
- Organize data with dictionaries.
- Build an interactive menu with loops and conditions.
- Handle errors for a smooth user experience.
- Simulate file handling for data persistence.

This project is a stepping stone to real-world apps like task managers or planners. Congratulations—you've built skills to create meaningful Python programs!

Reflection Prompt
How could a to-do list app help you in daily life? What extra features, like due dates or categories, would you add?

Paper Coding Exercise
Write a Python to-do list program on paper that:
- Uses a dictionary to store tasks and statuses.

- Has a menu with add, mark done, view, and quit options.
- Uses a function for at least one action.
- Includes error handling for empty inputs or invalid choices.

Run it in your head with sample tasks and fix any errors.

Chapter 21: What's Next? Your Coding Journey

Congratulations! You’ve built a calculator, a quiz game, a number guessing game, a journal app, and a to-do list app, mastering Python’s core concepts—variables, loops, functions, dictionaries, file handling, and error debugging. You’ve written programs on paper, proving you can think like a coder without a computer. Now, it’s time to look ahead: what can you do with these skills, and how can you keep growing? In this final chapter, we’ll explore ways to apply your Python knowledge, plan your next steps, and stay motivated. You’ll reflect on your progress and design one last paper-based program to solidify your skills. No computer is needed—just your brain, a pen, and some paper. Let’s dive in!

Reflecting on Your Progress

You’ve come a long way since Chapter 4, where you learned what a program is. You’ve gone from writing paper instructions to crafting Python programs with:
- **Variables and Data**: Storing numbers, strings, lists, and dictionaries.
- **Logic and Flow**: Using conditions (`if`/`elif`/`else`) and loops (`for`/`while`).
- **Functions**: Organizing code into reusable chunks.
- **Interactivity**: Getting user input and showing clear output.
- **Error Handling**: Debugging and catching mistakes like invalid inputs.
- **Projects**: Building real apps like a calculator, quiz, and to-do list.

These skills are the foundation of coding. Whether you want to build apps, analyze data, or automate tasks, you’re ready to take on new challenges.

Let’s reflect with an exercise.

Exercise: Reflect on Your Skills
On paper, write answers to these questions:
1. What’s one Python concept (e.g., loops, dictionaries) you feel confident about? Why?
2. What’s one concept that was tricky, and how did you overcome it?
3. Name a program you built (like the quiz or to-do list) that you’re proud of. What did it teach you?

Check: Are your answers specific? For example, instead of “loops,” say “using a `for` loop to show a list.” This helps you see your growth.

Applying Your Skills

Your Python skills can be used in many ways, even without a computer right now. Here are some ideas:
- **Personal Projects**: Plan programs like a budget tracker, a fitness log, or a story generator.
- **Problem-Solving**: Use coding logic to organize tasks, like scheduling or tracking goals.
- **Learning More**: Practice advanced Python concepts (like classes or web scraping) on paper to prepare for computer access.
- **Helping Others**: Share your knowledge by explaining coding to a friend or writing a guide.

When you get computer access, you can:
- Run your paper programs using tools like Python’s IDLE, VS Code, or online platforms like Replit.
- Explore libraries like `turtle` for graphics or `pandas` for data analysis.
- Join coding communities on platforms like X to ask questions and share ideas.

Let’s plan a future project.

Exercise: Plan a Future Program
On paper, describe a program you’d like to build. Example:
- Program: A habit tracker.
- Features: Add daily habits, mark them complete, show progress.
- Python Tools: Dictionary for habits, loop for menu, function to check progress.
- Purpose: Track goals like reading or exercising.

Write your own program idea. Include:
- What it does.
- At least two Python features (e.g., lists, functions).
- Why it’s useful to you.
Check: Is your plan clear? Could you start coding it with your skills? Refine it if needed.

Designing a Final Program

To wrap up, let’s design a new program that combines everything you’ve learned. This will be a *goal tracker* that lets users add goals, track progress, and view their status. You’ll write it on paper, using dictionaries, loops, functions, and error handling.

Here’s a sample program:

```
# Goal Tracker App
def update_goal(goals, goal_name, progress):
    try:
        progress = int(progress)
        if goal_name in goals:
            goals[goal_name] = progress
            return f"Updated {goal_name} to {progress}%"
        else:
            goals[goal_name] = progress
            return f"Added {goal_name} with {progress}%"
    except ValueError:
        return "Progress must be a number!"

goals = {}
while True:
    print("1. Add/Update goal")
    print("2. View goals")
    print("3. Quit")
    choice = input("Choose 1-3: ")

    try:
        if choice == "1":
            goal = input("Enter goal name: ")
            if goal.strip() == "":
                print("Goal name cannot be empty!")
            else:
                progress = input("Enter progress (0-100): ")
                print(update_goal(goals, goal, progress))
        elif choice == "2":
            if len(goals) == 0:
                print("No goals yet.")
            else:
                print("Your goals:")
                for i, (goal, prog) in enumerate(goals.items(), 1):
                    print(f"{i}. {goal}: {prog}% complete")
        elif choice == "3":
            print("Goals saved. Goodbye!")
            break
        else:
            print("Choose 1, 2, or 3!")
    except Exception:
        print("An error occurred. Try again!")
```

This program:
- Uses a dictionary (`goals`) to store goal names and progress percentages.
- Has a function (`update_goal`) to add or update goals.
- Runs a menu loop with options to add/update, view, or quit.
- Handles errors like empty names or non-numeric progress.
- Shows goals with numbers and percentages.

Let’s practice designing your own.

Exercise: Final Program Design
Write a Python program on paper for a goal tracker (or a similar app, like a book log or workout tracker). Example structure:
- Code:

```
# Simple goal tracker
def add_goal(goals, name):
    goals[name] = 0
    return f"Added {name}"
goals = {}
while True:
    print("1. Add goal  2. View goals  3. Quit")
    choice = input("Choose 1-3: ")
    if choice == "1":
        name = input("Goal name: ")
        if name.strip() == "":
            print("Enter a name!")
        else:
            print(add_goal(goals, name))
    elif choice == "2":
        if len(goals) == 0:
            print("No goals.")
        else:
            for i, g in enumerate(goals, 1):
                print(f"{i}. {g}")
    elif choice == "3":
        print("Done!")
        break
    else:
        print("Choose 1-3!")
```

- Result: Adds goals and shows them numbered, or quits.

Write your own program with:
- A dictionary to store data.
- A function for at least one action (e.g., adding or updating).
- A menu loop with at least three options (add, view, quit).
- Error handling for invalid inputs.
Run it in your head with sample inputs (e.g., add two goals, view them, quit) and fix any errors.

Staying Motivated

Coding is a journey, and you've already taken big steps. Here are tips to keep going:
- **Practice Regularly**: Write one small program a day on paper, like a counter or a greeting.
- **Think Big, Start Small**: Dream of big projects (like a game), but break them into small pieces (like a score tracker).
- **Learn from Others**: When you get computer access, check X for Python tips or join coding groups.
- **Celebrate Wins**: Every program you write, even on paper, is progress. Be proud!

Let's set a goal.

Exercise: Set a Coding Goal
On paper, write one coding goal for the next month. Example:
- Goal: Write 10 small Python programs on paper, like a budget tracker or a random picker.
- Plan: Spend 10 minutes daily planning and writing one program.
- Why: To build confidence and prepare for running code on a computer.

Write your own goal, including:
- What you'll do (e.g., design programs, learn a new concept).
- How you'll do it (e.g., 15 minutes daily, one chapter of a book).
- Why it matters to you.

Check: Is your goal clear and achievable? Adjust if needed.

Why This Matters

You've learned to think like a coder, plan programs, and write Python code that solves real problems. These skills open doors to:
- Creating tools for personal use, like trackers or planners.
- Exploring careers in tech, like programming or data analysis.
- Joining a community of coders to share ideas and grow.

This book may end here, but your coding journey is just beginning. Keep practicing, stay curious, and know that every line of code you write brings you closer to your goals.

Final Reflection Prompt
What's one thing you've learned from this book that surprised you? How will you use coding in your life, even without a computer right now?

Final Paper Coding Exercise
Write a Python program on paper for an app of your choice (e.g., goal tracker, book log, or mini-game). Include:
- A dictionary or list to store data.
- A function for at least one action.
- A menu loop with at least three options.
- Error handling for invalid inputs.
- Clear output with f-strings.

Run it in your head with sample inputs and fix any errors. Example: A book log that adds books, marks them read, and shows the list. Save this program—it's your first step to future projects!

Thank you for completing Code Without Bars: Learning Python from the Ground Up! Keep coding, keep learning, and keep dreaming big!

Have you ever wanted to create something that's *yours*? With "Code Without Bars", you'll learn Python, a powerful and beginner-friendly programming language, from the ground up. No fancy degree or computer required—just your curiosity and a willingness to try. From writing your first program to building tools like calculators, games, and journals, this book guides you step-by-step with clear explanations and hands-on projects. Written by Encrypter15, this is your key to unlocking creativity, problem-solving, and a future where you're in control.

Contact the author at Encrypter15@gmail.com for feedback or inquiries.