IN THIS CHAPTER

- Understanding how programming works
- Perusing a few programming languages
- Taking a peek under the coding hood
- Figuring out why you'd want to learn to code
- Learning how code is used in the real world

Chapter 1
What Is Coding?

Everybody should learn to program a computer because it teaches you how to think.

—STEVE JOBS

Okay, I'll admit it: I'm not one of those look-before-you-leap types. I'm a dedicated leaper. With almost every new thing I learn, my philosophy is that I learn best when I *do* the thing. I usually just jump in, get my hands dirty, make mistakes, fix them, and before long I develop a feel for this new hobby or skill.

Notice, though, that I said I do that with *almost* every new thing I learn. I don't do it when I'm learning something related to coding. Why not? "Just jumping in" is a lousy strategy for learning to code because the mistakes you make are dumb ones that teach you nothing (as opposed to smart mistakes that you can learn from). When it comes to learning anything related to code, it's always best to start with the basics and work your way slowly and steadily to competency, then proficiency, then mastery.

This chapter is your look-before-you-leap introduction to coding. Here you explore what coding is and how it works. You discover the myriad reasons why it's good to learn to code, and you investigate quite a few real-world uses for coding.

Programming: Making a Computer Do Your Bidding

A computer is a machine that follows instructions. Or, to put a finer point on it, a computer is a machine that does nothing until someone or something tells it what to do. That might sound surprising. After all, computers cost many hundreds, sometimes even thousands, of dollars, and are positively bristling on the inside with electronic gadgetry. Surely something so expensive and so complex must be capable of doing some useful tasks on its own.

Nope.

Sure, when you turn on a new computer for the first time, some hieroglyphics appear on the screen and you eventually end up in Windows or macOS or some other desktop. Doesn't that so-called boot process mean that the computer is doing something on its own?

Again, nope.

When you turn on a computer, it automatically loads a set of instructions that tell the computer what it must do to get the hardware (keyboard, mouse, screen, and so on) up and running and to get the operating system (Windows or macOS or whatever) loaded. That set of instructions is known as the computer's *firmware*, which is a special type of program embedded in the computer hardware. When the firmware has completed its job, it calls the *bootloader*, which loads the operating system.

Okay, that's all fine, but where did the firmware and bootloader come from? I'm glad you asked because at long last I can get to the point of all this: Someone coded them.

Some very smart person versed in the esoterica of computer hardware and system software programmed the firmware, and some other just as smart person coded the bootloader. So, let me repeat myself: Computers can't do anything unless someone or something tells them what to do. And the way you tell a computer what to do is via code.

Coding firmware or a bootloader is hideously complex and requires years of study. Happily, you won't be going anywhere near that level of complexity in this book. Whew! But everything you do learn here will be a variation on the overall theme of this section: telling a computer what you want it to do using code.

Am I talking about making a computer do anything you want? Alas, no, although that would be very useful! When you code, you're given a set of tools for the job; the tools you work with vary depending on the language you're using. As I discuss later in this book, the tools you get with Python (refer to Part 2) are much different than the tools you get with JavaScript (check out Part 3). As an analogy, the types of home projects you'd take on would be very different depending on whether you had a carpenter's toolbox or a sewing kit.


REMEMBER

But no matter how you code — no matter what programming tools you have at your disposal — you're almost always doing one (or sometimes both) of the following:

- **Solving a problem:** One of the most common reasons that a piece of code gets written is because the coder had a pain point or an inefficiency in their life and saw a way to use code to make their life easier or more streamlined.
- **Creating something new:** Another common reason to start coding is when you get a great idea and want more than anything to bring that idea to life.

No matter what you work on in your coding career, you're almost always doing one (or both) of these things — solving problems, creating new stuff, or combining the two to make something that's both new *and* improved.

What Is a Programming Language?

Python and JavaScript are programming languages. Okay, fine, but what does it mean to call something a *programming language?* To understand this term, you need look no further than the language you use to speak and write. At its most fundamental level, human language is composed of two things — words and rules:

- The words are collections of letters that have a common meaning among all the people who speak the same language. For example, the word *book* denotes a type of object; the word *heavy* denotes a quality; and the word *read* denotes an action.
- The rules are the ways in which words can be combined to create coherent and understandable concepts. If you want to be understood by other speakers of the language, you have only a limited number of ways to throw two or more words together. "I read a heavy book" is an instantly comprehensible sentence, but "book a I read heavy" is gibberish.

The key goal of human language is being understood by someone else who is listening to you or reading something you wrote. If you use the proper words to refer

to things and actions and if you combine those words according to the rules, the other person will understand you.

A programming language works in more or less the same way. That is, it, too, has words and rules:

- The words are a set of terms that refer to the specific things that your program works with or the specific ways in which those things can be manipulated. These words are known as *reserved words* or *keywords.*
- The rules are the ways in which the words can be combined to produce the desired effect. In the programming world, these rules are known as the language's *syntax.*

The crucial concept here is that just as the fundamental purpose of human language is to be understood by another person, the fundamental purpose of a programming language is to be understood by whatever machine is processing the language. The key, however, is that being "understood" by the machine really means being able to *control* the machine. That is, your code "sentences" are commands that you want the machine to carry out.

The Role of Programming Languages

Let's say you travel to Igboland in Nigeria and want to ask a local for directions to the nearest bathroom. If that person speaks only Igbo (the native language of Igboland), one solution would be to find someone who speaks both English and Igbo and ask that person to translate your request as well as the response. Problem solved!

The person who can translate your English into Igbo is called an *interpreter,* and that task is essentially how we're able to program a computer. The problem is that a computer understands only its native language, which is called *machine language* and consists of 1s and 0s. (I won't get into this topic here, but if you're curious to know more, check out the sidebar "How computers work: A crash course for would-be coders.") A very simple machine language instruction to a computer might look something like this:

```
10111000 00000001 00000000 00000000 00000000
10111111 00000001 00000000 00000000 00000000
01001000 10111110 00000000 01100000 01100000
00000000 00000000 00000000 00000000 00000000
10111010 00001101 00000000 00000000 00000000
```

```
00001111 00000101 10111000 00111100 00000000
00000000 00000000 00110001 11111111 00001111
00000101
```

Yikes! No sane human wants to deal with something as weird as machine language, so one of the first things that engineers did after computers were invented was come up with two remarkable inventions:

- A way of representing machine-language instructions as human-understandable English words
- A way of converting those English words back into the machine language that the computer understands

The first invention is called a *programming language* and consists of, in part, English (or, sometimes, English-like) words such as `if`, `while`, and `return`. You use these generally comprehensible terms to construct *statements*, which are commands that you want the computer to carry out on your behalf.

For example, the preceding machine language code began life, in part, as the following statement:

```
printf("Hello, World!");
```

This statement, which is written in the C programming language, outputs the text *Hello, World!* C is an example of a *high-level language*, which describes any programming language that abstracts away the mind-numbing complexity of the computer's native machine language.

TECHNICAL STUFF

HOW COMPUTERS WORK: A CRASH COURSE FOR WOULD-BE CODERS

You might have heard someone say, with great authority, that "computers operate by processing 1s and 0s." If, upon hearing that, you were flummoxed, let me tell you that your reaction is utterly normal. It really *is* incomprehensible to us mere mortals that computers, which can do all these incredible things, perform those wonders by slinging around just two values: 1 and 0. What's behind this mystery?

At the lowest level, a computer is basically a collection of billions of unimaginably teensy components called *transistors,* which operate essentially as on/off switches for electrical

(continued)

(continued)

current. When a transistor allows electrical current to pass through, by convention that state is represented by a 1. When a transistor blocks electrical current from passing through, by convention that state is represented by a 0. Each 1 or 0 is called a *binary digit,* or *bit.* One bit offers only two options: 1 or 0. Combining two bits offers four options: 00, 01, 10, or 11. Skipping ahead, I can tell you that combining eight bits offers 256 options, from 00000000 to 11111111 and every combo in between. A string of eight bits is called a *byte* and the 256 possible byte values is enough to code every letter, every number, every punctuation mark, plus a few other standard symbols that make up the American Standard Code for Information Interchange (ASCII) table. The uppercase letter *H,* for example, is 01001000 in binary. So, combine eight transistors, set them so that they form the byte 01001000, and you've got the letter *H* stored on your circuit board (which might be a memory module).

Do you need to memorize the byte values for every letter, number, and symbol to code a computer? No, not even close! In fact, the history of coding can be seen as the moving farther and farther away from how information is physically stored using transistors to being able to make the computer do your bidding using relatively simple English words.

C is a notoriously difficult language to learn, so aside from a brief mention in Chapter 3, I steer clear of it in this book. Instead, you learn two languages that reside at an even higher (read: easier) level than C: Python (covered in Part 2) and JavaScript (tackled in Part 3).

The second of the inventions I mentioned is called either an *interpreter* or a *compiler*, depending on the programming language. (I explain the difference between interpreters and compilers in Chapter 3.) Either way, the purpose of this invention is to take the English-like code of a programming language and convert it to something (such as machine language) that the computer can read and run. All of this happens behind the scenes, so, as a coder, you never have to lay your eyes on a single 1 or 0 (unless it's part of your Python or JavaScript code, of course).

Understanding How Code Is Written and Executed

At this very early stage of your programming career, the process of coding might seem more than a little mysterious, possibly even downright puzzling. After all, from the outside a computer is a mystifying machine, so the idea that you can somehow *control* this inscrutable hunk of electronica might seem the stuff of

fantasy. Or even if you've already convinced yourself that you can make a computer do your bidding, *how* you do that might still have you scratching your head.

Perhaps the secret to being able to code a computer is having the right equipment, something like needing a loom for weaving or a lathe for woodworking.

Nope, you're way off. Maybe the most surprising thing about code is that it's nothing but text. (To keep things simple, for now I'm ignoring non-text files such as images and videos that you might incorporate in, say, a web page.) Ever used Notepad in Windows or TextEdit on a Mac? Those bare-bones text files are essentially what you use to write your code.

REMEMBER

To describe the programming process in its most generic terms, I like to use what I call the "three-and-a-half Rs" of coding — write, run, revise, and repeat:

- **Write:** In a text file, you write your code as a series of statements, each of which is essentially an instruction to the interpreter or compiler for whatever programming language you're using.
- **Run:** You invoke the programming language's interpreter or compiler and tell it to process the code in the text file you wrote. The interpreter or compiler then executes the code, and the results appear, which could be the program's output or one or more error messages.
- **Revise:** Based on the results of the run, you edit your code to fix any errors that crop up or to improve your code.
- **Repeat:** You write more code (to, say, add new functionality), run it, revise it as needed, and then repeat the cycle until your program or app or web page or whatever is complete.

That's the bird's-eye view. The next two sections bring things slightly closer to the ground by looking at the coding processes specifically for Python and JavaScript.

How Python code works

I go into a pleasing amount of detail about Python in the chapters that make up Part 2, so here I just provide you with a quick overview of the Python coding process:

1. **Using a text editor or code editor, write your Python language statements in a plain text file.**

   When you save your text file for the first time, be sure to name the file with the `.py` file extension, which identifies it as a Python file.

2. **At the command line, type** python**, a space, and the name of the Python file, and then press Enter or Return.**

   For example, the following runs a file named `hello.py`:

   ```
   python hello.py
   ```

   The `python` part of the command invokes the Python interpreter, which processes the content in the Python file one statement at a time. Note that the Python interpreter is available on your computer only if you have installed Python, as I describe in Chapter 4.

   The interpreter then displays the results of the code, which might be some output you defined or one or more error messages.

   As I discuss in Chapter 4, there are other ways to execute Python code, including an interactive Python shell that enables you to run one Python statement at a time and code editors that enable you to run Python code from the editor's development environment.

3. **Return to your code editor and modify the code as needed based on the results of the most recent run, especially to troubleshoot any errors that cropped up.**

4. **Repeat Steps 1 through 3 as required until your Python program is complete.**

How JavaScript code works

JavaScript is the subject of the chapters in Part 3, so I'll just whet your appetite here with a short-and-sweet review of the JavaScript coding process:

1. **Using a text editor or code editor, write your JavaScript language statements in a plain text file.**

   I assume for simplicity that you want to run your JavaScript statements in a web page (that is, an HTML file, which usually uses the `.html` file extension), so there are two ways to go:

   - *Create an external JavaScript file:* Save your text file using the `.js` file extension, which identifies it as a JavaScript file, and then modify your web page code to tell the browser about the file. For example, the following tags reference a file named `hello.js`:

     ```
     <script src="hello.js"></script>
     ```

   - *Embed the JavaScript code inside the HTML file:* Your JavaScript statements reside within a `<script></script>` block:

```
<script>
    JavaScript statements go here
</script>
```

REMEMBER

If this rapid-fire overview is confusing, don't sweat it for now. I discuss all this in more careful detail in Chapter 10.

2. **Open the HTML file in a web browser or, if the HTML file is already open in the browser, refresh the page.**

   The web browser contains a built-in JavaScript interpreter, so as soon as the browser loads the HTML file, it begins processing the JavaScript code one statement at a time. The browser then displays the results of the code, which might be some output you defined, a web page modification, or one or more error messages.

3. **Switch back to your code editor and edit the code as required given what happened when you opened or refreshed the HTML file. In particular, be sure to tackle any errors that cropped up.**

4. **Repeat Steps 1 through 3 as required until your JavaScript code is running the way you want it.**

Why Learn to Code? Let Me Count the Ways

Since you're reading this book, I think it's safe to assume that you want to learn how to code. If that's true, feel free to skip merrily over the rest of this section. However, if you're still on the fence, trying to decide whether you want to spend the time and effort to learn to code, have I got a section for you!

If you're short on time, my immediate answer to the question, "Should I learn to code?" is a real timesaver: Yes, absolutely! If you still need to be convinced, let the next few sections serve as my long answer.

Coding isn't just for nerds

You might have an image of a stereotypical coder in your head, one that no doubt envisages some not-recently-washed nerd sitting in a dank, dimly lit basement surrounded by empty pizza boxes and crushed energy drink cans. Ah, so you *have* seen photos of my office!

I'm sure many coders fit that stereotype, but most don't and you certainly don't have to stop bathing and ruin your diet to code.

Lots of nerds code, but not all coders self-identify as nerds, so if the geeky reputation of coding is holding you back, forget about it. You can code just as you are.

Coding teaches you how to think

While it's true that the essence of coding is writing instructions for a computer to follow, it's not like writing a list of items for your spouse to pick up when they go to the grocery store. That is, you can't just pick up the coding equivalent of a pen and start writing things down as they pop into your head. Coding requires multiple kinds of thought, so in a sense coding teaches you how to think.

For example, when you're considering how to tackle a coding project, it always helps if you can break down the project into smaller, more manageable chunks. Similarly, computers are relentlessly logical beasts, so successful coding requires that you use your logical reasoning skills to "think" like the computer. Every program ever written contains errors, so a big part of coding is troubleshooting problems, which requires understanding how your code works. Converting an idea in your head into code that brings the idea to life is a task that requires large doses of imagination.

REMEMBER

All these skills — breaking down problems, logical reasoning, troubleshooting problems, critical thinking, and imagination — not only make you a good coder but are also tremendously useful outside programming. Whether it's business, finance, science, or trying to assemble some piece of IKEA furniture with its inscrutable instructions, the thinking skills you hone via coding will help with whatever you're doing.

Coding is fun and creative

In the preceding section, I mentioned that getting code to run requires a logical, channel-the-computer approach. I stand by that, but I'll also admit that "think like the computer" isn't a clarion call for fun. That's okay, though, because thinking logically is only part of what it takes to get a program running. Most of us associate creativity with artistic endeavors, but I'm here to tell you that coding is one of the most creative skills you can learn.

For starters, every non-trivial coding project you work on will present you with hurdles that at first seem insurmountable but will soon yield to some creative problem-solving.

But coding creativity really begins with designing and implementing whatever idea has fired your imagination. Want to make a game? Design a website? Create an app? Craft some digital artwork? Whatever it is, coding enables you to take almost any idea, no matter how pie-in-the-sky it might seem to you now, and turn it into an actual, working project that you and others can play, run, or view. Believe me when I tell you that building things from scratch and watching them come to life not only gives your creativity circuits a workout but is also the very definition of fun.

You can build (almost) anything you can imagine

Being able to code is like having a superpower: If you can imagine something, you can build it. Want to create a website for your side hustle? Code it. Have an idea for an awesome game? Make it. Need an app to remind you to drink water? Build it. (Then send it to me. I could really use that app!)

When you learn to code, you give yourself the near-magical ability to create something out of nothing. This ability is incredibly rewarding because now you're not just *using* apps — you're *making* them.

REMEMBER

Scoot over to the "Real-World Uses of Coding" section for a few practical and useful project ideas.

Coding is a universal language

When people with different native languages want to communicate, they can sometimes use another language that they have in common. That common language is called a *lingua franca.*

In today's technical world, code often acts as a kind of lingua franca because programming is one of the few skills that works the same across every industry and country. A JavaScript developer in Japan writes the same kind of code as one in Canada. A Python script written in the U.S. can be used by someone in Germany.

This means coding opens up opportunities around the world. If you ever dream of working internationally (or remotely for a global company), coding can help you get there.

Coding opens the door to high-paying jobs

Speaking of working, coding is one of the best ways to land yourself a great job. Let's start with a big one: money. Tech jobs consistently rank among the highest paying careers. What about opportunities? Even in the age of AI, the demand for programmers continues to grow. Companies in nearly every industry — finance, healthcare, entertainment, even agriculture — need developers. And many programming jobs offer remote work opportunities, flexible hours, and great benefits.

Here are just a few example careers:

- Software developer
- Web developer
- Data scientist
- Cybersecurity analyst
- AI/machine-learning engineer

But you need a computer science degree, right? Not necessarily. Plenty of people who become professional programmers are self-taught using books just like this one!

You don't have to be a pro to benefit from learning to code

Yep, I get it: Being a professional programmer isn't for everyone. It might be the hours, the constant sitting, all that screentime, or whatever. Turning pro is one coding path, but it's not the only one. Whatever field you work in (or want to work in), having even basic coding know-how can give you an edge over your peers.

For example, if you work in marketing, knowing how to code can enable you to automate reports and analyze customer data. If you work in finance, coding can help you write scripts to track stock prices and investments. If healthcare is your field, knowing how to code can help you manage patient data efficiently. If you're an educator, you can use code to create fun and interactive learning tools for students.

Whatever your career, knowing how to code is a bonus skill that makes you more valuable, more productive, and more creative.

Coding is easier to learn than ever before

Back in the dim mists of time otherwise known as the twentieth century, learning to code was hard. A few dedicated hobbyists taught themselves to program, mostly using a relatively simple language called BASIC, but the vast majority of programmers learned to code by obtaining expensive college degrees that required reading enormous textbooks filled with abstruse technical jargon and recondite computer science theory.

Today? Ah, today we have beginner-friendly languages like Python and JavaScript that enable *anyone* to learn to code, no fancy-schmancy college degree required. Forget the jargon and the theory. If you can think reasonably logically and you can break down a problem in smaller challenges, you can learn to code.

Coding is the future

Back in 2011, the venture capitalist Marc Andreesen wrote an op-ed piece titled "Why software is eating the world." He meant that software was and is transforming entire industries and disrupting traditional ways of doing business. He predicted that software would become a crucial and more deeply baked-in component of company operations, products, and services.

However, before software can eat anything, it has to be coded. Before software can be embedded into every facet of business, someone has to program it. Software is all around us now and will soon be ubiquitous. Learning to code now future-proofs your skills, ensuring that you stay relevant in this rapidly evolving, being-eaten-by-software world.

Real-World Uses of Coding

The overall theme of this chapter has been that coding, at its most basic, is just cajoling a computer into performing some task. As the early chapters in Part 2 (Python) and Part 3 (JavaScript) show, it's not that hard to write code that makes a computer do something trivial, such as display text on the screen. Simple and straightforward examples are a great learning tool and an easy way to build your coding confidence, but they lack, well, substance.

After writing and running a few such examples, you might start to wonder whether that's all there is to coding. Is programming all bun and no hamburger? All sizzle and no steak? Are any vegetarians still reading this?

Fortunately, even beginner-welcoming languages such as Python and JavaScript can be used to build useful and fun projects. You build some in this book a bit later, but for now I want to give you a taste of what's possible. Here, broken down into five development categories, are a few things that folks in the real world are building using Python:

- **Automation:** Python can automate boring, repetitive tasks like renaming files, sending emails, and scraping data from websites. For example, a marketer can write a Python script to automatically send out weekly email reports instead of doing it manually.
- **Data science:** Big-time firms such as Google, Netflix, and Facebook use Python to analyze the massive amounts of data they generate. Many business users take advantage of Python libraries such as Pandas and NumPy to help them make sense of customer behavior, market trends, and sales predictions.
- **Machine learning:** This branch of AI enables computers to learn from data and make decisions or predictions without being programmed. Python-based machine-learning tools such as TensorFlow and scikit-learn enable companies to develop AI-powered systems, such as recommendation engines (I'm looking at *you*, Netflix suggestions).
- **Scientific computing and engineering:** NASA and other scientific institutions take advantage of Python to create complex simulations and calculations. Python also helps engineers analyze large datasets in fields such as genetics, physics, and climate modeling.
- **Web development:** Python is used to build web applications using tools such as Django and Flask, and companies such as Instagram and Spotify rely on Python-based web services. However, in web development, Python is most often called upon for building server systems for handling chores such as data storage, user authentication, and security.

REMEMBER

Once you're comfortable with Python, be sure to read Chapter 9, where I take you step-by-step through a few useful Python projects.

Here are some projects that coders in the real world are building using JavaScript, arranged into five development categories:

- **Web page development:** A sprinkling of JavaScript turns a boring web page into something interactive and dynamic. Browser-based JavaScript can request data from a server, display that data on the page, and handle user input. For example, when you type some text in a web page search box and a list of matching items appears lickety-split, that tells you that JavaScript is working feverishly in the background to fetch and display those search results.

- **Web server development:** A JavaScript tool called Node.js runs on many web servers and is used for *back-end* tasks, such as dealing with data, authenticating users, and providing cloud services. Behemoth companies such as LinkedIn and PayPal use Node.js to power their web apps.
- **Mobile apps:** JavaScript tools are available that enable developers to build mobile apps. Using the framework React Native, Facebook and Instagram (and many others) use JavaScript to offer apps that work on both iOS and Android devices.
- **Games:** JavaScript tools such as Phaser.js enable developers to use JavaScript to build games that run either in the browser or on mobile devices.
- **Smart devices:** JavaScript is used to program smart home devices, such as lights, security cameras, and thermostats. And with Node.js, developers can connect JavaScript applications to Internet of Things (IoT) systems.

REMEMBER

After you have the JavaScript basics down, head over to Chapter 15 to learn how to code several practical JavaScript projects.