1 Hello, Reader

I love technology. I've loved it since I was a little girl and my parents bought me an Erector Set that I used to build a giant (to me) robot out of small pierced pieces of metal. The robot was supposed to be powered by a miniature, battery-driven motor. I was an imaginative kid; I convinced myself that once this robot was built, it would move around the house as easily as I did, and I would have a new robot best friend. I would teach the robot to dance. It would follow me around the house, and (unlike my dog) it would play fetch.

I spent hours sitting on the red wool rug in the upstairs hallway of my parents' house, daydreaming and assembling the robot. I tightened dozens of nuts and bolts using the set's tiny, child-sized wrenches. The most exciting moment came when I was ready to plug in the motor. My mom and I made a special trip to the store to get the right batteries for the motor. We got home, and I raced upstairs to connect the bare wires to the gears and turn on my robot. I felt like Orville and Wilbur Wright at Kitty Hawk, launching a new machine and hoping it would change the world.

Nothing happened.

I checked the diagrams. I flicked the on/off switch a few times. I flipped the batteries. Still nothing. My robot didn't work. I went to get my mom.

"You need to come upstairs. My robot isn't working," I said sadly.

"Did you try turning it off and turning it on again?" my mom asked.

"I did that," I said.

"Did you try flipping the batteries?" she asked.

"Yes," I said. I was getting frustrated.

"I'll come look at it," she said. I grabbed her hand and pulled her upstairs. She tinkered with the robot for a little bit, looking at the directions and fiddling with the wiring and turning the switch on and off a few times. "It's not working," she said finally.

"Why not?" I asked. She could have just told me that the motor was broken, but my mother believed in complete explanations. She told me that the motor was broken, and then she also explained global supply chains and assembly lines and reminded me that I knew how factories worked because I liked to watch the videos on *Sesame Street* featuring huge industrial machines making packages of crayons.

"Things can go wrong when you make things," she explained. "Something went wrong when they made this motor, and it ended up in your kit anyway, and now we're going to get one that works." We called the Erector hotline number printed on the instructions, and the nice people at the toy company sent us a new motor in the mail. It arrived in a week or so, and I plugged it in, and my robot worked. By that point, it was anticlimactic. The robot worked, but not well. It could move slowly across the hardwood floor. It got stuck on the rug. It wasn't going to be my new best friend. After a few days, I took the robot apart to make the next project in the kit, a Ferris wheel.

I learned a few things from making this robot. I learned how to use tools to build technology, and that building things could be fun. I discovered that my imagination was powerful, but that the reality of technology couldn't measure up to what I imagined. I also learned that parts break.

A few years later, when I began writing computer programs, I discovered that these lessons from robot building translated well to the world of computer code. I could imagine vastly complex computer programs, but what the computer could *actually* do was often a letdown. I ran into many situations where programs didn't work because a part failed somewhere in the computer's innards. Nevertheless, I persisted, and I still love building and using technology. I have a vast number of social media accounts. I once hacked a crockpot to build a device for tempering twenty-five pounds of chocolate as part of a cooking project. I even built a computerized system to automatically water my garden.

Recently, however, I've become skeptical of claims that technology will save the world. For my entire adult life, I've been hearing promises about what technology can do to change the world for the better. I began studying computer science at Harvard in September 1991, months after Tim Berners-Lee launched the world's first website at CERN, the particle physics lab run by the European

Organization for Nuclear Research. In my sophomore year, my roommate bought a NeXT cube, the same square black computer that Berners-Lee used as a web server at CERN. It was fun. My roommate got a high-speed connection in our dormitory suite, and we used his $5,000 computer to check our email. Another roommate, who had recently come out and was too young for Boston's gay bar scene, used the computer to hang out on online bulletin boards and meet boys. It was easy to believe that in the future, we would do everything online.

For youthful idealists of my generation, it was also easy to believe that the world we were creating online would be better and more just than the world we already had. In the 1960s, our parents thought they could make a better world by dropping out or living in communes. We saw that our parents had gone straight, and communes clearly weren't the answer—but there was this entire new, uncharted world of "cyberspace" that was ours for the making. The connection wasn't just metaphorical. The emerging Internet culture of the time was heavily influenced by the New Communalism movement of the 1960s, as Fred Turner writes in *From Counterculture to Cyberculture*, a history of digital utopianism.[1] Stewart Brand, the founder of the *Whole Earth Catalog*, laid out the connections between the counterculture and the personal computer revolution in an essay called "We Owe It All to the Hippies," in a 1995 special issue of *Time* magazine called "Welcome to Cyberspace."[2] The early Internet was deeply groovy.

By my junior year, I could make a web page or spin up a web server or write code in six different programming languages. For an undergraduate majoring in math, computer science, or engineering at the time, this was completely normal. For a woman, it wasn't. I was one of six undergraduate women majoring in computer science at a university of twenty thousand graduate and undergraduate students. I only knew two of the other women in computer science. The other three felt like a rumor. I felt isolated in all of the textbook ways that cause women to drop out of science, technology, engineering, and mathematics (STEM) careers. I could see what was broken inside the system, for me and for other women, but I didn't have the power to fix it. I switched my major.

I took a job as a computer scientist after college. My job was to make a simulator that was like a million bees with machine guns

attacking all at once so that we could deploy the bees against a piece of software, to test that the software wouldn't go down when it was deployed. It was a good job, but I wasn't happy. Again, it felt like there was nobody around who looked like me or talked like me or was interested in the things that interested me. I quit to become a journalist.

Fast-forward a few years: I went back to computer science as a data journalist. *Data journalism* is the practice of finding stories in numbers and using numbers to tell stories. As a data journalist, I write code in order to commit acts of investigative journalism. I'm also a professor. It suits me. The gender balance is better, too.

Journalists are taught to be skeptical. We tell each other, "If your mother says she loves you, check it out." Over the years, I heard people repeat the same promises about the bright technological future, but I saw the digital world replicate the inequalities of the "real" world. For example, the percentage of women and minorities in the tech workforce never increased significantly. The Internet became the new public sphere, but friends and colleagues reported being harassed online more than they ever were before. My women friends who used online dating sites and apps received rape threats and obscene photos. Trolls and bots made Twitter a cacophony.

I started to question the promises of tech culture. I started to notice that the way people talk about technology is out of sync with what digital technology actually can do. Ultimately, everything we do with computers comes down to math, and there are fundamental limits to what we can (and should) do with it. I think that we have reached that limit. Americans have hit a point at which we are so enthusiastic about using technology for everything—hiring, driving, paying bills, choosing dates—that we have stopped demanding that our new technology is *good*.

Our collective enthusiasm for applying computer technology to every aspect of life has resulted in a tremendous amount of poorly designed technology. That badly designed technology is getting in the way of everyday life rather than making life easier. Simple things like finding a new friend's phone number or up-to-date email address have become time-consuming. The problem here, as in so many cases, is too much technology and not enough people. We turned over record-keeping to computational systems but fired all the humans who kept the information up-to-date. Now, since nobody

goes through and makes sure all the contact information is accurate in every institutional directory, it is more difficult than ever to get in touch with people. As a journalist, a lot of my job involves reaching out to people I don't know. It's harder than it used to be, and it's more expensive, to contact anyone.

There's a saying: When all you have is a hammer, everything looks like a nail. Computers are our hammers. It's time to stop rushing blindly into the digital future and start making better, more thoughtful decisions about when and why to use technology.

Hence, this book.

This book is a guide for understanding the outer limits of what technology can do. It's about understanding the bleeding edge, where human achievement intersects with human nature. That edge is more like a cliff; beyond it lies danger.

The world is full of marvelous technology: Internet search, devices that recognize spoken commands, computers that can compete against human experts in games like Jeopardy! or Go. In celebrating these achievements, it's important not to get too carried away and assume that because we have cool technology, we can use it to solve every problem. In my university classes, one of the fundamental things I teach is that there are limits. Just as there are fundamental limits to what we know in mathematics and in science, so are there fundamental limits to what we can do with technology. There are also limits to what we *should* do with technology. When we look at the world only through the lens of computation, or we try to solve big social problems using technology alone, we tend to make a set of the same predictable mistakes that impede progress and reinforce inequality. This book is about how to understand the outer limits of what technology can do. Understanding these limits will help us make better choices and have collective conversations as a society about what we can do with tech and what we ought to do to make the world truly better for everyone.

I come to this conversation about social justice as a journalist. I specialize in a particular kind of data journalism, called *computational journalism* or *algorithmic accountability reporting*. An *algorithm* is a computational procedure for deriving a result, much like a recipe is a procedure for making a particular dish. Sometimes, algorithmic accountability reporting means writing code to investigate the algorithms that are being used increasingly to

make decisions on our behalf. Other times, it means looking at badly designed technology or falsely interpreted data and raising a red flag.

One of the red flags I want to raise in this book is a flawed assumption that I call *technochauvinism*. Technochauvinism is the belief that tech is always the solution. Although digital technology has been an ordinary part of scientific and bureaucratic life since the 1950s, and everyday life since the 1980s, sophisticated marketing campaigns still have most people convinced that tech is something new and potentially revolutionary. (The tech revolution has already happened; tech is now mundane.)

Technochauvinism is often accompanied by fellow-traveler beliefs such as Ayn Randian meritocracy; technolibertarian political values; celebrating free speech to the extent of denying that online harassment is a problem; the notion that computers are more "objective" or "unbiased" because they distill questions and answers down to mathematical evaluation; and an unwavering faith that if the world just used more computers, and used them properly, social problems would disappear and we'd create a digitally enabled utopia. It's not true. There has never been, nor will there ever be, a technological innovation that moves us away from the essential problems of human nature. Why, then, do people persist in thinking there's a sunny technological future just around the corner?

I started thinking about technochauvinism one day when I was talking with a twenty-something friend who works as a data scientist. I mentioned something about Philadelphia schools that didn't have enough books.

"Why not just use laptops or iPads and get electronic textbooks?" asked my friend. "Doesn't technology make everything faster, cheaper, and better?"

He got an earful. (You'll get one too in a later chapter.) However, his assumption stuck with me. My friend thought that technology was always the answer. I thought technology was only appropriate if it was the right tool for the task.

Somehow, in the past two decades, many of us began to assume that computers get it right and humans get it wrong. We started saying things like "Computers are better because they are more objective than people." Computers have become so pervasive in every aspect of our lives that when something goes awry in the machine, we assume that it's our fault, rather than assume something went

wrong within the thousands of lines of code that make up the average computer program. In reality, as any software developer can tell you, the problem is usually in the machine somewhere. It's probably in poorly designed or tested code, cheap hardware, or a profound misunderstanding of how the actual users would use the system.

If you're anything like my data scientist friend, you're probably skeptical. Maybe you're a person who loves your cell phone, or maybe you've been told your whole life that computers are the wave of the future. I hear you. I was told that too. What I ask is that you stick with me as I tell some stories about people who built technology, then use these stories to think critically about the technology we have and the people who made it. This isn't a technical manual or a textbook; it's a collection of stories with a purpose. I chose a handful of adventures in computer programming, each of which I undertook in order to understand something fundamental about technology and contemporary tech culture. All of those projects link together in a sort of chain, building an argument against technochauvinism. Along the way, I'll explain how some computer technology works and unpack the human systems that technology serves.

The first four chapters of the book cover a few basics about how computers work and how computer programs are constructed. If you already are crystal-clear on how hardware and software work together, or you already know how to write code, you'll probably breeze through chapters 1–3 on computation and go quickly to chapter 4, which focuses on data. These first chapters are important because all artificial intelligence (AI) is built on the same foundation of code, data, binary, and electrical impulses. Understanding what is real and what is imaginary in AI is crucial. Artificial superintelligences, like on the TV show *Person of Interest* or *Star Trek*, are imaginary. Yes, they're fun to imagine, and it can inspire wonderful creativity to think about the possibilities of robot domination and so on—but they aren't real. This book hews closely to the real mathematical, cognitive, and computational concepts that are in the actual academic discipline of artificial intelligence: knowledge representation and reasoning, logic, machine learning, natural language processing, search, planning, mechanics, and ethics.

In the first computational adventure (chapter 5), I investigate why, after two decades of education reform, schools still can't get students to pass standardized tests. It's not the students' or the teachers' fault. The problem is far bigger: the companies that create the most important state and local exams also publish textbooks that contain many of the answers, but low-income school districts can't afford to buy the books.

I discovered this thorny situation by building artificial intelligence software to enable my reporting. Robot reporters have been in the news in recent years because the Associated Press (AP) is using bots to write routine business and sports stories. My software wasn't inside a robot (it didn't need to be, although I'm not averse to the idea), nor did it write any stories for me (ditto). Instead, it was a brand-new application of old-school artificial intelligence that helped reveal some fascinating insights. One of the most surprising findings of this computational investigation was that, even in our high-tech world, the simplest solution—a book in the hands of a child—was quite effective. It made me wonder why we are spending so much money to put technology into classrooms when we already have a cheap, effective solution that works well.

The next chapter (chapter 6) is a whirlwind tour through the history of machines, specifically focused on Marvin Minsky—commonly known as the father of artificial intelligence—and the enormous role that 1960s counterculture played in developing the beliefs about the Internet that exist in 2017, the time this book was written. My goal here is to show you how the dreams and goals of specific individuals have shaped scientific knowledge, culture, business rhetoric, and even the legal framework of today's technology through deliberate choices. The reason we don't have national territories on the Internet, for example, is that many of the people who made the Internet believed they could make a new world beyond government—much like they tried (and failed) to make new worlds in communes.

In thinking about tech, it's important to keep another cultural touchstone in mind: Hollywood. A great deal of what people dream about making in tech is shaped by the images they see in movies, TV programs, and books. (Remember my childhood robot?) When computer scientists refer to *artificial intelligence*, we make a distinction between general AI and narrow AI. *General AI* is the

Hollywood version. This is the kind of AI that would power the robot butler, might theoretically become sentient and take over the government, could result in a real-life Arnold Schwarzenegger as the Terminator, and all of the other dread possibilities. Most computer scientists have a thorough grounding in science fiction literature and movies, and we're almost always happy to talk through the hypothetical possibilities of general AI.

Inside the computer science community, people gave up on general AI in the 1990s.[3] General AI is now called *Good Old-Fashioned Artificial Intelligence* (GOFAI). *Narrow AI* is what we actually have. Narrow AI is purely mathematical. It's less exciting than GOFAI, but it works surprisingly well and we can do a variety of interesting things with it. However, the linguistic confusion is significant. Machine learning, a popular form of AI, is not GOFAI. Machine learning is narrow AI. The name is confusing. Even to me, the phrase *machine learning* still suggests there is a sentient being in the computer.

The important distinction is this: general AI is what we want, what we hope for, and what we imagine (minus the evil robot overlords of golden-age science fiction). Narrow AI is what we have. It's the difference between dreams and reality.

Next, in chapter 7, I define machine learning and demonstrate how to "do" machine learning by predicting which passengers survived the *Titanic* crash. This definition is necessary for understanding the fourth project (chapter 8), in which I ride in a self-driving car and explain why a self-driving school bus is guaranteed to crash. The first time I rode in a self-driving car was in 2007, and the computerized "driver" almost killed me in a Boeing parking lot. The technology has come a long way since then, but it still fundamentally doesn't work as well as a human brain. The cyborg future is not coming anytime soon. I look at our fantasies about technology replacing humans and explore why it's so hard to admit when technology isn't as effective as we want it to be.

Chapter 9 is a springboard for exploring why *popular* is not the same as *good* and how this confusion—which is perpetuated by machine-learning techniques—is potentially dangerous. Chapters 10 and 11 are also programming adventures, in which I start a pizza-calculating company on a cross-country hackathon bus trip (it's popular but not good) and try to repair the US campaign finance

system by building AI software for the 2016 presidential election (it's good but not popular). In both cases, I build software that works—but it doesn't work as expected. Its demise is instructive.

My goal in this book is to empower people around technology. I want people to understand how computers work so that they don't have to be intimidated by software. We've all been in that position at one time or another. We've all felt helpless and frustrated in the face of a simple task that should be easy, but somehow isn't because of the technological interface. Even my students, who grew up being called digital natives, often find the digital world confusing, intimidating, and poorly designed.

When we rely exclusively on computation for answers to complex social issues, we are relying on artificial unintelligence. To be clear: it's the computer that's artificially unintelligent, not the person. The computer doesn't give a flying fig about what it does or what you do. It executes commands to the best of its abilities, then it waits for the next command. It has no sentience, and it has no soul.

People are always intelligent. However, smart and well-intentioned people act like technochauvinists when they are blind to the faults of computational decision making or they are excessively attached to the idea of using computers to the point at which they want to use computers for everything—including things for which the computer is not suited.

I think we can do better. Once we understand how computers work, we can begin to demand better quality in technology. We can demand systems that *truly* make things cheaper, faster, and better instead of putting up with systems that promise improvement but in fact make things unnecessarily complicated. We can learn to make better decisions about the downstream effects of technology so that we don't cause unintentional harm inside complex social systems. And we can feel empowered to say "no" to technology when it's not necessary so that we can live better, more connected lives and enjoy the many ways tech can and does enhance our world.

Notes

1. Turner, *From Counterculture to Cyberculture*.
2. Brand, “We Owe It All to the Hippies.”
3. Dreyfus, *What Computers Still Can’t Do*.

2 Hello, World

To understand what computers *don't* do, we need to start by understanding what computers do well and how they work. To do this, we'll write a simple computer program. Every time a programmer learns a new language, she does the same thing first: she writes a "Hello, world" program. If you study programming at coding boot camp or at Stanford or at community college or online, you'll likely be asked to write one. "Hello, world" is a reference to the first program in the iconic 1978 book *The C Programming Language* by Brian Kernighan and Dennis Ritchie, in which the reader learns how to create a program (using the C programming language) to print "Hello, world." Kernighan and Ritchie worked at Bell Labs, the think tank that is to modern computer science what Hershey is to chocolate. (AT&T Bell Labs was also kind enough to employ me for several years.) A huge number of innovations originated there, including the laser and the microwave and Unix (which Ritchie also helped develop, in addition to the C programming language). C got its name because it is the language that the Bell Labs crew invented after they wrote a language called "B." C++, a still-popular language, and its cousin C# are both descendants of C.

Because I like traditions, we'll start with "Hello, world." Please get a piece of paper and a writing utensil. Write "Hello, world" on the paper.

Congratulations! That was easy.

Behind the scenes, it was more complex. You formed an intention, gathered the necessary tools to carry out your intention, sent a message to your hand to form the letters, and used your other hand or some other parts of your body to steady the page while you wrote so that the physics of the situation worked. You instructed your body to follow a set of steps to achieve a specific goal.

Now, you need to get a computer to do the same thing.

Open your word-processing program—Microsoft Word or Notes or Pages or OpenOffice or whatever—and create a new document. In that document, type “Hello, world.” Print it out if you like.

Congratulations again! You used a different tool to carry out the same task: intention, physics, and so on. You’re on a roll.

The next challenge is to make the computer print “Hello, world” in a slightly different way. We’re going to write a program that prints “Hello, world” to the screen. We’re going to use a programming language called Python that comes installed on all Macs. (If you’re not using a Mac, the process is slightly different; you’ll need to check online for instructions.) On a Mac, open the Applications folder and then open the Utilities folder inside it. Inside Utilities, there’s a program called *Terminal* (see figure 2.1). Open it.

Figure 2.1 Terminal program in the Utilities folder.

Congratulations! You’ve just leveled up your computer skills. You’re now close to the metal.

The metal means the computer hardware, the chips and transistors and wires and so on, that make up the physical substance of a computer. When you open the terminal program, you’re giving yourself a window through the nicely designed graphical user

interface (GUI) so that you can get closer to the metal. We're going to use the terminal to write a program in Python that will print "Hello, world" on the computer screen.

The terminal has a blinking cursor. This marks what's called the *command line*. The computer will interpret, quite literally and without any nuance, everything that you type in at the command line. In general, when you press Return/Enter, the computer will try to execute everything you just typed in. Now, try typing in the following:

```
python
```

You'll see something that looks like this:

```
Python 3.5.0 (default, Sep 22 2015, 12:32:59)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.72)] on
darwin
Type "help," "copyright," "credits" or "license" for more
information.
>>>
```

The triple-carat marks (>>>) tell you that you're in the Python interpreter, not the regular command-line interpreter. The regular command line uses a kind of programming language called a *shell programming language*. The Python interpreter uses the Python programming language instead. Just as there are different dialects in spoken language, so too are there many dialects of programming languages.

Type in the following and press Return/Enter:

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

Congratulations! You just wrote a computer program! How does it feel?

We just did the same thing three different ways. One was probably more pleasant than the others. One was probably faster and easier than the others. The decision about which one was easier and which one felt faster has to do with your individual experience. Here's the radical thing: *one was not better than the other*. Saying that it's better to do things with technology is just like saying it's better to write "Hello, world" in Python versus scrawling it on a piece of paper. There's no innate value to one versus the other; it's about how the

individual experiences it and what the real-world consequences are. With “Hello, world,” the stakes are very low.

Most programs are more complex than “Hello, world,” but if you understand a simple program, you can scale up your understanding to more complex programs. Every program, from the most complex scientific computing to the latest social network, is made by people. All those people started programming by making “Hello, world.” The way they build sophisticated programs is by starting with a simple building block (like “Hello, world”) and incrementally adding to it to make the program more complex. Computer programs are not magical; they are made.

Let’s say that I want to write a program that prints “Hello, world” ten times. I could repeat the same line many times:

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

Ugh. Nope, not going to do that. I’m already bored. Pressing Ctrl+P to paste eight more times would require far too many keystrokes. (To think like a computer programmer, it helps to be lazy.) Many programmers think typing is boring and tedious, so they try to do as little of it as possible. Instead of retyping, or copying and pasting, the line, I’m going to write a loop to instruct the computer to repeat the instruction ten times.

```
x=1
while x<=10:
    print("Hello, world!\n")
    x+=1
```

That’s way more fun! Now the computer will do all the work for me! Wait—what just happened?

I set the value of x to 1 and created a WHILE loop that will run until it reaches the stop condition, `x>10`. On the first time through the loop, `x=1`. The program prints “Hello, world!” followed by a carriage return, or end of line character, which is indicated by `\n` (pronounced backslash-n). A backslash is a special character in Python. The Python interpreter is programmed to “know” that when it reads that special character, it should do something special with the text that happens immediately afterward. In this case, I am telling the computer to print a carriage return. It would be a pain to

start from scratch every time and program each dumb hunk of metal to perform the same underlying functions, like read text and convert it to binary, or to carry out certain tasks according to the conventions of the syntax of our chosen programming language. Nothing would ever get done! Therefore, all computers come with some built-in functions and with the ability to add functions. I use the term *know* because it's convenient, but please remember that the computer doesn't "know" the way that a sentient being "knows." There is no consciousness inside a computer; there's only a collection of functions running silently, simultaneously, and beautifully.

In the next line, `x+=1`, I am incrementing x by one. I think this stylistic convention is particularly elegant. In programming, you used to have to write `x=x+1` every time you wanted to increment a variable to make it through the next round of a loop. The Python stylists thought this was boring, and they wrote a shortcut. Writing `x+=1` is the same as writing `x=x+1`. The shortcut is taken from C, where a variable can be incremented with the notation `x++` or `++x`. There are similar shortcuts in almost every programming language because programmers do a *lot* of incrementing by one.

After one increment, `x=2`, and the computer hits the bottom of the loop. The indentation of the lines under the *while* statement mean that these lines are part of the loop. When it reaches the end of the loop, the computer goes back to the top of the loop—the *while* line—and evaluates the condition again: is `x<=10`? Yes. Therefore, the computer goes through the instructions again and prints "Hello, world!\n" which appears on the screen like this:

```
Hello, world!
```

Then, it increments `x` again. Now, `x=3`. The computer returns to the top of the loop again, and again—until `x=11`. When `x=11`, the stop condition is met, so the loop ends. Here's another way of thinking about it:

```
IF: x<=10
THEN: DO_THE_INSTRUCTIONS_INSIDE_THE_LOOP
ELSE: PROCEED_TO_THE_NEXT_STEP.
```

Each routine (or subroutine) is a small step. If you assemble a lot of small steps together, you can do very big things. Computer

programmers get very good at looking at a task, breaking the task down into small parts, and programming the computer to take care of each of the small parts. Then, you put the parts together and tinker with them a bit to make them work with each other, and soon you have a working computer program. Today's programs are modular, meaning that one programmer can build the first module, another programmer can build the second module, and both modules will be able to work together if they're hooked up the right way.

Now that we've written a program, let's talk about data. Data can be the input or the output of a program. We generate data, meaning information points or units of information, about the world in a variety of ways. The National Weather Service gathers data on the high and low temperatures in thousands of American locales each day. A pedometer can track the number of steps you take in a day, yielding a pattern of steps taken in a day, a week, or a year. A kindergarten teacher I know has his students tally up the number of pockets in his classroom on Mondays. Data can show us the number of people who bought a particular hat; it can show us how many endangered white rhinos are left in the wild; it can show us the rate at which the polar ice caps are melting. Data is fascinating. It gives us insights. It allows us to learn about the world and to grapple with concepts that are beyond our current understanding. (Although if you're old enough to read this book, hopefully you've already come to grips with the idea of other people's pockets.)

Although the data may be generated in different ways, there's one thing all the preceding examples have in common: all of the data is generated by people. This is true of *all* data. Ultimately, data always comes down to people counting things. If we don't think too hard about it, we might imagine that data springs into the world fully formed from the head of Zeus. We assume that because there is data, the data must be true. Note the first principle of this book: *data is socially constructed*. Please let go of any notion that data is made by anything except people.

"What about computer data?" a savvy kindergarten pocket-data collector might ask. That's a very good question. Data generated by computers is ultimately socially constructed because people make computers. Math is a system of symbols entirely created by people. Computers are machines that compute: they perform millions of

mathematical calculations. Computers are not built according to any kind of absolute universal or natural principles; they are machines that result from millions of small, intentional design decisions made by people who work in specific organizational contexts. Our understanding of data, and the computers that generate and process data, must be informed by an understanding of the social and technical context that allows people to make the computers that make the data.

One way to understand what comes out of computers is to understand what goes *into* computers. There are certain physical realities to the computer. Most computers are protected by a hard case, and inside the case is a bunch of circuit boards and stuff. Let me be more specific about this *stuff*. The important parts are the power source, the connection to the screen, the transistors, the built-in memory, and the writeable memory. All these things fall under the category of *hardware*. Hardware is physical; software is anything that runs on top of the hardware.

I first learned about the physical reality of a computer in high school in the 1990s. I was in a special engineering program for kids that was sponsored by Lockheed Martin. There was a Lockheed plant in my small New Jersey town. The building was shaped like a battleship and was surrounded by miles of unused farmland. The rumor back then was that the plant manufactured nuclear weapons, and that under the amber waves of grain were missile silos that would rise and shoot nuclear missiles in the case of attack by the Soviet Union. This was just before the end of the Cold War era, and everyone had seen the terrifying TV movie *The Day After* about the aftermath of a nuclear apocalypse, so we regularly had conversations about where the US missiles were, where the Soviets' missiles would land, and what we would do afterward. A few times a month, I took a school bus to the Lockheed plant to meet up with a handful of other teenagers from local schools and learn about engineering.

People sometimes say that a computer is like a brain. It isn't. If you take a piece out of a brain, the brain will reroute pathways to compensate. Think about the traumatic brain injury suffered by Arizona Congresswoman Gabby Giffords in 2011. Giffords was holding a meeting with constituents in the parking lot of a Safeway grocery store when a lone gunman, Jared Lee Loughner, shot her in the head at point-blank range. Loughner next shot blindly around

the parking lot, killing six people and wounding eighteen. He had been stalking Giffords.

Giffords's intern, Daniel Hernandez Jr., held her upright and applied pressure to the wound while bullets flew through the parking lot. Eventually, bystanders subdued Loughner and police and emergency services arrived. Giffords was in critical condition. Doctors performed emergency brain surgery and then put her into a medically induced coma to allow her brain to heal. Four days after the attack, Giffords opened her eyes. She couldn't speak, she could barely see—but she was alive.

Giffords courageously faced the long road to recovery. With intensive therapy, she relearned how to speak. Like most people who suffer this kind of traumatic brain injury, Giffords's voice was very different than it was before the attack. Her new voice was slower, and her speech sounded labored. Speaking left her tired. Her brain created new pathways that were different than the old, missing pathways. This is one of the amazing things that a brain can do: it can, under very specific conditions and in very specific ways, repair itself.

A computer can't do this. If you take a piece out of a computer, it simply won't work. Everything stored in computer memory has a physical address. The working draft of this book is stored in a particular spot on my computer's hard drive. If that spot was erased, I would lose all these carefully crafted pages. It would be bad; I might have a small breakdown and miss my deadline. However, the ideas would still exist in my brain, so I could recreate the text if necessary. A brain is more flexible and adaptable than a hard drive.

This was one of the many useful things I learned at Lockheed. I also discovered that at tech companies, there are always plenty of slightly outdated spare parts lying around because people upgrade their computers or leave the company. Each teenager in the program was given a case for an Apple II computer, a circuit board, some memory chips, some brightly colored ribbon cables, and miscellaneous other parts scavenged from various offices in the (possibly nuclear) plant. We plugged these components in, and our teacher explained what each part did. The cases were dirty and the keyboards were slightly sticky and all the circuit boards were dusty, but we didn't care. We were building our own computers, and it was fun. After we built our computers, we learned to program them using

a simple programming language called BASIC. At the end of the semester, we got to keep the computers.

I tell this story because it's important to think of a computer as an object that can be and is constructed by human hands. Often, the students who show up in my programming for journalists classes are intimidated by technology. They worry that they are going to break the computer or make some kind of catastrophic misstep. "The only way you can break the computer is with a hammer," I tell them. They rarely believe me at first. By the end of the semester, they are more confident. Even if they break something, they have faith that they can fix it or figure it out. This confidence is key in technological literacy.

You are not in my classroom, so I can't hand you a computer, but I encourage you to take apart an old one. You may have one lying around; otherwise, old computers are often available at thrift stores for not very much money. You might ask around at an office; usually, the system administrator or web person will have some old technology lying around as decoration, or because they haven't gotten around to recycling it yet. A desktop computer is the easiest to use for this activity.

Take the computer apart. You'll probably need a very small screwdriver if you are dismantling a laptop. The interior of the desktop computer probably looks something like the image in figure 2.2.

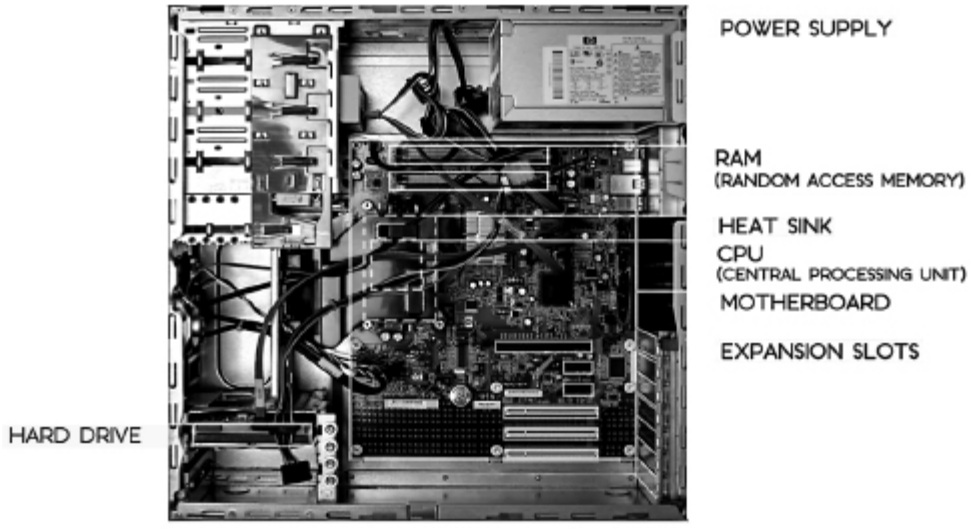

Figure 2.2 The innards of a desktop computer.

Look at the parts, how they are put together. Follow the wires from the inputs (USB port, video port, speaker port, etc.) and see where they connect. Touch the rectangular blobs that seem cemented on the circuit board. Find the microprocessor chips: these are the pieces that probably say "Intel" and are the key to this whole endeavor. They are important. Find the plug that connects the computer hardware to the monitor. It's probably connected to an extremely strong, flexible, plasticky ribbon. This carries information about graphics to the screen, then the screen displays the graphics specified in the code.

When you wrote your Python program, you typed on the keyboard. That information was carried into the computer body from the keyboard, then was interpreted character by character. Then, the computer sent out an instruction from the body to another part of the machine—the monitor—telling it to print the text "Hello, world." This cycle happens over and over again, with simple or complex instructions.

Dismantling a computer is a great activity to do with a kid. I once took apart a laptop with my son when he was in elementary school. I wanted to recycle a couple of laptops, and I was pulling out the hard drives to smash them with a hammer before dropping them at the recycler. (I discovered at some point that smashing a hard drive is easier, and often more satisfying, than erasing it.) I asked my son if he wanted to help me take the hard drive out of the computer. "Are you kidding? I want to take the whole thing apart," he said. So, we spent an enjoyable hour or two disassembling the two laptops on the kitchen counter.

In my university class, we play with hardware and then move on to talking about software—including "Hello, world." Software is everything that runs on top of the hardware. It's what allows you to write an instruction on the keyboard and have the machine act on the instruction. It's what allows the "Hello, world" program to run. Behind the scenes, the text you write is being compiled into instructions that the machine can follow. Hardware is physical; software is everything else. *Computer programming* and *writing software* usually are the same thing.

I'm not going to lie: programming is math. If anyone tries to convince you that it isn't, or that you can really learn programming without doing math, they're probably trying to sell you something.

The good news is, the math that you need for introductory programming is the math you learn around fourth or fifth grade. You'll need to have mastered addition, subtraction, multiplication, division, fractions, percentages, and remainders. You'll need basic geometry like area, perimeter, radius, circumference. You'll need to know basic graphing terms like x, y, and z axes. Finally, you'll need to know the basics of functions—that to turn 2 into 22, we perform a mathematical function on it.

If you have a major math phobia, you probably want to stop reading now. That's OK! There's a lot of rhetoric out there that suggests everyone should learn to code. I don't agree with this. If you really can't do math, coding will probably make you miserable. However, if you're confident that you can calculate the tip at a restaurant, and you can do everyday things like estimate how big a rug to get for your living room, you'll be fine.

To get beyond introductory programming to intermediate programming requires knowing linear algebra, some geometry, and some calculus. However, many people do just fine in their careers with "only" introductory programming skills. Programming can be both an art and a craft. For programming as a craft, you can apprentice and learn and earn a decent living. Programming as an art requires craftsmanship plus training in advanced mathematics. This book assumes you're primarily interested in craft.

There are technical ways to describe how software and hardware work together. For the moment, I'm going to use a metaphor instead. Understanding the layers of a computer is like understanding the layers of a turkey club sandwich (figure 2.3).

Figure 2.3 A turkey club sandwich.

The turkey club is a familiar sight. It has lots of different parts, but they all work well together and result in a delicious sandwich. Just like you build a turkey club in a specific order to achieve a certain effect, a computer runs in a specific order.

Building a turkey club starts with the base layer of bread. That's like the hardware in a computer. The hardware doesn't "know" anything—it just knows how to deal with binary data, 0s and 1s. By *deal with*, I mean *calculate*. Remember that everything a computer can do comes down to math.

On top of the hardware is a layer that allows you to translate words into binary (0s and 1s). Let's call this the *machine-language layer*. It's like the layer of turkey that comes next in the club sandwich. Machine language translates symbols into binary so that the computer can perform calculations. Those symbols are the words and numbers that we humans use to communicate meaning to each other. It's a constructed system. The dialect you use to "speak" machine language is called *assembly language*. It assembles symbols into machine code.

Assembly language is difficult. Here's a sample of an assembly language program to write "Hello, world" ten times, which I copied from a post on a developers' site called Stack Overflow:

```
org
     xor ax, ax
     mov ds, ax
     mov si, msg
boot_loop:lodsb
     or al, al
     jz go_flag
     mov ah, 0x0E
     int 0x10
     jmp boot_loop
go_flag:
     jmp go_flag
msg db 'hello world', 13, 10, 0
     times 510-($-$$) db 0
     db 0x55
     db 0xAA
```

Assembly language is not easy to read or write. Very few people want to spend their days in this language. To make it easier for humans to communicate instructions, we put something on top of the machine-language layer. This is called an *operating system*. On my Mac, the operating system is Linux, which is named after its creator, Linus Torvalds. Linux is based on Unix, the operating system developed by Ritchie of "Hello, world" fame. You probably know operating systems well, even if you don't know what they're called. Part of the personal computer revolution of the 1980s was the triumph of operating systems, which run on top of the machine-language layer and are far easier to interact with if you're a human.

At this point, you have a perfectly serviceable (if plain) computer. You can run all kinds of exciting, interesting programs just using Linux. However, Linux is primarily text-based, and it's not intuitive —so, on the Mac, there's another operating system, OSX, the recognizable Mac interface. It's called a graphical user interface (GUI). The GUI was one of Steve Jobs's great innovations: he realized that using the text-based interface was difficult, so he popularized the practice of putting pictures (icons) on top of the text and using the mouse as a way of navigating among the pictures. Jobs got the idea of the desktop GUI and the mouse from Alan Kay's team

at Xerox PARC, another research lab, which released a computer with a GUI and mouse in 1973. Although we like to credit individuals for technological innovations, rarely is it the case that a lone inventor created any modern computational innovations. When you look closely, there's always a logical predecessor and a team of people who worked on the idea for months or years. Jobs paid for a tour of Xerox PARC, saw the idea of a GUI, and licensed it. The Xerox PARC mouse-and-GUI computer was a derivative of an earlier idea, the oN-Line System (NLS), demonstrated by Doug Engelbart in the "mother of all demos" at the 1968 Association for Computing Machinery conference. We'll look at this intricate history in chapter 6.

The next layer to think about is another software layer: a program that runs on top of an operating system. A web browser (like Safari or Firefox or Chrome or Internet Explorer) is a program that allows you to view web pages. Microsoft Word is a word processing program. Desktop video games like Minecraft are also programs. These programs are all designed to take advantage of certain underlying features of the different operating systems. That's why you can't just run a Windows program on a Mac (unless you use another software program—an emulator—to help you). These programs are designed to seem very easy to use, but underneath they're highly precise.

Let's add some complexity. Imagine that you're a journalist who writes a weekly online column about cats. You use a software program to compose your column. Most journalists compose in a word processing program like Microsoft Word or Google Docs. Either of these programs can run either locally or in the cloud. *Locally* means that the program is running on the hardware on your computer. *In the cloud* means that the program is running on someone else's computer. The cloud is a wonderful metaphor, but practically speaking, *the cloud* just means "a different computer, probably located with thousands of other computers in a large warehouse in the tristate area." The content you create is the truly unique part that comes from your imagination: your elegant, pithy, lovingly crafted story about cats riding Roomba vacuum cleaners or whatever. To the computer, every story is the same, just a collection of 0s and 1s stored on a hard drive somewhere.

After you compose your story, you put it into a content-management system (CMS) so that it can be seen by your editor and

eventually by your audience. A CMS is an essential piece of software for the modern media organization. Media organizations handle hundreds of stories each day, every day. Each story is due at a different time of day; each story is in a different state of editing (or disarray) at any given time; each story has a different headline for print and for online; each story has a different excerpt to be used on each social media platform; each story has images or video or data visualizations or code associated with it; each story is created by a person who needs to be complimented or paid or managed; and all this goes on 24 hours a day, 365 days a year. The scale is vast. I can't stress this enough. It would be foolish to try to manage this type of endeavor without software. The CMS is a tool for managing all the stories and images and so forth that the media organization publishes in print or online.

The CMS also allows the media organization to apply a uniform design template to each story so that the stories all look similar. This is good for branding, but it's also practical. If every single story had to be individually designed for digital presentation, it would take forever to publish anything. Instead, the CMS imposes a standardized design template on top of the raw text that you, the reporter, type into the CMS.

Consider the process of deciding what parts of the design template you will use in your story to decorate it. Will you use pull quotes? Will you include hyperlinks? Will you embed social media posts by people you quote in the story? These are all small design decisions that will affect the reader's experience of your story.

Finally, the story needs to go out into the world. A web server, another piece of software, is used to take the story from the CMS to a person who wants to read the story. The reader accesses the story via a web browser like Chrome or Safari. The web browser is called a *client*. The web server serves the story (which the CMS converts to an HTML page) to the client. The client-server model, the endless sending and receiving of information, is how the web works. The terms *client* and *server* come from restaurants. One way to understand the client-server model is to think about a human server at a restaurant, who distributes food to human clients of the restaurant.

This is the underlying process (more or less) every time you access something on the web. There are many steps and thus many

opportunities for things to go wrong. Really, it's quite impressive that things don't go wrong more often.

Every time you use a computer, you are using this complex set of layers. There is no magic to it, although the results can seem amazing. Understanding the technical realities is important because it allows you to anticipate how, why, and where things will go wrong in a computerized scenario. Even if you feel like the computer is talking to you, or you feel like you are having an interaction with a computer, what you are really doing is having an interaction with a program written by a human being with thoughts, feelings, biases, and background.

This often works out beautifully. It is straight-up fun to interact with Eliza, the 1966 text interaction bot that responds to questions in the manner of a Rogerian psychotherapist. To this day, there are bots on Twitter that respond to users with the patterns pioneered by the Eliza software. A simple Internet search will turn up many examples of Eliza code.[1] Eliza's canned responses are based on the user's input. The replies include the following:

```
Don't you believe that I can _____?
Perhaps you would like to be able to _____.
You want me to be able to _____.
Perhaps you don't want to _____.
Tell me more about such feelings.
What answer would please you the most?
What do you think?
What is it you really want to know?
Why can't you _____?
Don't you know?
```

Try to build an Eliza bot, and the limitations of the form quickly become apparent. Can you build a set of responses that work in any situation? No way. You can think of responses that would suit most situations, but not all. There will always be limitations to what a computer can say in response to a human, because there will always be limits to the imagination of the human computer programmer. Even crowdsourcing will not be adequate, because there will never be enough people to predict every situation that has ever arisen or will ever arise in the future. The world changes; so do conversational styles. Even Rogerian therapy is no longer considered the latest and

greatest interaction style; cognitive behavioral therapy is far more in vogue right now.

Trying to predict every possible response for a bot is doomed in part because we can't get away from unforeseen events. I'm reminded of the time that I found out a friend committed suicide by jumping in front of a New York City subway train. I didn't know this was coming, and I didn't know what to do once I heard. For a while, everything seemed to stop.

Eventually, the shock passed and I began to mourn. But until it happened, I didn't have any way to predict that this particular tragedy would be something that I'd have to assimilate. We're all the same in this regard. Programmers are no better than anyone else at anticipating unexpected, terrible situations. Social groups tend to have a collective blind spot when it comes to imagining the worst. It's a kind of cognitive bias that sociologist Karen A. Cerulo calls "positive asymmetry" in her book *Never Saw It Coming: Cultural Challenges to Envisioning the Worst*. Positive asymmetry is a "tendency to emphasize only the best or most positive cases," she writes. Cultures tend to reward those who focus on the positive and shun or punish those who bring up the downside. The programmer who brings up the potential new audience for a product gets more attention than the programmer who points out that the new product will likely be used for harassment or fraud.[2]

Eliza's responses reflect its designer's basically playful outlook. Looking at Eliza's responses, it's easy to see how voice assistants like Apple's Siri are programmed. The original Eliza had a few dozen responses; Siri includes many, many responses crafted by many, many people. Siri can do a lot: it can send messages, place phone calls, update a calendar with appointments, or set an alarm. It can be fun to stump Siri. Little kids take especial delight in testing the outer limits of what Siri will say. However, Siri and the other voice assistants are limited in their verbal responses by the collective imagination (and positive asymmetry) of their programmers. A team at the Stanford School of Medicine tested the various voice assistants to see whether the assistants recognized a health crisis, responded with respectful language, and referred the person to an appropriate resource. The programs responded "inconsistently and incompletely," the authors wrote in *JAMA Internal Medicine* in 2016. "If conversational agents are to respond fully and effectively to

health concerns, their performance will have to substantially improve."[3]

Technochauvinists like to believe that computers do a better job than people at most tasks. Because the computer operates based on mathematical logic, they think that this logic translates well to the offline world. They are right about one thing: when it comes to calculating, computers do a far better job than people alone. Anyone who has ever graded a student math paper will happily admit that. But there are limits to what a computer can do in certain situations.

Consider the tacocopter, a fanciful idea that had a moment of online popularity. It sounds delightful: a quadcopter drone that delivers a hot, tasty bag full of tacos right to your door! However, when you think about the hardware and software, the flaws in the idea become apparent. A drone is basically a remote-controlled helicopter with a computer and a camera. What happens when it rains? Electrical things don't do well in rain, snow, or fog. My cable television service always malfunctions in a rainstorm, and a wireless drone is far more fragile. Is the tacocopter supposed to come to the window? The front door? How will it push the button in the elevator, or open a stairway door, or push an intercom bell? These are all mundane tasks that are easy for humans, but insanely difficult for computers. How might a tacocopter be co-opted to deliver other, less nutritious and legal substances? What would happen when it inevitably gets shot out of the sky by a freaked-out homeowner with a gun? Only a technochauvinist would imagine that a tacocopter is better than the human-based system that we have now.

If you ask Siri if tacocopters are a good idea, she will look up that phrase for you online. What you'll get are a bunch of news articles about the tacocopter, including one from *Wired* magazine (more on that publication and one of its founders, Stewart Brand, in chapter 6) that debunks the concept more fully than I have done here. The founder admits it's logistically impossible, not least because of FAA regulations on the commercial use of unmanned aerial vehicles. But, she claims, keeping the vision of the idea alive is still important. "Like what cyberpunk did for the internet," she says. "Mull the possibilities, give people things to think about."[4]

What seems to be missing here is a more complete vision of what a world with functioning tacocopters would be like. What would it mean to design buildings and urban environments to enable drones

instead of humans? How would our access to light and air change if windows became docking stations for food-delivery vehicles? What might be the social costs of eradicating even that most mundane and insignificant of interactions—a bag of food being passed from one human hand to another? Do we really want to say "Hello, world" to that reality?

Notes

1. Weizenbaum, "Eliza."
2. Cerulo, *Never Saw It Coming*.
3. Miner et al., "Smartphone-Based Conversational Agents and Responses to Questions about Mental Health, Interpersonal Violence, and Physical Health."
4. Bonnington, "Tacocopter."

3 Hello, AI

We've covered hardware, software, and programming. It's time to move on to a more advanced programming topic: artificial intelligence. To most people, the phrase *artificial intelligence* suggests something cinematic—maybe Commander Data, the lifelike cyborg from *Star Trek: The Next Generation*; perhaps Hal 9000 from *2001: A Space Odyssey*; or Samantha, the AI system from the movie *Her*; or Jarvis, the AI majordomo that helps Iron Man in the Marvel comics and movies. Regardless, here's what's important to remember: *those are imaginary*. It's easy to confuse what we imagine and what is real—especially when we want something very badly. Many people want AI to be real. This usually takes the form of wanting a robot butler to attend to your every need. (I will confess to having had many late-night undergraduate conversations about the practical and ethical considerations of having a robot butler.) A disproportionate number of the people who make tech fall into the camp of desperately wanting Hollywood robots to be real. When Facebook's Mark Zuckerberg built an AI-based home automation system, he named it Jarvis.

One excellent illustration of the confusion between real and imaginary AI happened to me at the NYC Media Lab's annual symposium, a kind of science fair for grownups. I was giving a demo of an AI system I built. I had a table with a monitor and a laptop hooked up to show my demo; three feet away was another table with another demo by an art school undergraduate who had created a data visualization. Things became boring when the crowd died down, so we got to chatting.

"What's your project?" he asked.

"It's an artificial intelligence tool to help journalists quickly and efficiently uncover new story ideas in campaign finance data," I said.

"Wow, AI," he said. "Is it a *real* AI?"

"Of course," I said. I was a little offended. I thought: *Why would I spend my day demonstrating software at this table if I hadn't made something that worked?*

The student came over to my table and started looking closely at the laptop hooked up to the monitor. "How does it work?" he asked. I gave him the three-sentence explanation (you'll read the longer explanation in chapter 11). He looked confused and a little disappointed.

"So, it's not real AI?" he asked.

"Oh, it's real," I said. "And it's spectacular. But you know, don't you, that there's no simulated person inside the machine? Nothing like that exists. It's computationally impossible."

His face fell. "I thought that's what AI meant," he said. "I heard about IBM Watson, and the computer that beat the champion at Go, and self-driving cars. I thought they invented real AI." He looked depressed. I realized he'd been looking at the laptop because he thought there was something in there—a "real" ghost in the machine. I felt terrible for having burst his bubble, so I steered the conversation toward a neutral topic—an upcoming Star Wars movie—to cheer him up.

This interaction stuck with me because it helps me remember the difference between how computer scientists think about AI and how members of the public—including highly informed undergraduates working on tech—think about AI.

General AI is the Hollywood kind of AI. General AI is anything to do with sentient robots (who may or may not want to take over the world), consciousness inside computers, eternal life, or machines that "think" like humans. *Narrow AI* is different: it's a mathematical method for prediction. There's a lot of confusion between the two, even among people who make technological systems. Again, general AI is what some people want, and narrow AI is what we have.

One way to understand narrow AI is this: narrow AI can give you the most likely answer to any question that can be answered with a number. It involves quantitative prediction. Narrow AI is statistics on steroids.

Narrow AI works by analyzing an existing dataset, identifying patterns and probabilities in that dataset, and codifying these patterns and probabilities into a computational construct called a *model*. The model is a kind of black box that we can feed data into

and get an answer out of. We can take the model and run new data through it to get a numerical answer that predicts something: how likely it is that a squiggle on a page is the letter *A*; how likely it is that a given customer will pay back the mortgage money a bank loans to him; which is the best next move to make in a game of tic-tac-toe, checkers, or chess. Machine learning, deep learning, neural networks, and predictive analytics are some of the narrow AI concepts that are currently popular. For every AI system that exists today, there is a logical explanation for how it works. Understanding the computational logic can demystify AI, just like dismantling a computer helps to demystify hardware.

AI is tied up with games—not because there's anything innate about the connection between games and intelligence, but because computer scientists tend to like certain kinds of games and puzzles. Chess, for example, is quite popular in their crowd, as are strategy games like Go and backgammon. A quick look at the Wikipedia pages for prominent venture capitalists and tech titans reveals that most of them were childhood Dungeons & Dragons enthusiasts.

Ever since Alan Turing's 1950 paper that proposed the *Turing test* for machines that think, computer scientists have used chess as a marker for "intelligence" in machines. Half a century has been spent trying to make a machine that could beat a human chess master. Finally, IBM's Deep Blue defeated chess champion Garry Kasparov in 1997. AlphaGo, the AI program that won three of three games against Go world champion Ke Jie in 2017, is often cited as an example of a program that proves general AI is just a few years in the future. Looking closely at the program and its cultural context reveals a different story, however.

AlphaGo is a human-constructed program running on top of hardware, just like the "Hello, world" program you wrote in chapter two. Its developers explain how it works in a 2016 paper published in *Nature*, the international journal of science.[1] The opening lines of the paper read: "All games of perfect information have an optimal value function, $v^*(s)$, which determines the outcome of the game, from every board position or state *s*, under perfect play by all players. These games may be solved by recursively computing the optimal value function in a search tree containing approximately b^d possible sequences of moves, where *b* is the game's breadth (number of legal moves per position) and *d* is its depth (game length)." This is

perfectly clear to someone who has years of high-level mathematical training, but many of us would prefer a plainer-language explanation.

To understand AlphaGo, it helps to start by thinking about tic-tac-toe, a game that most children have mastered. If you go first in tic-tac-toe and choose the space in the middle of the nine-square grid, you can always play to a win or a draw. Going first gives you an advantage: you will have five moves to your opponent's four. Most kids grasp this intuitively and insist on going first when playing with an indulgent older opponent.

It's also relatively easy to write a computer program to play tic-tac-toe against a human opponent. The first one was written in 1952. There's an *algorithm*, a set of rules or steps, that you can deploy so that the computer always plays to a win or a draw. Like "Hello, world," building a tic-tac-toe game is a common exercise in introductory computing classes.

Go is far more sophisticated than tic-tac-toe, but it's also a game played on a grid. Each Go player receives a pile of either black or white stones. Beginners play on a grid made of nine vertical and nine horizontal lines; advanced players use a nineteen-by-nineteen grid. Black goes first and places a black stone at an intersection of two lines. White then places her stone at a different intersection. The players alternate turns, with the goal of "capturing" the opponent's stones by surrounding a stone with the opposite color.

People have been playing Go for three thousand years. Computer scientists and Go aficionados have been studying patterns in the game since at least 1965. The first computerized Go program was written in 1968. There's an entire subfield of computer science research devoted to Go, called (unsurprisingly) *Computer Go*.

For years, Computer Go players and researchers have been amassing records of games. A game record looks like this:

```
(;GM[1]
FF[4]
SZ[19]
PW[Sadavir]
WR[7d]
PB[tzbk]
BR[6d]
DT[2017-05-01]
PC[The KGS Go Server at http://www.gokgs.com/]
```

```
KM[0.50]
RE[B+Resign]
RU[Japanese]
CA[UTF-8]
ST[2]
AP[CGoban:3]
TM[300]
OT[3x30 byo-yomi]
;B[qd];W[dc];B[eq];W[pp];B[de];W[ce];B[dd];W[cd];B[ec];W[cc];B[
df];W[cg];B[kc];W[pg];B[pj];W[oe];B[oc];W[qm];B[of];W[pf];B[pe]
;W[og];B[nf];W[ng];B[nj];W[lg];B[mf];W[lf];B[mg];W[mh];B[me];W[
li];B[kh];W[lh];B[om];W[lk];B[qo];W[po];B[qn];W[pn];B[pm];W[ql]
;B[rq];W[qq];B[rm];W[rl];B[rn];W[rj];B[qr];W[pr];B[rr];W[mn];B[
qi];W[rh];B[no];W[on];B[nn];W[nm];B[nl];W[mm];B[ol];W[mp];B[ml]
;W[ll];B[np];W[nq];B[mo];W[mq];B[lo];W[kn];B[ri];W[si];B[qj];W[
qk];B[kq];W[kp];B[ko];W[jp];B[lp];W[lq];B[jq];W[jo];B[jn];W[in]
;B[lm];W[jm];B[ln];W[hq];B[qh];W[rg];B[nh];W[re];B[rd];W[qe];B[
pd];W[le];B[md])
```

The text may look like gobbledygook to a human, but it's highly structured so that a machine can process it easily. The structure is called *smart games format* (SGF). The text shows who played the game, where, what each of the moves were, and how the game was resolved.

The large text area shows all the moves. Columns in the Go grid are labeled in alphabetical order from left to right, and rows are labeled from top to bottom. In this game, Black (B) went first and placed a stone at the intersection of column q and row d. This is shown as `;B[qd]`. Then, the text `;W[dc]` shows that White (W) placed a stone at the intersection of column d and column c. Each subsequent move is listed in this format. The resolution (RE) of the game is shown in the text `RE[B+Resign]`, which means that Black resigned the game.

The AlphaGo designers amassed a massive dataset of thirty million SGF game files. The dataset wasn't randomly generated; those thirty million games were actual games played by actual people (and some computers). Whenever amateurs or professionals played Go on one of many online sites, that data was saved. It's not hard to create a Go video game; many versions of instructions and free code are posted online. All video games *can* save game data, of course. Some do; some don't. Some save your game data and use it for creating reports for the game company. The people who ran various online Go sites decided to publish their saved game data online in huge batches.

Eventually, these batches were pooled, resulting in the thirty million games collected by the AlphaGo team.

The programmers used the thirty million games to "train" the model that they named *AlphaGo*. What you must remember is that people who play Go professionally spend *ages* playing Computer Go. It's how they train. Therefore, the thirty million games recorded included data from the world's greatest Go players. Millions of hours of human labor went into creating the training data—yet most versions of the AlphaGo story focus on the magic of the algorithms, not the humans who invisibly and over the course of years worked (without compensation) to create the training data.

The developers programmed AlphaGo to use a method called *Monte Carlo search* to pick a set of moves from the thirty million games that would most likely lead to a win. Then, they instructed it to use an algorithm to select the next move from the set. They also instructed it to use a different algorithm that calculated the probability of a win for each possible move in the set. The calculations happened on a scale that the human mind can barely imagine. There are 10^{170} possible board configurations in Go. By layering a variety of computational methods and always choosing the move with the greatest probability of success, the designers created a program that defeated the world's greatest Go players.

Is AlphaGo smart? Its designers certainly are. They solved a math problem that was so hard that it took decades of great minds to work on it. One of the amazing things about math is that it allows you to see underlying patterns in how the world works. Many, many things operate according to mathematical patterns: crystals grow in regular patterns, and cicadas hibernate underground for years and emerge when soil temperature conditions are just right, to name just two. AlphaGo is a remarkable mathematical achievement that was made possible by equally remarkable advances in computing hardware and software. AlphaGo's team of designers deserves praise for this outstanding technical achievement.

AlphaGo is not an intelligent machine, however. It has no consciousness. It does only one thing: plays a computer game. It contains data from thirty million games played by amateurs and by the world's most talented players. On some level, AlphaGo is supremely dumb. It uses brute force and the combined effort of many, many humans to defeat a single Go master. The program and

its underlying computational methods will likely be deployed for other useful tasks involving massive number-crunching, and that's good for the world—but not everything in the world is a calculation.

Once we get past the mathematical and physical reality of a program like AlphaGo, we're in the realm of philosophy and future speculation. Those are very different intellectual landscapes. There are futurists who *want* AlphaGo to signify the beginning of an era in which people and machines become fused. Wanting something doesn't make it true, however.

Philosophically, there are lots of interesting questions to discuss centering on the difference between calculation and consciousness. Most people are familiar with the Turing test. Despite what the name suggests, the Turing test is not a quiz that a computer can pass to be considered intelligent. In his paper, Turing proposed a thought experiment about talking to a machine. He rejected the question "Can machines think?" as absurd and claimed it was best answered by an opinion poll. (Turing was a bit of a snob about math. Like many mathematicians then and a smaller number now, he believed in the superiority of mathematics to other intellectual pursuits.) Instead, Turing proposed an "imitation game" played by a man (A), a woman (B), and an interrogator (C). C sits in a room alone and submits typewritten queries to A and B. Turing writes: "The object of the game for the interrogator is to determine which of the other two is the man and which is the woman. He knows them by labels X and Y, and at the end of the game he says either 'X is A and Y is B' or 'X is B and Y is A.'"[2]

Turing then breaks down the kind of kind of questions the interrogator is allowed to ask. One is about hair length. A, the man, wants the interrogator to make the wrong assumptions and is willing to lie. B, the woman, wants to help the interrogator and can tell him or her that she is the woman—but A can lie and say that as well. Their answers are written down, so that the quality and tone of voice cannot provide clues. Turing writes: "We now ask the question, 'What will happen when a machine takes the part of A in this game?' Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and a woman? These questions replace our original, 'Can machines think?'"

If the questioner can't tell the difference between a response provided by a human or the response provided by a machine, the computer is said to be *thinking*. For many years, this was considered foundational in computing. A vast amount of ink has been spilled trying to respond to Turing's ideas in this paper and to make a machine that can perform to Turing's specifications. However, undergirding the entire thought experiment is a philosophical and cultural misnomer that throws the entirety into question, and that is gender. Turing's specifications do not conform to what we now understand about gender. Gender is not a binary, but a continuum. Hair length is no longer a signifier of male or female identity; anyone can rock a short haircut. Moreover, as Turing writes, "The object of the game for the third player (B) is to help the interrogator." A game to determine "intelligence," in which the woman is assigned to be the helper? And the man is told that he can lie? The underpinnings are absurd, from a critical perspective, in that both the man and woman are given gender-coded physical and moral attributes.

The philosophical underpinnings of Turing's argument are unsound. One of the most compelling counterarguments was addressed by the philosopher John Searle in an argument known as the Chinese Room. Searle summarized it in a 1989 piece in the *New York Review of Books*:

> A digital computer is a device which manipulates symbols, without any reference to their meaning or interpretation. Human beings, on the other hand, when they think, do something much more than that. A human mind has meaningful thoughts, feelings, and mental contents generally. Formal symbols by themselves can never be enough for mental contents, because the symbols, by definition, have no meaning (or interpretation, or semantics) except insofar as someone outside the system gives it to them.
>
> You can see this point by imagining a monolingual English speaker who is locked in a room with a rule book for manipulating Chinese symbols according to computer rules. In principle he can pass the Turing test for understanding Chinese, because he can produce correct Chinese symbols in response to Chinese questions. But he does not understand a word of Chinese, because he does not know what any of the symbols mean. But if he does not understand Chinese solely by virtue of running the computer program for "understanding" Chinese, then neither does any other digital computer because no computer just by running the program has anything the man does not have.[3]

Searle's argument that symbolic manipulation is not equivalent to understanding can be seen in the popularity of voice interfaces in 2017. "Conversational" interfaces are popular, but they are far from intelligent.

Amazon's Alexa and other voice-response interfaces don't understand language. They simply launch computerized sequences in response to sonic sequences, which humans call verbal commands. "Alexa, play 'California Girls'" is a voice command that a computer can follow. *Alexa* is the trigger word that tells the computer that a command is coming. *Play* is a trigger word that means "retrieve an MP3 from memory and send the command *play* to a previously specified audio player, along with the MP3 file name." The interface is also programmed to capture whatever word comes after *play* and before the pause (the end of the command). That value is put into a variable such as *songname*, which is retrieved from memory and fed to the audio player. This process is procedural and unthreatening and shouldn't make anyone think that the machines are going to rise up and take over the world. Right now, a

computer can't reliably distinguish whether it should respond to the previous command by playing Katy Perry's "California Gurls" or the Beach Boys' "California Girls." In fact, this exact problem is solved by running a popularity contest. Whichever song is played more often by all Alexa users is assumed to be the default choice. This is good for Katy Perry fans, but not so good for Beach Boys fans.

I'm going to ask you to keep the two competing ideas about narrow and general AI, and the idea of limitations, in your mind as you read. In this book, we'll stay squarely in the realm of reality: the world in which we have unintelligent computing machines that we *call* intelligent machines. However, we'll also look at how imagination—which is powerful and wonderful and exciting—sometimes confuses the way we talk about computers, data, and technology. I'd also ask you not to be disappointed like the art student at the science fair when you come up against what a colleague calls the *ghost-in-the-machine fallacy*—the reality that there is no little person or simulated brain inside the computer. There are different ways to react to this news: you can be sad that the thing you dreamed of is not possible—or you can be excited and embrace what *is* possible when artificial devices (computers) work in sync with truly intelligent beings (humans). I prefer the latter approach.

Notes

1. Silver et al., "Mastering the Game of Go with Deep Neural Networks and Tree Search," 484.
2. Turing, "Computing Machinery and Intelligence."
3. Searle, "Artificial Intelligence and the Chinese Room."