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.

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

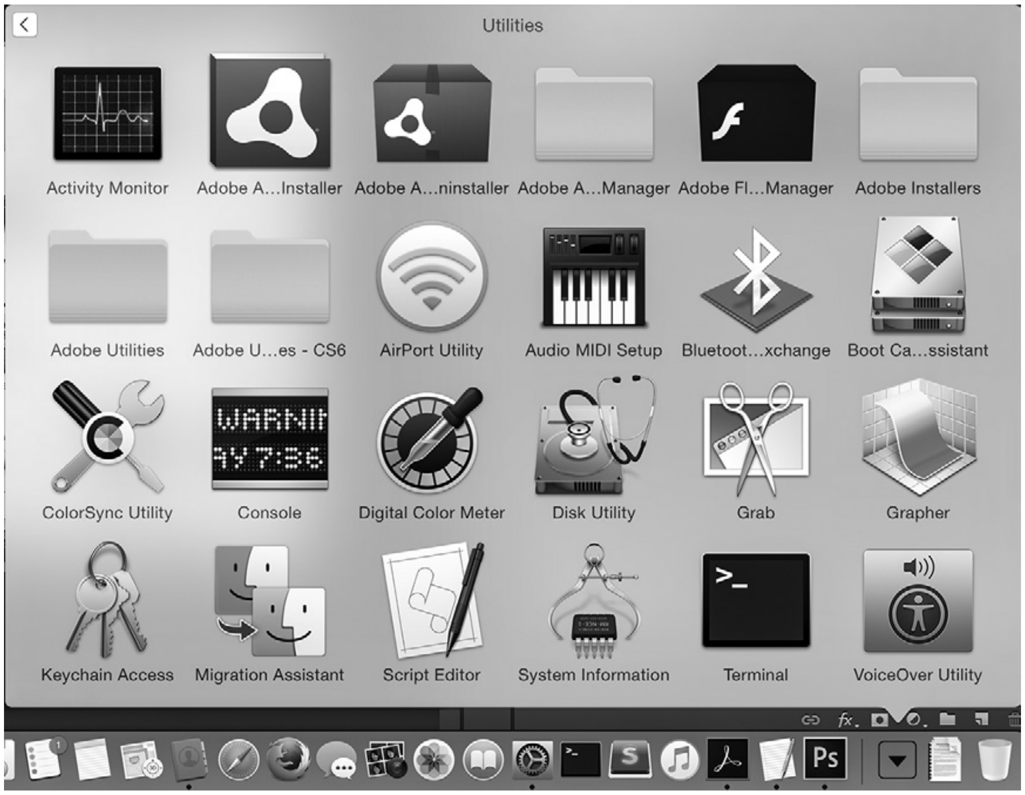

Figure 2.1
Terminal program in the Utilities folder.

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.

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.

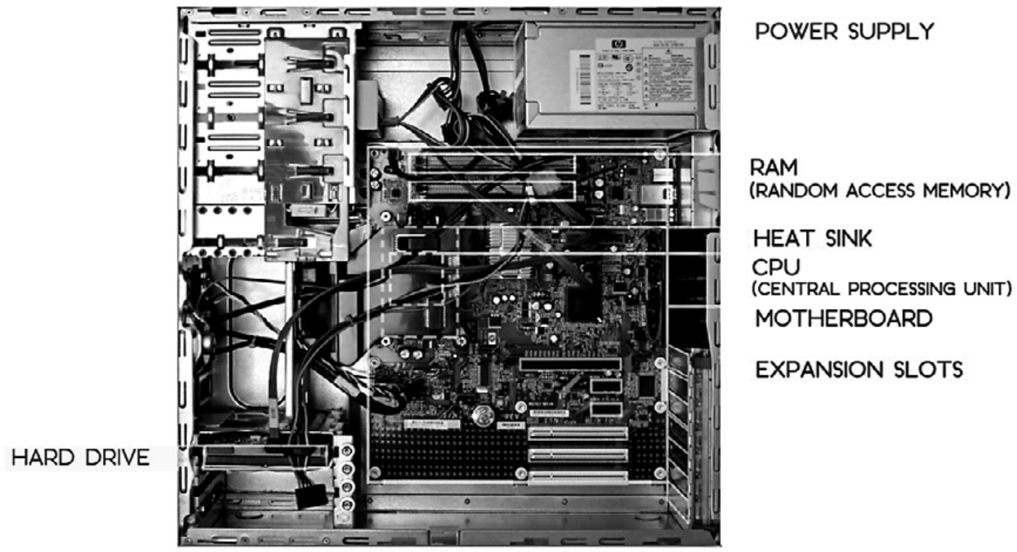

Figure 2.2
The innards of a desktop computer.

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).

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

Figure 2.3
A turkey club sandwich.

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?