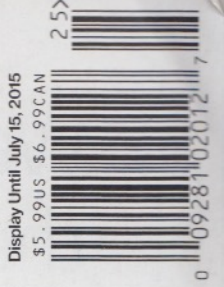
Bloomberg Businessweek

SPECIAL DOUBLE ISSUE June 15 — June 28, 2015 | bloomberg.com



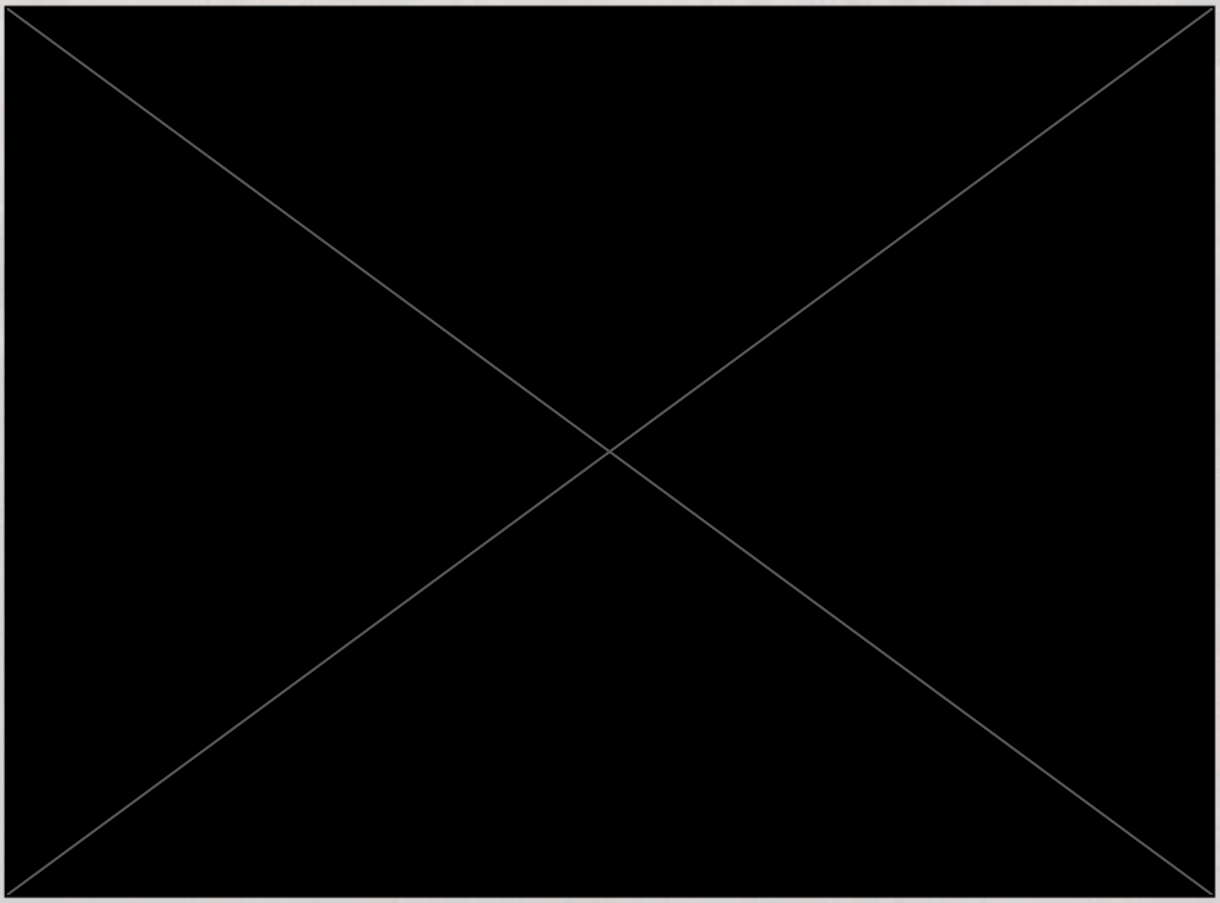

```
import datetime

class Issue():
    """TODO write docs here"""
    def __init__(self, **kwargs):
        # TODO: Validate input
        self.__dict__.update(kwargs)

    def publish(self):
        return ('This is the {0.pubdate:%B %d, %Y} issue of {0.title}. ' +
                'It is {0.pages:,} pages long, and ' +
                'costs ${0.price:.5}. '
                'It is about {0.subject}.').format(self)

if __name__ == '__main__':
    bbw = Issue(title='Bloomberg Businessweek',
                price=5.99,
                # That price is only USD;
                # TODO figure out international pricing/currencies
                pages=112,
                pubdate=datetime.datetime(2015, 6, 15),
                subject="code")
    print(bbw.publish())
```

If You Can't Read That, You'd Better Read This

Code: An Essay p.13

The Code Issue

By Paul Ford



On the cover (in Python 3): This is the June 15, 2015, issue of Bloomberg Businessweek. It's 112 pages long and costs $5.99. It's about code.

Let's Begin

A computer is a clock with benefits. They all work the same, doing second-grade math, one step at a time: Tick, take a number and put it in box one. Tick, take another number, put it in box two. Tick, *operate* (an operation might be addition or subtraction) on those two numbers and put the resulting number in box one. Tick, check if the result is zero, and if it is, go to some other box and follow a new set of instructions.

You, using a pen and paper, can do anything a computer can; you just can't do those things billions of times per second. And those billions of tiny operations add up. They can cause a phone to boop, elevate an elevator, or redirect a missile. That raw speed makes it possible to pull off not one but multiple sleights of hand, card tricks on top of card tricks. Take a bunch of pulses of light reflected from an optical disc, apply some math to unsqueeze them, and copy the resulting pile of expanded impulses into some memory cells–then read from those cells to paint light on the screen. Millions of pulses, 60 times a second. That's how you make the rubes believe they're watching a movie.●

Apple has always made computers; Microsoft used to make only software (and occasional accessory hardware, such as mice and keyboards), but now it's in the hardware business, with Xbox game consoles, Surface tablets, and Lumia phones. Facebook assembles its own computers for its massive data centers.

So many things are computers,● or will be. That includes watches, cameras, air conditioners, cash registers, toilets, toys, airplanes, and movie projectors. Samsung makes computers that look like TVs, and Tesla makes computers with wheels and engines. Some things that aren't yet computers–dental floss, flashlights–will fall eventually.

When you batch process a thousand images in Photoshop or sum numbers in Excel, you're programming, at least a little. When you use computers too much–which is to say a typical amount–they start to change you. I've had Photoshop dreams, Visio dreams, spreadsheet dreams, and Web browser dreams. The dreamscape becomes fluid and can be sorted and restructured. I've had programming dreams where I move text around the screen.

You can make computers do wonderful things, but you need to understand their limits. They're not all-powerful, not conscious in the least. They're fast, but some parts–the processor, the RAM–are faster than others–like the hard drive or the network connection. Making them seem infinite takes a great deal of work from a lot of programmers and a lot of marketers.

The turn-of-last-century British artist William Morris once said you can't have art without resistance in the materials. The computer and its multifarious peripherals are the materials. The code is the art.

● We're all rubes now.

● Which of the following is a computer?

- ○ a. Mugs
- ○ b. Cars
- ○ c. Thermostats
- ○ d. Stereos
- ○ e. ATMs
- ○ f. Cameras
- ○ g. Dinner forks
- ○ h. Tennis rackets
- ○ i. Toilets
- ○ j. All of the above

How Do You Type An "A"?

Consider what happens when you strike a key on your keyboard. Say a lowercase "a." The keyboard is waiting for you to press a key, or release one; it's constantly scanning to see what keys are pressed down. Hitting the key sends a scancode.

Just as the keyboard is waiting for a key to be pressed, the computer is waiting for a signal from the keyboard. When one comes down the pike, the computer interprets it and passes it farther into its own interior. "Here's what the keyboard just received–do with this what you will."

It's simple now, right? The computer just goes to some table, figures out that the signal corresponds to the letter "a," and puts it on screen. Of course not–too easy. Computers are machines. They don't know what a screen or an "a" are. To put the "a" on the screen, your computer has to pull the image of the "a" out of its memory as part of a font, an "a" made up of lines and circles. It has to take these lines and circles and render them in a little box of pixels in the part of its memory that manages the screen. So far we have at least three representations of one letter: the signal from the keyboard; the version in memory; ▶

Answer: j. All of the above

and the lines-and-circles version sketched on the screen. We haven't even considered how to store it, or what happens to the letters to the left and the right when you insert an "a" in the middle of a sentence. Or what "lines and circles" mean when reduced to binary data. There are surprisingly many ways to represent a simple "a." It's amazing any of it works at all.●

Coders are people who are willing to work backward to that key press. It takes a certain temperament to page through standards documents, manuals, and documentation and read things like "data fields are transmitted least significant bit first" in the interest of understanding why, when you expected "ü," you keep getting "  ."

"It's amazing any of it works at all" ●

From Hardware To Software

Hardware is a tricky business. For decades the work of integrating, building, and shipping computers was a way to build fortunes. But margins tightened. Look at Dell, now back in private hands, or Gateway, acquired by Acer. Dell and Gateway, two world-beating companies, stayed out of software, typically building PCs that came preinstalled with Microsoft Windows—plus various subscription-based services to increase profits.

This led to much cursing from individuals who'd spent $1,000 or more on a ▶

PHOTOGRAPH BY BORIS O'BRIEN O'CONNELL FOR BLOOMBERG BUSINESSWEEK; SET DESIGN: DAVE BRYANT

computer and now had to figure out how to stop the antivirus software from nagging them to pay up.

Years ago, when Microsoft was king, Steve Ballmer, sweating through his blue button-down, jumped up and down in front of a stadium full of people and chanted,

"Developers!
Developers!
Developers!
Developers!"

He yelled until he was hoarse: "I love this company!" Of course he did.● If you can sell the software, if you can light up the screen, you're selling infinitely reproducible nothings. The margins on nothing are great–until other people start selling even cheaper nothings or giving them away. Which is what happened, as free software-based systems such as Linux began to nibble, then devour, the server market, and free-to-use Web-based applications such as Google Apps began to serve as viable replacements for desktop software.

Expectations around software have changed over time. IBM unbundled software from hardware in the 1960s and got to charge more; Microsoft rebundled Internet Explorer with Windows in 1998 and got sued; Apple initially refused anyone else the ability to write software for the iPhone when it came out in 2007, and then opened the App Store, which expanded into a vast commercial territory–and soon the world had Angry Birds. Today, much hardware comes with some software–a PC comes with an operating system, for example, and that OS includes hundreds of subprograms, from mail apps to solitaire. Then you download or buy more.

There have been countless attempts to make software easier to write, promising that you could code in plain English, or manipulate a set of icons, or make a list of rules–software development so simple that a bright senior executive or an average child could do it. Decades of efforts have gone into helping civilians write code as they might use a calculator or write an e-mail. Nothing yet has done away with *developers, developers, developers, developers.*

Thus a craft, and a professional class that lives that craft, emerged. Beginning in the 1950s, but catching fire in the 1980s, a proportionally small number of people became adept at inventing ways to satisfy basic human desires (know the time, schedule a flight, send a letter, kill a zombie) by controlling the machine. Coders, starting with concepts such as "signals from a keyboard" and "numbers in memory," created infinitely reproducible units of digital execution that we call software, hoping to meet the needs of the marketplace. Man, did they. The systems they built are used to manage the global economic infrastructure.■1 If coders don't run the world, they run the things that run the world.

Most programmers aren't working on building a widely recognized application like Microsoft Word. Software is everywhere. It's gone from a craft of fragile, built-from-scratch custom projects to an industry of standardized parts, where coders absorb and improve upon the labors of their forebears (even if those forebears are one cubicle over). Software is there when you switch channels and your cable box shows you what else is on. You get money from an ATM–software. An elevator takes you up five stories–the same. Facebook releases software every day to something like a billion people, and that software runs inside Web browsers and mobile applications. Facebook looks like it's just pictures of your mom's crocuses or your son's school play–but no, it's software.

How Does Code Become Software?

We know that a computer is a clock with benefits, and that software starts as code, but how?

We know that someone, somehow, enters a program into the computer and the program is made of code. In the old days, that meant putting holes in punch cards.●

● Ballmer

● A history of input media:

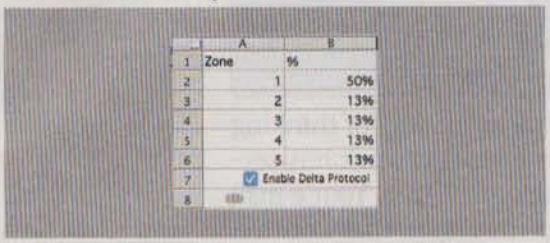
■1 Not bad for six or seven decades–but keep it in perspective. Software may be eating the world, but the world was previously eaten by other things, too: the rise of the telephone system, the spread of electricity, and the absolute domination of the automobile. It's miraculous that we have mobile phones, but it's equally miraculous that we can charge them.

Then you'd put the cards into a box and give them to an operator who would load them, and the computer would flip through the cards, identify where the holes were, and update parts of its memory, and then it would–OK, that's a little too far back. Let's talk about modern typing-into-a-keyboard code. It might look like this:

```
ispal: {x-|x}
```

That's in a language called, simply, K, famous for its brevity.■2 That code will test if something is a palindrome. If you next typed in `ispal "able was i ere i saw elba"`, K will confirm that yes, this is a palindrome.

So how else might your code look? Maybe like so, in Excel (with all the formulas hidden away under the numbers they produce, and a check box that you can check):



	A	B
1	Zone	%
2	1	50%
3	2	13%
4	3	13%
5	4	13%
6	5	13%
7	☑ Enable Delta Protocol	
8		

But Excel spreadsheets are tricky, because they can hide all kinds of things under their numbers. This opacity causes risks. One study by a researcher at the University of Hawaii found that 88 percent of spreadsheets contain errors.

Programming can also look like Scratch, a language for kids:

That's definitely programming right there–the computer is waiting for a click, for some input, just as it waits for you to type an "a," and then it's doing something repetitive, and it involves hilarious animals.

Or maybe:

```
PRINT *, "WHY WON'T IT WORK
END
```

That's in Fortran. The reason it's not working is that you forgot to put a quotation mark at the end of the first line. Try a little harder, thanks.

All of these things are coding of one kind or another, but the last bit is what most programmers would readily identify as code. A sequence of symbols (using typical keyboard characters, saved to a file of some kind) that someone typed in, or copied, or pasted from elsewhere. That doesn't mean the other kinds of coding aren't valid or won't help you achieve your goals. Coding is a broad human activity, like sport, or writing. When software developers think of coding, most of them are thinking about lines of code in files. They're handed a problem, think about the problem, write code that will solve the problem, and then expect the computer to turn word into deed.

Code is inert. How do you make it ert? You run software that transforms it into machine language. The word "language" is a little ambitious here, given that you can make a computing device with wood and marbles. Your goal is to turn your code into an explicit list of instructions that can be carried out by interconnected logic gates,● thus turning your code into something that can be executed–software.

A compiler is software that takes the symbols you typed into a file and transforms them into lower-level instructions. Imagine a programming language called Business Operating Language United System, or Bolus. It's a terrible language that will have to suffice for a few awkward paragraphs. It has one real command, `PRINT`. We want it to print `HELLO NERDS` on our screen. To that end, we write a line of code in a text file that says:

```
PRINT {HELLO NERDS}
```

▶

● A logic gate is a physical thing—a circuit—that corresponds to the rules of Boolean logic. Take an AND gate:

It's got two wires to the left and one wire coming out. If you send one signal through the first wire? Nothing. Through the second? Nothing. Through both? Yes, the rightmost wire will transmit the signal. You can make a small table, called a truth table, to show how it all works:

A	B	OUT
-	-	-
-	+	-
+	-	-
+	+	+

There are different kinds of logic gates, each with its own truth table.

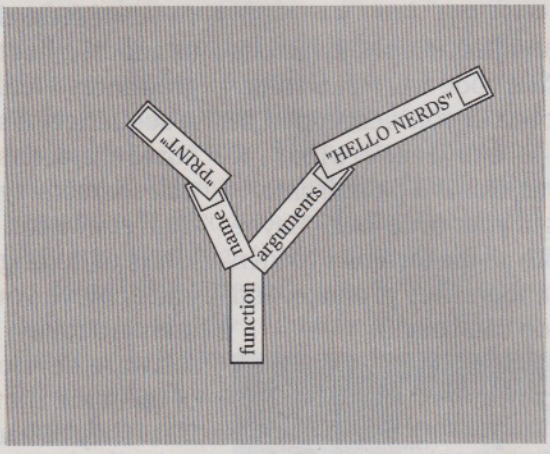
A computer is a bunch of gates wired together. You can make an "adder" to add up the signals, like doing grade-school arithmetic.

■2 The world of code is filled with acronyms. K is modeled on another language called APL, which stands for A Programming Language. Programmers are funny, like your uncle. They hold the self-referential and recursive in the highest regard. Another classic: GNU, which means GNU's Not Unix. Programmer jokes make you laugh and sigh at once. Or just sigh.

And we save that as `nerds.bol`. Now we run `gnubolus nerds.bol`, our imaginary compiler program. How does it start? The only way it can: by doing lexical analysis, going character by character, starting with the "p," grouping characters into tokens, saving them into our one-dimensional tree boxes. Let's be the computer.

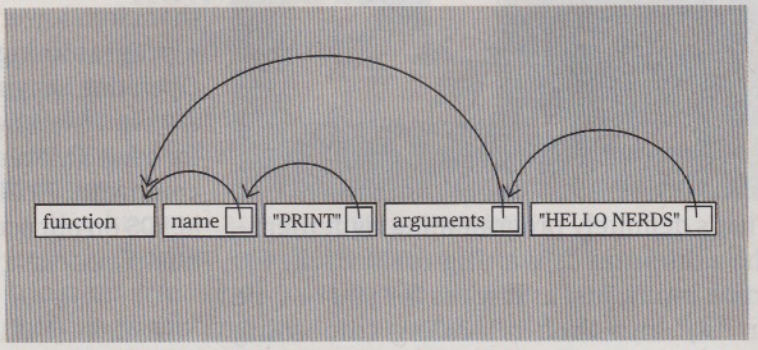
Character	Meaning
P	*Hmmmm...?*
R	*Someone say something?*
I	*I'm waiting...*
N	[drums fingers]
T	*Any time now...*
Space	*Ah, "PRINT"*
{	*String coming!*
H	*These*
E	*letters*
L	*don't*
L	*matter*
O	*la*
Space	*la*
N	*just*
E	*saving*
R	*them*
D	*for*
S	*later.*
}	*Stringtime is over!*
End of file	*Time to get to work.*

The reason I'm showing it to you is so you can see how every character matters. Computers usually "understand" things by going character by character, bit by bit, transforming the code into other kinds of code as they go. The Bolus compiler now organizes the tokens into a little tree. Kind of like a sentence diagram. Except instead of nouns, verbs, and adjectives, the computer is looking for functions and arguments. Our program above, inside the computer, becomes this:



Trees are a really pleasant way of thinking of the world. Your memo at work has sections that have paragraphs? Tree. Your e-mail program contains messages that contain subject lines and addresses? Tree. Your favorite software program that has a menu bar with individual items that have subitems? Tree. Every day is Arbor Day in Codeville.

Of course, it's all a trick. If you cut open a computer, you'll find countless little boxes in rows, places where you can put and retrieve bytes. Everything ultimately has to get down to things in little boxes pointing to each other. That's just how things work. So that tree is actually more like this:



Every character truly, truly matters. Every single stupid misplaced semicolon,● space where you meant tab, bracket instead of a parenthesis–mistakes can leave the ▶

● On April 13, 2012, someone filed a bug report on GitHub, asking for a single semicolon in some JavaScript code. The Great Semicolon Debate resulted in 291 comments, multiple blog posts, and commentary from the most influential people in the JavaScript community. If printed on one continuous sheet of 11-inch-wide paper, it would be as long as a sperm whale.

The thread was finally shut down in March 2014, but its legacy remains as a warning to us all.

computer in a state of panic. The trees don't know where to put their leaves. Their roots decay. The boxes don't stack neatly. For not only are computers as dumb as a billion marbles, they're also positively Stradivarian in their delicacy.

That process of going character by character can be wrapped up into a routine—also called a function, a method, a subroutine, or component. (Little in computing has a single, reliable name, which means everyone is always arguing over semantics.) And that routine can be run as often as you need. Second, you can print anything you wish, not just one phrase. Third, you can repeat the process forever, and nothing will stop you until the machine breaks or, barring that, heat death of the universe. Obviously no one besides Jack Nicholson in *The Shining* really needs to keep typing the same phrase over and over, and even then it turned out to be a bad idea.

Instead of worrying about where the words are stored in memory and having to go character by character, programming languages let you think of things like strings, arrays, and trees. That's what programming gives you. You may look over a programmer's shoulder and think the code looks complex and boring, but it's covering up repetitive boredom that's unimaginably vast.■3

This thing we just did with individual characters, compiling a program down into a fake assembly language so that the nonexistent computer can print each character one at a time? The same principle applies to every pixel on your screen, every frequency encoded in your MP3 files, and every imaginary cube in Minecraft. Computing treats human language as an arbitrary set of symbols in sequences. It treats music, imagery, and film that way, too.

It's a good and healthy exercise to ponder what your computer is doing right now. Maybe you're reading this on a laptop: What are the steps and layers between what you're doing and the Lilliputian mechanisms within? When you double-click an icon to open a program such as a word processor, the computer must know where that program is on the disk. It has some sort of accounting process to do that. And then it loads that program into its memory–which means that it loads an enormous to-do list into its memory and starts to step through it. What does that list look like?

Maybe you're reading this in print. No shame in that. In fact, thank you. The paper is the artifact of digital processes. Remember how we put that "a" on screen? See if you can get from some sleepy writer typing that letter on a keyboard in Brooklyn, N.Y., to the paper under your thumb. What framed that fearful symmetry?

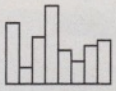
Thinking this way will teach you two things about computers: One, there's no magic, no matter how much it looks like there is.● There's just work to make things look like magic.● And two, it's crazy in there.

"There's no magic, no matter how much it looks like there is."●

What Is an Algorithm?

"Algorithm" is a word writers invoke to sound smart about technology. Journalists tend to talk about "Facebook's algorithm" or a "Google algorithm," which is usually inaccurate. They mean "software."

Algorithms don't require computers any more than geometry does. An algorithm solves a problem, and a great algorithm gets a name. Dijkstra's algorithm, after the famed computer scientist Edsger Dijkstra, finds the shortest path in a graph. By the way, "graph" here doesn't mean



but rather

Think of a map; streets connect to streets at intersections. It's a graph! There are graphs all around you. Plumbing, electricity, code compilation, social networks, the Internet, all can be represented as graphs! (Now to monetize...)

● **"It takes these very, very simple-minded instructions—'Go fetch a number, add it to this number, put the result here, perceive if it's greater than this other number'—but executes them at a rate of, let's say, 1,000,000 per second. At 1,000,000 per second, the results appear to be magic."**
—*Steve Jobs describing a computer to* Playboy *in 1985*

■3 Compilation is one of the denser subjects in computer science, because the lower down you go, the more opportunities there are to do deep, weird things that can speed up code significantly—and faster is cheaper and better. You can write elegant, high-level code like F. Scott Fitzgerald, and the computer will compile you into Ernest Hemingway. But compilers often do several passes, turning code into simpler code, then simpler code still, from Fitzgerald, to Hemingway, to Stephen King, to Stephenie Meyer, all the way down to Dan Brown, each phase getting less readable and more repetitive as you go.

Many algorithms have their own pages on Wikipedia. You can spend days poking around them in wonder. Euclid's algorithm, for example, is the go-to specimen that shows up whenever anyone wants to wax on about algorithms, so why buck the trend? It's a simple way of determining the greatest common divisor for two numbers. Take two numbers, like 16 and 12. Divide the first by the second. If there's a remainder (in this case there is, 4), divide the first, 16, by that remainder, 4, which gives you 4 and no remainder, so we're done—and 4 is the greatest common divisor. (Now translate that into machine code, and we can get out of here.)

There's a site called Rosetta Code that shows you different algorithms in different languages. The Euclid's algorithm page is great. Some of the examples are suspiciously long and laborious, and some are tiny nonsense poetry, like this one, in the language Forth:

```
: gcd ( a b -- n )
  begin dup while tuck mod repeat
drop ;
```
■4

Read it out loud, preferably to friends. Forth is based on the concept of a stack, which is a special data structure. You make "words" that do things on the stack, building up a little language of your own. PostScript,■5 the language of laser printers, came after Forth ▶

■4 I find code on the printed page to be hard to read. I don't blame you if your eyes blur. I try to read lots of code, but it makes more sense on the computer, where you could conceivably change parts of it and mess around. Every now and then I'll find some gem; the utility programs in the Unix source code are often amazingly brief and simple and obvious, everything you'd hope from a system that prides itself on being made up of simple, composable elements.

■5 Adobe created PostScript in the early 1980s and licensed it to Apple, its first success. Three-plus decades later, Adobe is valued at $38 billion. PDF is a direct descendant of PostScript, and there are PDFs everywhere. In code as in life, ideas grow up inside of languages and spread with them.

but is much like it. Look at how similar the code is, give or take some squiggles:

```
/gcd {
{
    {0 gt} {dup rup mod} {pop
exit} ifte
} loop
}.
```

And that's Euclid's algorithm in PostScript. I admit, this might be fun only for me. Here it is in Python (all credit to Rosetta Code):

```
def gcd(u, v):
    return gcd(v, u % v) if v \
    else abs(u)
```

A programming language is a system for encoding, naming, and organizing algorithms for reuse and application. It's an algorithm management system. This is why, despite the hype, it's silly to say Facebook has an algorithm. An algorithm can be translated into a function, and that function can be called (run) when software is executed. There are algorithms that relate to image processing and for storing data efficiently and for rapidly running through the elements of a list. Most algorithms come for free, already built into a programming language, or are available, organized into libraries, for download from the Internet in a moment. You can do a ton of programming without actually thinking about algorithms–you can save something into a database or print a Web page by cutting and pasting code. But if you want the computer to, say, identify whether it's reading Spanish or Italian, you'll need to write a language-matching function. So in that sense, algorithms can be pure, mathematical entities as well as practical expressions of ideas on which you can place your grubby hands.

One thing that took me forever to understand is that computers aren't actually "good at math." They can be programmed to execute certain operations to certain degrees of precision, so much so that it looks like "doing math" to humans.■6 Dijkstra● said: "Computer science is no more about computers than astronomy is about telescopes."■7 A huge part of computer science is about understanding the efficiency of algorithms–how long they will take to run. Computers are fast, but they can get bogged down–for example, when trying to find the shortest path between two points on a large map. Companies such as Google, Facebook, and Twitter are built on top of fundamental computer science■8 and pay great attention to efficiency, because their users do things (searches, status updates, tweets) an extraordinary number of times. Thus it's absolutely worth their time to find excellent computer scientists, many with doctorates, who know where all the efficiencies are buried.

It takes a good mathematician to be a computer scientist, but a middling one to be an effective programmer. Until you start dealing with millions of people on a network or you need to blur or sharpen a million photos quickly, you can just use the work of other people. When it gets real, break out the comp sci. When you're doing anything a hundred trillion times, nanosecond delays add up. Systems slow down, users get cranky, money burns by the barrel.■9

The hardest work in programming is getting around things that aren't computable, in finding ways to break impossible tasks into small, possible components, and then creating the impression that the computer is doing something it actually ▶

● Dijkstra distributed a remarkable and challenging set of memos to the global computer science community, starting in the 1960s and continuing up until his death in 2002, known as EWDs, many of them handwritten, which can be found at cs.utexas.edu/~EWD/.

■6 Two plus two usually equals four, but in a language like JavaScript if you add 0.4 + 0.2, the answer is 0.6000000000000001. That's because those numbers are interpreted as "floating point" (the point is the period), and the JavaScript language uses a particular way of representing those numbers in memory so that sometimes there are (entirely predictable) rounding errors. This is just one of those things that you have to know if you are a committed Web programmer.

■7 Well, he might have said it. It's attributed to him, but it might be folklore. Nonetheless, great quote!

■8 Meaning those companies are so huge that they can't use as much off-the-shelf, prepackaged code as the rest of us but rather need to rebuild things to their own very tight specifications.

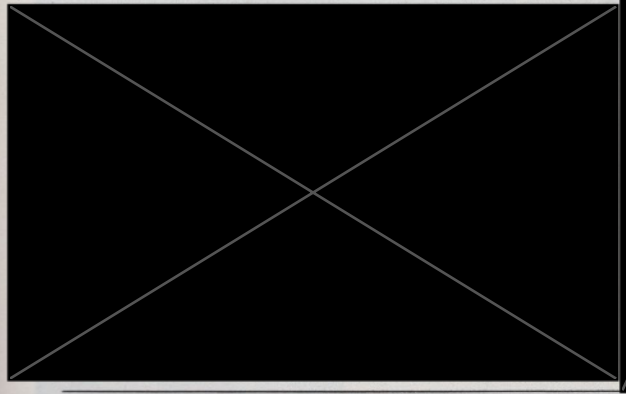
■9 By the way, that earlier assertion about how $100,000 in singles can fit in a barrel? It comes from a calculation made in Wolfram Alpha, a search engine that works well with quantities. The search was, "1 US dry barrel/volume of 1 US dollar banknote," and the result is 101,633.

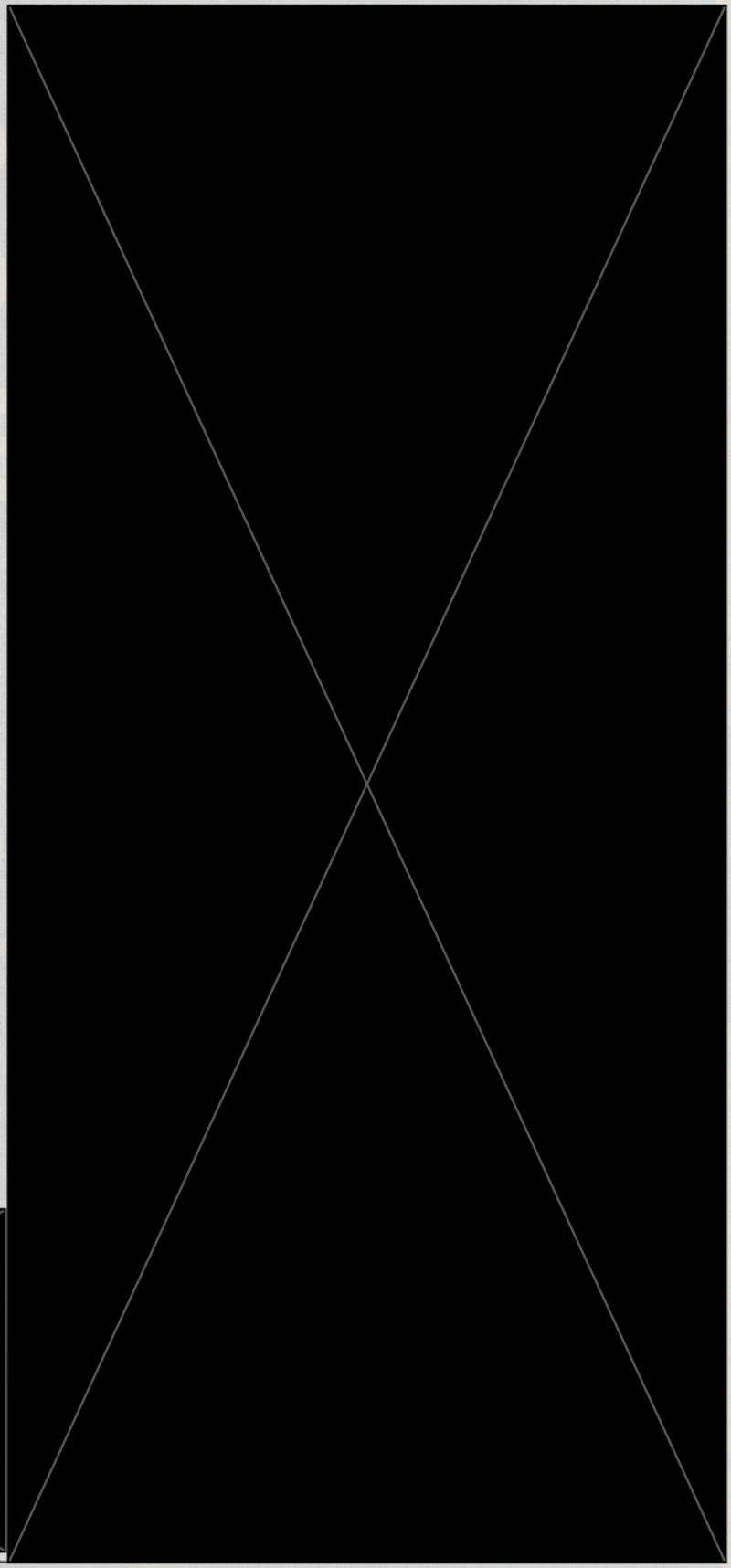
isn't, like having a human conversation. This used to be known as "artificial intelligence research," but now it's more likely to go under the name "machine learning" or "data mining." When you speak to Siri or Cortana and they respond, it's not because these services understand you; they convert your words into text, break that text into symbols, then match those symbols against the symbols in their database of terms, and produce an answer. Tons of algorithms, bundled up and applied, mean that computers can fake listening.

A programming language has at least two jobs, then. It needs to wrap up lots of algorithms so they can be reused. Then you don't need to go looking for a square-root algorithm (or a genius programmer) every time you need a square root. And it has to make it easy for programmers to wrap up new algorithms and routines into functions for reuse. The DRY principle, for Don't Repeat Yourself, is one of the colloquial tenets of programming. That is, you should name things once, do things once, create a function once, and let the computer repeat itself. This doesn't always work. Programmers repeat themselves constantly. I've written certain kinds of code a hundred times. This is why DRY is a principle.

Enough talk.

Let's code!





■10 User stories are often written on paper cards and arranged on a wall; they can also be two-dimensional computerized cards that are then moved around with a mouse and "assigned" to programmers.

ILLUSTRATION BY SIMON ABRAMOWICZ