Code to joy

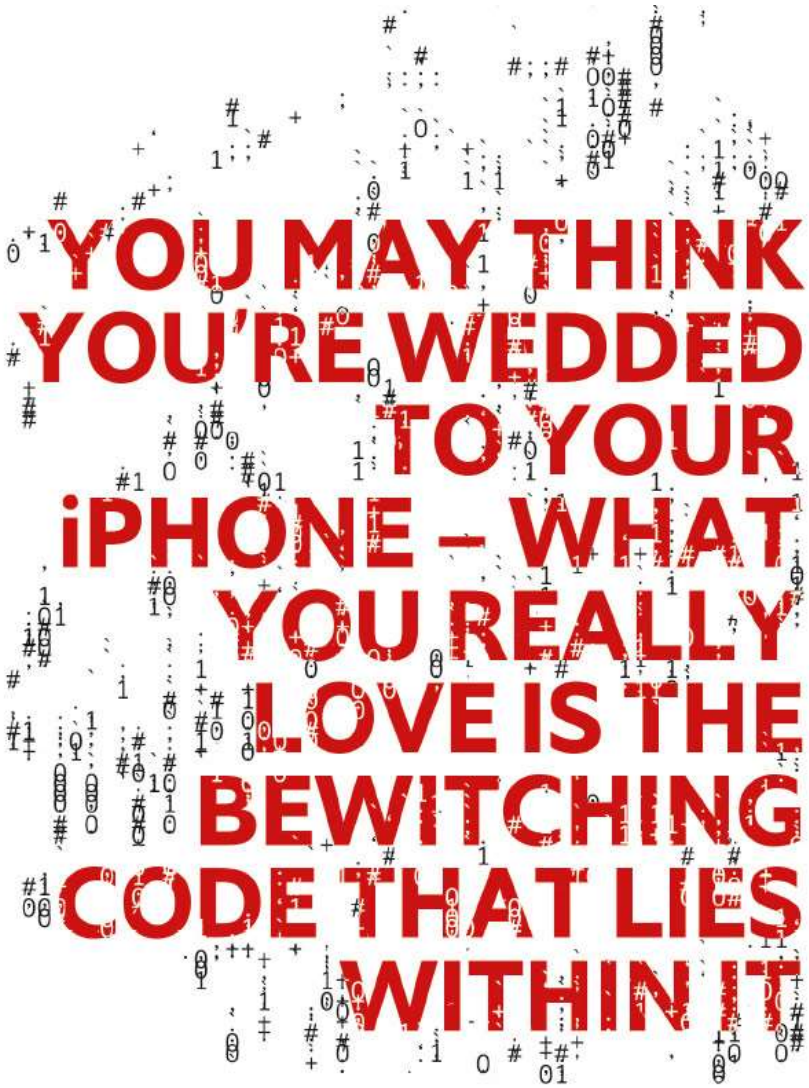
Is learning to code in middle age a fool's errand or a committed act of digital citizenship?

Is learning to code in middle age a fool's errand or a committed act of digital citizenship?

I remember the moment when code began to interest me. It was the tail end of 2013 and a cult was forming around a mysterious "crypto-currency" called bitcoin in the excitable tech quarters of London, New York and San Francisco. None of the editors I spoke to had heard of it – very few people had back then – but eventually one of them commissioned me to write the first British magazine piece on the subject. The story is now well known. The system's pseudonymous creator, Satoshi Nakamoto, had appeared out of nowhere, dropped his ingenious system of near-anonymous, decentralised money into the world, then vanished, leaving only a handful of writings and 100,000 lines of code as clues to his identity. Not much to go on. Yet while reporting the piece, I was astonished to find other programmers approaching Satoshi's code like literary critics, drawing conclusions about his likely age, background, personality and motivation from his style and approach. Even his choice of programming language – C++ – generated intrigue. Though difficult to use, it is lean, fast and predictable. Programmers choose languages the way civilians choose where to live and some experts suspected Satoshi of not being "native" to C++. By the end of my investigation I felt that I knew this shadowy character and tingled with curiosity about the coder's art. For the very first time I began to suspect that coding really was an art, and would reward examination.

Machines drove the first Industrial Revolution. The power of the one roiling our world now lies not in silicon and brushed steel but in a teeming cosmos of software: the code conjured by an invisible cadre of programmers. Unlike the achievements of previous revolutions, what code does is near impossible for most people to understand. Museums around the world memorialise the age of steam; picture books explain to children how engines work. It is hard to imagine how you might curate an exhibition on the age of code. Yet though most of us barely give it a thought, our relationship with code has become symbiotic, governing nearly every aspect of our lives. The accelerator in your new car no longer has

any physical connection to the throttle – the motion of your foot will be converted into binary numbers by some of the 100m lines of code that tell the vehicle what to do. Turn on your TV or radio, use a credit card, check in a bag at the airport, change the temperature in your fridge, get an X-ray at the dentist, text a family member, listen to music on anything other than vinyl or read this article online and each of your desires will be fulfilled by code. You may think you're wedded to your iPhone – what you really love is the bewitching code that lies within it.



Like most people, I have spent my life entirely ignorant of how the code that increasingly corrals me actually works. I know that microprocessors use tiny electrical switches to create and manipulate great waves of 0s and 1s. But how do these digits mesh with the world? Programmers speak to machines in impenetrable "languages" and coding, as the pop-cultural cliché suggests, appeared to be the impregnable domain of spectrumy maths geniuses. Though code makes our lives easier and more efficient, it is becoming increasingly

apparent how easily it can be turned to malign purposes. It's used by terrorists to spread viruses, car manufacturers to cheat emissions tests and hostile powers to hack elections. As the line between technology and politics blurs, I wondered whether my ignorance of its workings compromised my capacity to understand what it should and should not do. Being unable to speak to these mavens in their own terms, as they encoded the parameters of my world, brought a sense of helplessness for which only I, as a citizen, could take responsibility. And two questions that embraced all the others began to form. Should I learn to code? Could I learn to code? With a trepidation I later came to recognise as deeply inadequate, I decided there was only one way to find out.

There are now more than 1,700 computer languages that enable human desires to be translated into the only language a computer understands: numbers and logic. Most have been written by unpaid, clever individuals out of some opaque mix of glory and the hell of it. If you want to program a computer, you have to learn one of them. A daunting task, you might think. But even before that, you must engage in the hair-pulling frustration of picking your language.

To make my choice I trawl the web and consult every programmer I know. Each person either hedges or contradicts the last one. At first the litany of names sound like cleaning products or varieties of roses – Perl, Python, C, Ruby, Java, PHP, Cobol, Lisp, Pascal, Fortran. All have different specialisations and affinities that I feel unprepared to assess. Nevertheless, aided by websites that rank languages in terms of popularity on a monthly basis (the pace of coding fashion makes haute couture look agricultural), three scroll into view as not-stupid options: Python, JavaScript and C++. All are widely employed and on the rise, and have lots of fans and learning resources. Which to go for?

On a trip to New York I beg help from someone I trust. Paul Ford is co-founder of Postlight, a digital-production studio based on 5th Avenue. He is also the author of a superb and improbably entertaining novella-sized essay entitled "What is Code?" If anyone can help me choose a language, it's him. The son of an experimental poet, Ford came to programming late, after a career in journalism. Unlike most single-track peers, he can empathise with my discombobulation.

Computer scientists use the metaphor of a stack to describe how directly a language communicates with the computer hardware. Lowest in the stack is machine code. It consists of orders issued directly to a silicon chip, where tiny electrical switches called logic gates create and process binary and hexadecimal numbers. At the very top are beginner's languages such as the child-friendly

Scratch, where most of the machine's weirdness is hidden behind user-friendly shortcuts and guardrails designed to prevent mistakes. But ease of use comes at the expense of fine control and speed of processing. In an inversion of civilian usage, coders employ the soubriquet "low-level" to describe the difficult languages that engage computer processors most directly; and "high-level" to denote the more accessible ones further up the stack. When I meet Ford, I'm still reeling from my terror-breeding assumption that "low-level" was the best entry point for a beginner.

"That stuff is not comfortable, is it?" Ford says with a grimace when I ask if even a seasoned coder can translate seamlessly between different levels. "A mature programmer can go from very high in the stack to very low in the stack and explain how the pieces work. But that's maturity: there are really good [software] developers who, once you get below the level of what the web browser is doing, have no idea."

He points me in the direction of a *Playboy* interview Steve Jobs gave to coincide with the launch of the Apple Macintosh home computer in 1985. Jobs compares computation to showing the way to the toilet to someone with no sense of direction or instinctual control of their own body: "I would have to describe it…in very specific and precise instructions. I might say, 'Scoot sideways two metres off the bench. Stand erect. Lift left foot. Bend left knee until it is horizontal. Extend left foot and shift weight 300 centimetres forward…' and on and on." Computers operate on the same principle, just a million times quicker. "They don't know how to do anything except add numbers very fast," Ford concludes.

Ford steers me away from C++, which he likens to a shotgun: fast, efficient and ready to blow your foot off. If a misplaced semi-colon indicates a wish to erase the British archive of a major Hollywood movie studio – a friend of a friend claims this happened to him – C++ will follow through with unquestioning obedience. Not for me, I think.

JavaScript, by contrast, has the advantage of ubiquity: it is high level and serves as the primary language of interactive web pages. But it's also hard to think your way into, which is the price it pays for having evolved chaotically – like the web – with no central authority. Devotees claim that JavaScript is improving rapidly, which sounds great until you consider that your hard-won skills might be obsolete in six months' time. This fear haunts all coders and explains why they defend their own languages so fiercely, in what techies only half-jokingly refer to as "Religious Wars". Like a lot of professional coders, Ford uses JavaScript and

Python. He admires the latter, he says, for its svelte, modern syntax, its relative logic and versatility and the proactive disciples who generate lots of useful support. It has been used for everything from website development to algorithmic trading on the stockmarket.

In the course of our discussion, Ford offers nothing to allay the monumental nature of my choice. Languages are not expressive, he explains, but they do form cultures with attendant ways of seeing and being. My choice isn't simply a practical matter. Much like the musical decisions I made as a teenager, I have to ask myself who I want to hang with. Python was named after "Monty Python's Flying Circus" and within the community is considered hip. All of which I find discreditably appealing – until, out of the blue, Ford asks a question I hadn't considered.

"What do you want to do?"

Do? What? Of course. Programming is about making things happen. My mind goes blank, until slowly an answer emerges. A few years ago someone built a website for me but it doesn't work and I would love a flashy replacement. And if I want to work on the web, most roads lead to JavaScript.

On the internet, I discovered a helpful organisation called freeCodeCamp. Founded in 2014 by Quincy Larson, an idealistic schoolteacher turned programmer, freeCodeCamp began with the explicit goal of rationalising the learning process in order to deepen the otherwise shockingly shallow coding gene pool, which is around 95% white or Asian male. Larson established a free, user-friendly online course, alongside a growing international network of self-organising local groups. Thousands have used freeCodeCamp as a stepping stone to software-development careers.

The freeCodeCamp course begins with HTML5, or "Hypertext Markup Language", the basic design language of the web. I soon learn that coders don't regard HTML as a programming language, because it contains no algorithms, the pieces of code that tell computers how to process different inputs. Simple algorithmic logic is defined by three words: "if", "then", and "else". An example of an algorithm would be "if the customer orders size 13 shoes, THEN display the message 'Sold out, Sasquatch!'; ELSE ask for a colour preference." Programs, according to Paul Ford, are no more than "recipes which wrap up lots of algorithms so they can be re-used".

HTML does not do this. Instead, it tells your browser how to organise the elements of a web page – the headers, subheads, images and paragraphs. For all this, HTML does make use of the same symbolic lexicon as a programming language, and in that sense counts as code. I would recommend anyone to spend an hour or two on freeCodeCamp, simply to know the joy of feeling properly in charge of a computer. There is also a real thrill to learning how the scrambled algebra-like snippets we've all seen in passing, so inert and intimidating out of context, actually work. One of my favourite revelations is that all those tech brands with names that consist of elided capitalised words, like AirBnB, iPhone and freeCodeCamp, are not just trying to be cute: this style is called CamelCase and is commonly used in programming environments, where spaces are not always neutral.

During the six weeks I spend building a mini-website, I learn some valuable lessons. The most important is that computers are insanely literal and pedantic. An absent colon or extraneous bracket is easy to miss and is likely to cause the machine to stare at you uncomprehendingly, metaphorical fingers drumming on the table. One high-end programmer tells me how she spent six months hunting a bug within a big system, knowing others had tried and failed, finally tracking the problem to an underscore that should have been a dash. A shift-devil, she calls it. No wonder coders can appear nervy and terse to outsiders.

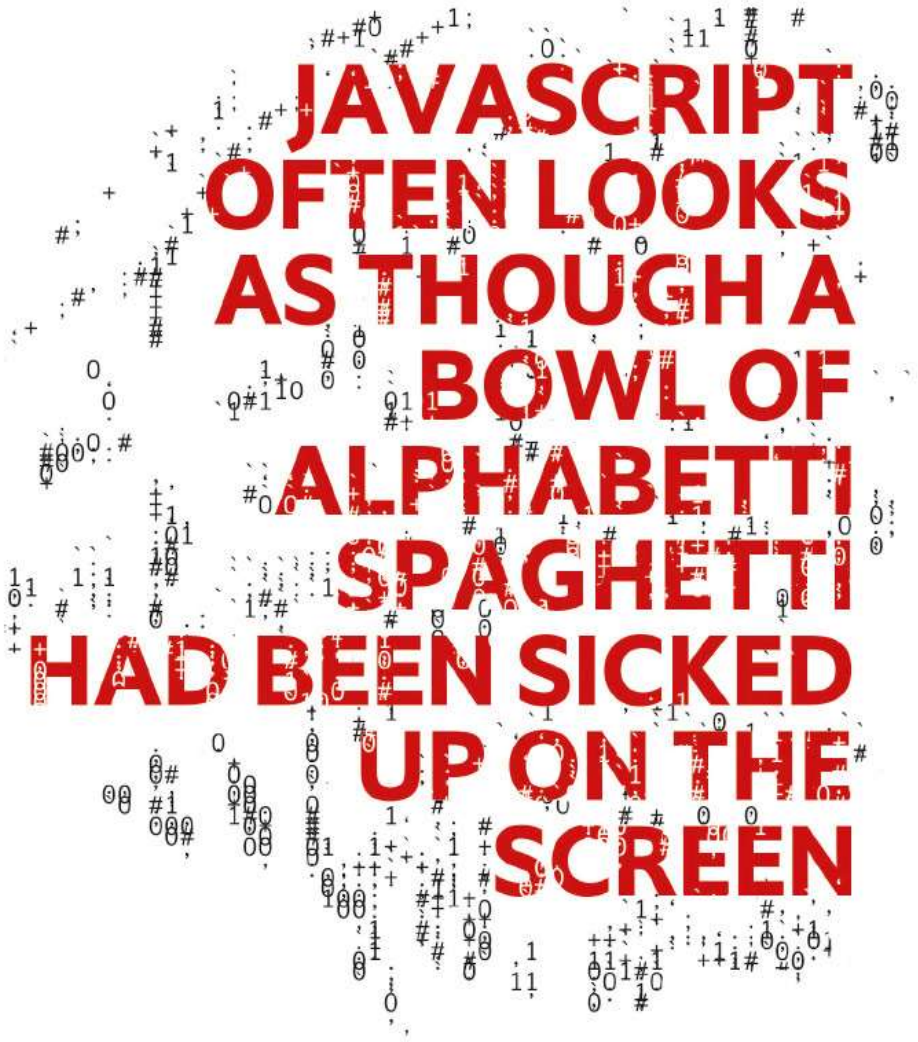
Another lesson is that the decisions you make about programming are not clear-cut. An experienced coder points out that designing my own website from scratch is anachronistic in the age of off-the-peg providers like Squarespace and Wordpress. Third, HTML goes hand in hand with JavaScript, which animates web pages. Try as I might, I am not enjoying the latter. I'm frustrated by its messiness and struggle to think my way into it. Perhaps coding is not going to be for me after all.

Aware that mature heads have been nudging me towards Python from the start, I return to one of these early advisers, who puts me in touch with a British programmer called Nicholas Tollervey, a graduate of the Royal College of Music and former professional tuba player. He has additional degrees in philosophy, computing and education – not like your stock coder cowboy. Fluent in a range of languages, he is a particular advocate for Python, which he teaches and develops tools for. Is there a reason people keep hinting I should try Python again, I ask?

"Well, I don't want to get into Religious Wars here," he says, "so I should say that I enjoy both. But yes, there is a reason."

At the turn of the 1990s, Guido van Rossum, Python's Dutch creator, set out to write a programming language that would be clear, concise and as close to natural English as possible. Simplicity was paramount and he wanted only one obvious solution to most problems. This was in explicit contrast to languages such as JavaScript that have an almost comic absence of overarching logic and syntactic elegance, and a baffling number of ways to do the same thing. Python does not encourage showboaters or individualists, because it prizes communication and collaboration. In Van Rossum's words, "readability counts". Intrigued, I decide to adapt a suggestion made months ago by Ford, setting myself the goal of writing a Python-based app with which authors can scrape Twitter for mentions of their books. Tollervey wishes me well and offers help if I need it. Together we decide on the app's name: BookOmatic9000.

I instantly appreciate the concision of Van Rossum's layout, which, like written prose and many other programming languages, uses indentation to delineate blocks of code rather than littering the screen with symbols (such as arrays of brackets and braces) to achieve the same result. JavaScript often looks as though a bowl of Alphabetti Spaghetti had been sicked up on the screen; Python is uncluttered and comparatively easy to take in at a glance. I am shocked to understand that my affinity for Python is primarily an aesthetic matter, a quirk of my own temperament. I'm beginning to understand why it is that particular character types cluster around different languages.



The app I want to write will rove Twitter feeds looking for keywords provided by a user. The algorithm I am creating effectively tells the machine: “If you find word or phrase *a*, look for word or phrase *b*; if not, move to the next tweet; repeat until you find a tweet containing both *a* and *b*, then display the tweet’s text, date and author on screen.” Once I have this simple program up and running, I can upgrade it to return more results and save them to a Python data structure called a dictionary.

To my regret, freeCodeCamp doesn’t yet teach Python, so I’ve settled for an online “Beginning Python” tutorial series and a well-reviewed book called “Python Crash Course”. Most importantly, I can call Nicholas Tollervey whenever I get stuck. I soon find that, for all the dizzying range of online resources, there remains no substitute for the expert eye over the shoulder.

A month down the line I have learnt – or think I've learnt – the key concepts I'm going to need. Fundamental to the algorithmic thinking required from programming are variables, functions and loops. Variables are best (if imperfectly) understood as the vessels within which pieces of data are contained, ready to be worked on. Of many possible data types, the most straightforward are numbers and strings, string being the name given to text.

Functions are the particular operations that can be performed on data, such as ordering a list or counting items. In a bracing low-level language like C++, a task like capitalising names would involve writing multiple lines of code each time the operation was required. Python offers a simpler option. Coders can either write a function themselves to perform the action each time. Or, if the function already exists, they can import it from an official repository known to Pythonistas as the Cheese Shop after the Monty Python sketch of the same name (in which the shop has no cheese). In addition, Python comes with an extensive range of built-in functions, known as methods, which can solve common problems. Finally, a loop asks the computer to keep repeating a block of code until a particular set of conditions is met. For example, "keep searching until an author's surname and the title of one of her books are found in the same tweet".

I reckon that I've learnt enough to build my app. But then something extraordinary happens. I look at my blank code editor and my head empties. I look longer and – by some strange quantum effect, I assume – my head seems to empty further. With the training wheels off, I suddenly feel that I know nothing and am panicked by the sensation. FreeCodeCamp's Larson chuckles when I tell him about this phenomenon.

"The thing that gets lost," he says, "and which I think is important to know, is that programming is never easy. You're never doing the same thing twice, because code is infinitely reproducible and if you've already solved a problem and you encounter it again, you just use your old solution. So by definition you're kind of always on this frontier where you're out of your depth. And one of the things you have to learn is to accept that feeling – of being constantly wrong."

Which makes coding sound like a branch of Zen Buddhism.

I regroup and accept the advice of a developer who reveals to me that, for all the ubiquity of screens in the coding realm, he sketches the structure of his apps on paper. This is common practice. Coders also imagine "stories" pertaining to a user's probable transit through an app or page, in order to understand what features are needed and how they will fit together. I do this and see that, first and

foremost, I must learn how to connect with Twitter's API, or Application Programming Interface, which provides developers with access to the company's feed. I must also become familiar with Tweepy, a library of Python tools specially written to talk to Twitter. To this end I spend an entire exhausting day reading the copious online documentation about this software. Tolstoy must look like a quick skim to these people. Needless to say, my first two tasks turn out to be anything other than straightforward. But with patience and the odd late-night SOS to Tollervey, I am at length connected to the social network's feed and can use functions from Tweepy as needed. Now to build the app.

Central to BookOmatic's operation will be the user's input of keywords. A simple input function is already embedded in Python. I use this to prompt for the book title and author surname, which will be combined into a variable named "search_term" and sent to the Twitter API. When the results come back, I will extract the information I want and print the results to screen. Sounds easy. Except that I start wondering whether a full name rather than surname, or a Twitter handle, might be more effective; whether some extra terms like "book", "read" or "reading", hidden in the background, to run unseen on all searches, might increase accuracy. While I fiddle, hours fly by with minimal improvement, until the changes I've made start to affect the app, which grinds to a halt. By the time I can admit my hubris to myself, the better part of a day has vanished and I'm kicking myself. Pro-coders have terms to describe the kind of hole I've fallen into, like "yak-shaving" or "bike-shedding" (as in "who cares if it's red or green, it's a *?!$ing bikeshed"). I'm not sure which pertains to me. Probably both.

Eventually I retrench, remembering an email sent to me by Gerald Weinberg, author of a book called "The Psychology of Computer Programming", in which he suggests that a computer is "like a mirror of your mind that brightly reflects all your poorest thinking". My greatest personal weakness? A tendency to attempt too much and over-complicate things (I'm an "architecture astronaut" in coding parlance). I delete my code and start again. Later, Nicholas Tollervey tells me in consolation that my time wasn't wasted, because I was practising – the developer equivalent of character-building – and I am almost tempted to believe his kindness.

I've learnt a lesson and draw confidence from my newfound discipline. But, as if I didn't know by now, nothing should be taken for granted in the world of the naive coder and any satisfaction I feel at having stormed the input stage is rapidly dispelled when I discover that, after a promising first date, the Twitter API is no longer responding to my app. I test the first half of my code and it works, then

spend ten hours reading and researching, and watching endless YouTube how-to videos, in an effort to tempt it back, all to no avail. I examine the requests package used to relay search requests to the API, but can't find a problem there, so go to the Cheese Shop and import a promising alternative called TwitterSearch, which ties me in further knots. Now I'm getting endless "syntax error" messages that stop my code from doing anything at all. Hours later, at two in the morning, nerves stretched as if the entire staff of Facebook has thrown them out the window and shimmied down them to escape, I send an SOS to Tollervey, grateful, for the first time in my life, for the eight-hour time-zone lag between San Francisco, where I live, and the UK. To my unbounded relief, he answers straight away and arranges a screen share to help solve my problem. He looks for a moment, then laughs.

"You probably don't feel like it right now, but you're *so* close."

```
import twitter

api = twitter.Api(consumer_key="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
                  consumer_secret="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
                  access_token_key="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
                  access_token_secret="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx")

book_name = input('Please enter name of book here: ')
author_surname = input('Author surname: ')
search_term = book_name + ' ' + author_surname
print('Now searching for: "' + book_name.title() + '"!')

results = api.GetSearch(term=search_term)

print("\n\nThe results are:\n\n")

for tweet in results:
    print(tweet.text)
    print("       ~ {} ({})\n\n".format(tweet.user.screen_name, tweet.created_at))


```

A stray parenthesis had thrown the whole program into chaos. Tollervey removes it and the code works. I stare at the screen in disbelief. We're done. Too wired to sleep, I stay up talking to Tollervey about programming for another hour. My app is crude and unlikely to change the world or disrupt anything soon, but it feels amazing to have made it. More than anything, I'm astonished at how few lines it contains. With the Twitter API security keys redacted, it appears as above.

After all the caffeine, sweat and tears, were my efforts to learn to code worthwhile? A few hours on freeCodeCamp, familiarising myself with programming syntax and the basic concepts, cost nothing and brought me huge potential benefits. My beginner's foray has taught me more than I could have guessed, illuminating my own mind and introducing me to a new level of mental

discipline, not to mention a world of humility. The collaborative spirit at code culture's heart turns out to be inspiring and exemplary. When not staring at my screen in anguish, I even had fun and now thrill to look at a piece of code and know – or at least have some idea – what's going on. I fully intend to persist with Python.

More powerful than any of this is a feeling of enfranchisement that comes through beginning to comprehend the fascinating but profoundly alien principles by which software works. By accident more than design, coders now comprise a Fifth Estate and as 21st-century citizens we need to be able to interrogate them as deeply as we interrogate politicians, marketers, the players of Wall Street and the media. Wittgenstein wrote that "the limits of my language mean the limits of my world." My world just got a little bigger.