[MUSIC PLAYING] DAVID MALAN: All right. This is CS50, Harvard University's Introduction to the Intellectual Enterprises of Computer Science and the Art of Programming. My name is David Malan, and I actually took this class myself some years ago, but I almost didn't. It was my first year in college and my roommates and I were living in Matthews Hall, for those familiar.

[CHEERING]

Nice, Matthews. Our claim to fame, actually, at the time was that our room was literally Matt Damon's just three years' prior. So onward from that. But my first year, I didn't really have the nerves to set foot in this classroom, let alone computer science. In fact, for me, computer science, and CS50 in particular, was very much this class to beware. Like, I was kind of comfortable with computers, but I certainly wasn't among those more comfortable with computers.

And so I shied away my first year. Instead, I actually took a whole lot of classes in government. In fact, in high school, I was really enjoying history. I loved this constitutional law class that I took my senior year of high school. And so I figured, OK, if that's what I liked in high school and if that's where my comfort zone was, that's probably what I should be doing in college. And so I essentially declared as my concentration or major government for the first year, year and a half of school.

But my sophomore year when I was living actually in Mather House instead. OK, no one from Mather. In Mather House instead, it was-- I sort of followed some friends, I think the first week of class in September of that year to a class called CS50. And honestly, once I stepped foot in that classroom, the professor at the time was a famous computer scientist by the name of Brian Kernighan, now at Princeton.

Like, I was hooked. And literally would I go back thereafter on Friday evenings when, at the time, problem sets were released and sit down at 7:00, 8:00 PM on Friday nights and dive into homework. Which isn't necessarily something we recommend, per se. But for me, it was like this sign that, wow, I've sort of found my calling. I found my classmates here on campus. And that's not going to be the case for everyone. Certainly, we have no expectations that after taking one computer science class that you will or might want to take others.

But what's so empowering about computer science and CS50 in particular is it's so applicable to the broader world, whether you're in the arts, humanities, social sciences, natural sciences, or beyond, it's just so applicable the concepts and the practical programming skills with which you will exit a class like this. Now it's going to be challenging along the way, and indeed, I put in my time back in the day, and even I did find it challenging.

And here, for instance, is a photograph of a very famous MIT hack, so to speak, from down the road, whereby it communicates that getting an education from MIT is like trying to drink from a fire hose. Which is to say there's going to be so much information, like so much new stuff that certainly on any given day of the week, you're not going to be able to absorb all of it that first time around. But at the end of the day, it's through that challenge, putting the time in, that the returns are therefore just so much higher at the end of the course.

And indeed, will you walk out of the course with a much better understanding, not only of computer science and programming, but ultimately how to teach yourself new technologies and beyond. For the next three plus months, will we have teaching fellows, teaching assistants, course assistants, and myself by your side guiding you through the course's material. But the goal by term's end is to take those and leave those training wheels off so that you're well-equipped to go teach yourself new ideas beyond the class itself.

Take comfort, though, in knowing that most CS50 students have never taken a CS course before, and indeed, as per the syllabus, what ultimately matters in this

course is not so much where you end up relative to your classmates but where you end up relative to yourself when you began. And when you began is thus today. And so consider just how comfortable or uncomfortable you are with computing, let alone computer science and programming, particularly if you've never explored either in a classroom before, and consider the delta just a few months from now that will really describe how far you have come.

And that is all that matters, not how much the student to the left or the right, in front or behind you right now, knows. With that said, let me add some additional inspiration, if I may. Here's a photograph of my own very first homework assignment in CS50 from 1996, and I will draw your attention to the fact that even though this is a so-called hello world program that we'll play with ourselves next week, it is pretty much literally the shortest, easiest program you can write in a programming language called C.

I still somehow got minus 2 on my very first homework assignment, which is to say, we're all going to make mistakes along the way. But the goal will be to learn and enjoy that process here on out. At the end of the day, too, like me, you'll exit with your own proudly held high took CS50 t-shirt as is our tradition too. With that said, there are so many other traditions within CS50, both on campus and off. And in particular, do we try in CS50 to provide not only the academic support structure that you might want going through the class, but also a collective shared community experience.

Which is to say in just a few days we'll start off the term formally with CS50 Puzzle Day, which is not only an opportunity to get together with friends, with pizza and prizes and also logic puzzles of sorts, but really to send the message that computer science itself is not about programming, it's not about C, it's not about Python, it's not about programming languages per se, but about problem solving, ever more so collaboratively with other smart people by your side in this class or beyond.

And indeed, are there, toward the end of the semester, reinforcements of the same by way of a little something that we call the CS50 Hackathon, which will be an opportunity overnight to dive into your own final projects, the capstone of this course, thereafter followed by the CS50 fair, which will be a campus wide exhibition for students, faculty, and staff across campus of your very own final projects, be it your very own web app or mobile app or anything else you decide to create by term's end.

And indeed, the objective at the end of the day, truly with that final project in particular, is going to be to create for yourselves, for your classmates, for attendees, something that we didn't even teach you how to do. And indeed, that will signal ultimately that you're indeed on your way and ready. Toward that end, thought we would give you a sense of CS50's past by way of this short video, if we could dim the lights, that paints a picture of all that awaits here and beyond.

[VIDEO PLAYBACK]

[MUSIC PLAYING]

[END PLAYBACK]

DAVID MALAN: All right, so welcome aboard to CS50 and computer science itself. So what is computer science? Well, put simply, it's the study of information. Like, how do you represent it and how do you process it. But more fundamentally, what we'll teach you in this class is computational thinking. That is to say, the application of ideas from computer science to problems of interest to us within the class and problems of interest to you after the class.

And so at the end of the day, what computer science really is is about problem solving, ergo that sort of global applicability. And by problem solving, we mean something quite simple. In fact, we can distill it as follows with this mental

image. This is problem solving. You've got some problem to solve, thus known as the input that you want to solve. And the goal, of course, to problem solving is to actually produce a solution. So the output in this model would be the solution.

The interesting part ultimately is going to be in how you process that input and turn it into that output, ergo solving problems. But before we can do that, we all kind of have to agree how to represent sent these inputs and outputs, especially if we want to do it in a standardized global way using literally computers, be them laptops, desktops, phones, or most any other kind of electronic device nowadays. So how can we do that? Well, there's different ways to represent information in the world.

For instance, if I were to take attendance old school style, maybe in a smaller room, I might do 1, 2, 3, 4, 5, 6, 7, and so forth and just count people on my human hands. That's actually known as unary notation, otherwise mathematically known as base one, because you're using your fingers literally in this model as digits. But a little quick question.

How high can you count with one human hand? Five is incorrect if you use a different base system than one. So it's obviously correct if you're just using unary and just counting 1, 2, 3, 4, 5. But I dare say I can come up with more patterns in my human hand alone that would enable me, without a second hand or a couple of feet, to count higher than five. In fact, maybe for those more comfortable, how high could you actually count on a single human hand, perhaps?

So 31, believe it or not, is, in fact, the correct answer. But why? Well, here I initially started pretty naively. 1, 2, 3, 4, 5. And I just combined all of my fingers and counted the collective total. But what if I'm a little more clever and take into account the pattern of fingers that go up. So maybe this is still zero. This is one. But now maybe we just agree universally that this is two. Even though it's just my single pointer finger.

Maybe we all just agree that this is three with two fingers up. Maybe we agree that this is often offensive with just one middle finger up, but this would then be four. This could then be five. This could then be six. This could be seven. And if I keep permuting my fingers in this way-- allow me to spoil it-- this would be, in fact, 31. But again, why? But the difference here is that we're no longer using unary or base one as a mathematician would say, but rather base two.

Because if we take into account not just the total number of fingers that I'm using, but whether each finger is down or up being therefore in two potential states. Down, up, A, B, black, white, however you want to distinguish those two states of the world, you're now operating what's called base two, and perhaps more familiarly, even if you're not a computer person per se, this is the so-called binary system. And odds are, even if you're not a computer science person at all, you probably generally know that computers only understand or speak what alphabet, so to speak?

So ones and zeros, zeros and ones, otherwise known as the binary system. And in fact, there's a term of art here that's worth noting. When you're using zeros and ones, which, of course, are a total of two digits, you have binary digits, so to speak-- bi implying two, which means there's two possibilities, zero or one. If we actually get rid of some of these letters and then join these two phrases, here you have a technical term that is a bit. A bit is just a binary digit, which is to say it's a zero or one.

And this is in contrast, of course, with the system you and I know as the decimal system. Dec implying 10, because in the real world you and I daily use zero through nine, which is 10 possibilities. Computers only use zero and one, that is to say two bits, to represent information instead. So how do we represent this information, especially when at the end of the day what we're using are indeed computers and electronic devices? Well, if I want to represent

zero, I can actually think of this as analogous to the physical world.

Maybe I have a light bulb that's off or on controlled by a switch that turns it off or on. So you can think of a binary digit that is a zero as really just being a light bulb that is off. By contrast, if you think of a one in the digital world as, of course, being the second of two possibilities, you can think of that in the human or analog world, the physical world, as being a light bulb that is on. And in fact, what's inside of your Mac, your PC, your Android phone, your iPhone are millions of tiny little light switches known as transistors that just can be turned on or off, on or off.

And essentially, you can use those transistors to store information because if you want to store a zero, you turn one of those switches off. If you want to store a one, you turn one of those switches on. Of course, that sort of invites the question, well, how do we count higher than zero or one? Well, we would seem to need to use more than just maybe a single bit, a single light bulb. So if we wanted to count higher than, for instance, zero or one, why don't we go ahead and maybe do this?

So just so I have some place to put these, let me borrow some of our actual physical light bulbs here from the stage. And let me propose that now, with three bits on the stage, three light switches, three transistors, whatever metaphor you're most comfortable with, this is how a computer would represent a zero, because all of them are off. So it's off, off, off.

But if a computer wanted to count to one, we could do naively this. We could turn this on. If the computer wanted to turn represent two, we could do this. And if a computer wanted to represent three, we could do this. But I'm kind of painting myself into a corner and not using these light bulbs as cleverly as I could, because at the moment I've only counted as high as three. So if I want to count to four, to five, to six, I'm going to need more and more light bulbs. Can we be a little more clever?

Well, again, someone else who's among those more comfortable, what's the spoiler here? How high using binary zeros and ones could I count with three light bulbs total? In back? Yeah. So seven here is the answer. And if that, too, you're sort of wondering, how are people figuring out 31 and 7? That's the goal at hand here. So let me do this. Let me turn all of these off again so that my three light bulbs or switches again represent zero.

And the first one's easy. This is how a computer would represent the number one. It would be on, off, off. How, though, is a computer going to represent two? Well, just like my proposed finger example. Let's do this. Let's turn this one off and this one on. That is how a computer would represent two. By saying off, on, off. In other words, 010 would be the way to pronounce it digitally. What if I instead want to represent three? That's how on my finger I did this, with two fingers.

Well, I'm going to turn this one on. This is three. Now, this will, for those less comfortable, be non-obvious. This now is how I would represent the number four. This is how I would represent five. This is how I would represent six. And this, as per the spoiler, is how I would represent seven. So perhaps very non-obvious what it was I just did or why I chose those patterns. But I dare say if you rewind in your mind's eye or literally later on video, you'll find that I actually did show you eight distinct patterns of light bulbs.

The first one was off, off, off. The last one was, on, on, on. And there were another six total in between then. Well, wait, why seven? Well, if you start counting at zero and I claim there's eight possibilities, you can only count from zero to seven, as we will soon see. So how are these patterns coming about and what is it that our computers are actually doing? Well, it's actually doing something a little like this, quite like in decimal.

So in the human world, you and I are very much in the habit of using base 10,

zero through nine, a.k.a. Decimal. Well, how do we use it instinctively as humans? Well, what's this number obviously on the screen? 123. But why is it 123? Like, for years you haven't really thought about why this pattern of symbols or digits on the screen, one, two, three, represents mathematically this number that you know obviously as 123.

But if you rewind to grade school, odds are, like me, you were taught that the rightmost digit is in the ones column, this second digit is in the tens column, this digit is in the hundreds column, and so forth. So even though none of us have to do this math explicitly, what you're instantly doing is 100 times 1 plus 10 times 2 plus 1 times 3, which gives you 100 plus 20 plus 3. Oh, that's why it is 123, because these digits in this order have that significance. The digits to the left have more weight, so to speak, than the digits to the right.

So what can we take away from this? Well, let's generalize it first as just any three digit number. So number, number, number. The ones column, the tens column, the hundreds column. But there's some math going on there, and it's not particularly sophisticated. Those are actually powers of 10. So 10 to the 0, 10 to the 1, 10 to the 2, and there's your decimal system. Because the base in this value is a 10, that's because there's 10 possibilities for each of those placeholders, zero through nine.

But in the binary world, in the world of computers where all they have are zeros and ones, why? Because all they have physically is transistors. Tiny, tiny, tiny light bulbs that can be off or on. If you only have two digits to play with, the 10 base should, of course, become a two base. And now if we do some math here, 2 to the 0, 2 to the 1, and 2 to the 2, you get the ones column, the twos column, the fours column. And if we keep going 8, 16, 32, 64, 128 and so forth, its powers of 2 instead of powers of 10.

But this is to say computers represent information in exactly the same way you and I have since childhood, but they have fewer digits at their disposal, so these columns need to be weighted differently. So we can still count from zero all the way up toward infinity. So what does this mean? Well, here we have three bits on the screen, 000. If we were to convert this now mentally or on paper pencil to decimal, how do we do it? Well, 4 times 0 plus 2 times 0 plus 1 times 0.

That gives us the mathematical number you and I know as zero. That was three light bulbs. Off, off, off. Well, what if we turn on one light bulb all the way on the right? What decimal number does this binary number, 001 represent? Just one, because it's 4 times 0, 2 times 0, 1 times 1. Here's where things got more interesting, even if non-obvious in light bulb form or even physical hand form. 010 in binary is what in decimal? Two, because it's 2 times 1 and that's it. 011 in binary is, of course now three.

This is now four. This is now five. This is now six and seven. On, on, on or 111 is the highest we can count with these three bits. All right. So how might a computer intuitively count as high as eight? What do you need to do, presumably? You're going to need to add a bit. So you need another light bulb, another switch. You need more memory, so to speak, to use nomenclature with which you're probably familiar.

So in fact, if we change all of those to zero but we give ourself one more bit for a total of four, that's got to be the eighth place because there's just another power of two. So 1000 is the decimal number eight. You don't say 1,000 in binary. You literally say 1000. But that is the number you and I know as eight. And you can keep going up and up and up. And how then computers with Excel or any kind of number crunching software count really high and keep track of really big numbers?

The computer just throws more and more transistors at it, more and more bits to count higher and higher and higher than this. It turns out, though, one bit, three bits, even four bits aren't that useful in practice because literally you

can count as high as seven or maybe 15 or 31. So more commonly, as is commonly known, is to use a byte of bits instead. How many bits is in a byte, for those familiar? So it's just eight. Why eight? It's just more useful than one or two or three or some other number.

And as an aside, it happens to be a power of two, which is just useful electronically as well. So a byte then is just 8 bits. And here are those columns I rattled off off the top of my head. Here is how a computer would represent zero in decimal, but using eight binary digits or bits. Little trivia. And again, this is not what computer science is about, but it helps to know the lower bounds and the upper bounds of these kinds of values. How high can you count with 8 bits or 1 byte if this is zero?

Yeah. So it's actually 255. So if I were to change all of these zeros to ones and then do some quick mental or calculator math, 128 plus 64 plus 32, 16, 8, 4, 2, and 1 would actually give me 255 total. Plus 0, which gives me 256 total possibilities. So this is only to say-- this is not, again, the kind of math will frequently do, but you'll commonly see in the computer world and programming world powers of two, numbers like 255, 256.

Why? Because these are the common units of measures that systems tend to use. So let me pause here and see, with respect to binary, zeros, and ones, transistors and the like, any questions or confusion we can clear up? Oh, really good question. Why are bits just on or off instead of maybe sort of 0%, 50%, 100% by playing with voltages? So the voltage inference of yours is actually correct. That's what computers typically do.

Maybe they use 0-ish volts to represent 0, maybe 5-ish volts to represent 1. It turns out it's just really easy to do extremes in computers. If you start to split that range of voltage levels, for those who remember any of their electricity, it just gets harder and harder to be exact. And if you get things a little too murky, you might mistake a zero for a one or a two or a three. So it turns out it's just simpler to use the binary system. But there do exist computers known as ternary computers that actually use three values, zero, one, and two, which is somewhere, of course, between binary and decimal.

But you can do different things. It's just simple on and off. In case in point, I don't want to really be dramatic and turn off my computer, but if I pulled out the power plug, that could be off, literally, a.k.a. zero. Plug it back in, that's a one. There's just a cleanliness and simplicity to that. Other questions or confusion that we can clear up? No?

OK. So if you're in agreement for the moment that, OK, using just zeros and ones, we can represent any number we want from zero on up, let me propose that we do more useful things with our computers and our pockets and desktops and laps like represent letters, for the sake of Google Docs, Microsoft Word, or any kind of text that we might want to write.

So knowing now that computers only contain or only use zeros and ones, and therefore only contain hardware like transistors, how could you represent something like a capital letter A in English inside of a computer? Which, of course, is not a number anymore. Like, what could we do? Yeah?

AUDIENCE: We could use the alphabet and then use numbers.

DAVID MALAN: OK, yeah. So we could take the alphabet A through Z in English and we could just assign each letter A number. And honestly, that is not only the correct answer, it's really the only answer. Because at the end of the day, if all you have are zeros and ones available to you, that is the entirety of the potential solution to this problem. So it turns out that, yes, capital letter A, some years ago, was decided by a bunch of people in a room shall be represented with this pattern of zeros and ones. 01000001.

And now, trained as you are to do a bit of quick binary math, what decimal

number is used to represent apparently capital A? So 65, because that's 64 plus 1 plus 1 times 1 is 65. What is B? Turns out it's 66. What is C? 67. So they kept things simple there on out. Might have been nice if A were zero or maybe a were one. But nope, we're stuck with 65 instead. But everything after that is very much predictable.

And in fact, for lowercase letters, there's a whole other set of numbers such as lowercase A happens to be 97, lowercase B happens to be 98, and so forth. But again, this is like CS trivia. But what's important here is that there are indeed contiguous from 65 to 66 to 67 on up. That's something we're going to be able to leverage beyond the letter A alone. What is this system? What is this mapping that you yourself propose? It's ASCII, the American Standard Code for Information Interchange.

And indeed, it was a bunch of Americans years ago who came up with this system. Unfortunately, at the time, they only allocated 7 and eventually 8 bits total for representing letters, both uppercase and lowercase, numbers on the keyboard as well, punctuation symbols as well. And so per our conversation a moment ago, if the Americans in this room, so to speak, only used 8 bits total, how many different characters can we represent with a computer in this story?

So only 255, technically 256 if we, again, start counting from zero. So that's not nearly enough to represent all human languages, but it is enough to represent at least English, among some others. So here, for instance, is a chart of the ASCII mapping. And sure enough, if we zoom in on this column here, 65 is capital A, 66 is capital B, dot, dot, dot 72 is H, 73 is I, and so forth. So there is a standardized mapping for at least all of these English letters.

Well, suppose you were to receive an email or a text message or like a Google Doc containing this pattern of zeros and ones. So 01001000 and so forth and so forth. So 3 bytes worth. Three sets of 8 bits. That is to say 3 bytes, each of which represents a single letter in ASCII. What message have you received? Well, I'll do the math this time so we don't have to. Suppose what you really received was decimal 72, 73, 33.

What message did you just receive? If you recall the previous chart. Hi was in fact correct. Why? Because H is 72, I is 73. And wait a minute, 33. So here's H. Here's I. 33, if we highlight it instead, happens to be an exclamation point. So that is literally what is going on underneath the hood, so to speak, when you get a text message today that literally says in all caps and an exclamation point, HI! Your phone has received at least three bytes, each of which represents a letter of the alphabet.

Your computer is quickly doing the mental math to figure out exactly what numbers those are and then looking up in the so-called ASCII chart in its memories, in some sense, what letter should you actually see on the screen there. And so if you were to then display that message, you would see it indeed in English as opposed to those numeric equivalents. How else might we use this then?

Well, here again is that chart. And maybe just to vary things, maybe take a little pressure off of me here, why don't we try spelling something else? This time a different three letter word, but maybe eight volunteers. Could we get a bytes' worth of volunteers? And I can sweeten the deal with some stress balls in exchange. You just have to be comfortable coming up on stage and being on the internet. So yes. One, two. How about three, four. How about five, six, seven. And how about eight. Come on up. A round of applause for our volunteers. Yep.

[APPLAUSE]

All right. So what I'm going to have each of you do is represent a bit in a particular order. So if you want to just, in any order, line yourselves up here facing the audience. Come on over. All right. And we will have you represent-- well, we got to see who ends up where. Scooch this way a little bit. This way,

this way.

All right. So you shall be the ones place and just hold that in front of you. Twos place. Threes. Fours place. Eights place. 16, 32, 64, and 128. And just compress yourselves a little bit if you could. So each of these folks represents a bit in a particular place. And let's say this. If you're just standing there uncomfortably without any hand raise, we'll assume that you're representing a zero, quite simply.

If your hand goes up, though, the audience should assume that you're representing a one. And therefore, what we'll do is spell out a three letter word, and on each round of this, you'll either stay, stay like this, or you'll raise your hand. But first, let's actually meet some of our volunteers here, starting with position number one, if you'd like to say your name, perhaps where you're from and/or studying.

AUDIENCE: Hi. My name is Brooke. I'm from Indiana, and I'm studying biology and computer science.

DAVID MALAN: Nice. Welcome.

AUDIENCE: Hi, I'm Becca. I'm from, like, Maryland, DC area, and I'm studying electrical engineering.

DAVID MALAN: Welcome.

AUDIENCE: Hi, I'm Addison. I'm from Maryland. I'm studying engineering.

AUDIENCE: Hi. I'm Sharon. I'm from Rwanda and I'm studying CS and math.

DAVID MALAN: Welcome.

AUDIENCE: Hi, I'm Grace. I'm from Alabama and I'm studying electrical engineering.

DAVID MALAN: Welcome.

AUDIENCE: Hi, I'm Angelina. I'm from Maryland. And also, I stay in Matthews.

DAVID MALAN: Nice. Matthews. Nice.

[APPLAUSE]

AUDIENCE: And I'm studying applied math and econ, as well as environmental science and public policy.

DAVID MALAN: Welcome.

AUDIENCE: I'm Owen Bells and I'm from rural Virginia and I'm studying CS.

DAVID MALAN: Nice, welcome. And?

AUDIENCE: My name is Max. I'm from London. I'm also staying in Matthews and I'm studying computer science and neuroscience. Thank you.

DAVID MALAN: Welcome aboard as well. If you're wondering, too, why I was wearing these glasses at the start-- so very common on the internet nowadays as these POV videos. So it turns out these Ray-Bans actually record video footage, and we have a couple of them, and we'd thought we would offer them to a couple of volunteers. If anyone wants to record their point of view for everyone here. And Vlad here is going to help make sure they're recording. Second volunteer. Yes, number two.

All right. So as Vlad gets those set, on the backs of your pieces of paper you

have instructions for the following three rounds. Each round represents a letter. The audience participation part of this is to actually do the mental math to figure out what number these volunteers are representing. So go ahead and execute round one, either keeping your hand down or raising it appropriately. OK. What number are our volunteers here representing?

AUDIENCE: 66.

DAVID MALAN: 66, because we have a 64 plus a 2. That then maps to what ASCII letter?

AUDIENCE: B.

DAVID MALAN: B was the first letter. OK, hands down. Round two, go. A little harder. What's now being represented?

AUDIENCE: 79.

DAVID MALAN: I'm starting to hear it. 79 is in fact correct. 79, because we have a 64 and an 8 and 4 and 2 and a 1. So if it's a 79, we have the ASCII letter O. So we've got BO, and then lastly, third round. Go. We have 01010111. What number is this?

AUDIENCE: 87.

DAVID MALAN: 87. Which spells the letter?

AUDIENCE: W.

DAVID MALAN: W. Which spells the word?

AUDIENCE: Bow.

DAVID MALAN: Not bow. Take a bow if you could. All right. A round of applause for our volunteers here.

[APPLAUSE]

And come on off this way and help yourself to a CS50 stress ball. Thank you to our volunteers. So this is only to say we've now agreed on how we can represent numbers from zero on up. We've agreed on how we can represent letters. But at least letters using ASCII, and in fact, these are more than just decoration. In fact, it's a little bit of trivia by lecture's end. If you to come up for your very own CS50 stress ball, turns out there are 64 light bulbs at the foot of the stage here.

If you break them down into 8 byte or single-- 8 bit or single byte chunks, there's an eight letter word that happens to be spelled out today using this here ASCII chart. So today's mystery is what exactly is that there word. But of course, if you have only 8 bits, you can only represent, like, 256 characters, which sounds like plenty for English, and indeed, it is. Zero through nine, A through B, capital and lowercase, uppercase and lowercase, as well as punctuation.

But there's so many other human languages in the world that have other characters. For instance, we have not just the English alphabet we might see here on a US English keyboard. We have accented characters, we have various Asian languages have even many more glyphs. We need more than 256 possible characters. And so nowadays computers do not just use 7 or even 8 bits. They might use 8 bits for some letters, like all of the English letters.

They might use 16 bits for certain other languages. Maybe even 24 or 32 bits. And fun fact, if you have 32 bits-- and we have more than that on the stage. If you've got 32 bits, you can actually represent as many as 4 billion possible

characters, which is quite a bit. No pun intended. So what else can we represent? Well, the goal of this system, not just ASCII, but something known as Unicode, which is a newer standard, is to be backwards compatible with ASCII.

So humans left all of those other numbers alone, 65, 66, 67 and so forth, but they added to it a super set of representations that maybe are 16, 24, or 32 bits. The goal being to be able digitally to represent all human languages, past, present, and future, and even through pictograms, things like smiley faces and the like, even people, places, things and emotions that transcend human language.

And in fact, odds are within the past few minutes or hours, most of you have used one or more of these here emoji, these pictograms, which it turns out are just characters on a keyboard. You might have to hit a special button to pull up that form of the keyboard, but these are just here characters. And so these emoji have exploded in popularity for a number of reasons, one of which is, my God, what are we going to do with 4 billion possible patterns of zeros and ones?

We can start to have some fun with it and represent things beyond English and human languages alone. Now, as an aside, when it comes to Unicode, it turns out Unicode, years ago, standardized this pattern of 32 zeros and ones to represent just one of those emoji. So emoji tend to use even more bits here. Anyone know what decimal number this is? This is not a fun mathematical exercise. The spoiler is 4,036,991,106 is the decimal number that actually represents, as of present, the most popular emoji in the world. Does anyone want to hazard a guess what emoji this here number represents?

AUDIENCE: Heart.

DAVID MALAN: Heart? Hearts? No, but it's actually this so-called face with tears of joy. So perhaps think about the frequency with which you send that one. And even though it's obviously a picture on the screen, sure, it actually is more like a font, because underneath the hood, it's indeed just a pattern of zeros and ones or a decimal number that the computer is storing. But the computer, be it Mac OS or Windows or iOS or Android, know to display that pattern as this here picture.

But the pictures might look different depending on the hardware. Why? Because there's companies like Google and Microsoft and Meta and others that have their own artists on staff as employees, and those artists interpret the descriptions like face with tears of joy differently. So those of you with an Android phone actually see face with tears of joy looking a little something like this.

And if you have Telegram, for instance, installed on your phone, it's even more animated than that. It's this here emoji using the same pattern of zeros and ones. So different artists render these here emoji in different ways, but all they are here are patterns. Now, for all of the other answers, save one that was shouted out a moment ago, this is a sort of cloud diagram of the most popular emoji as of a couple of years ago per Unicode, whereby the size of the emoji indicates its relative popularity. So heart, I did here over here, is indeed one of the most popular ones as well. Question?

AUDIENCE: Why do certain emojis show up [INAUDIBLE]?

DAVID MALAN: Oh, really good question. Why do certain emoji not show up on one device or another? It depends on how recent the software is. Pretty much every year the humans behind the Unicode consortium release new emoji. Which is to say they decide that this other pattern will represent this new emoji, this other pattern will represent this new emoji.

And unless you update your phone, your laptop, your desktop to the very latest software and the manufacturer of that device or software also updates by hiring an artist to draw those pictures in their own fonts, in their own style, you're

going to see usually just an empty black square or maybe just a black and white heart instead of something more colorful. Really just placeholders, because, it's as though you don't have the right font installed or really, you have an older version of that same font installed. But it's become sort of an annual tradition that new and more emoji are released every year, which is among the reasons why these updates contain more and more. Yeah?

AUDIENCE: How do you represent color in bytes?

DAVID MALAN: That is an amazing segue. How do you represent color in bytes? Well, you use RGB, which happens to be, by coincidence, the next slide. So let's again, recap. We know how to represent letters. We know how to represent numbers. We can even represent emoji. But those emoji technically on the screen are, of course, composed of colors, like a whole bunch of yellow for that there smiley face? How do computers, then, using only zeros and ones, represent colors?

Well, by convention, they typically use a system that, by an acronym, is called RGB. Red, green, blue. And this is to say that a computer, to represent a single dot on the screen-- maybe this one, this one, this one-- will allocate some number of bits or some number of bytes to represent the color of just that, their dot, otherwise known as a pixel. You can actually see pixels on your phone or even on your TV or monitor. If you go really close, especially if it's an older monitor, you can see the tiny little squares.

Each of those has some number of bits telling the device what color to use. In particular, these devices typically use three numbers in total, three bytes. So that is to say 24 bits per pixel. And they do this. If you were to represent a single dot on the screen using these three numbers, just by intent here, this is 72, 73, 33, which in the context of a text message, an email, a Google Doc represents, of course, hi, textually.

What if a computer uses that same pattern of zeros and ones, that is the same pattern of decimal digits, to represent the color on a screen? Which is germane if you're opening an image using Photoshop. So using a different piece of software that knows about colors and images and not just text. Well, this would imply that you want that dot on the screen to have a medium amount of red, a medium amount of green, and a little bit of blue. Why do I say medium and little?

Well, again, if each of these numbers is using 8 bits or 1 byte, the highest we can count, as we discovered, was 255. So I'm kind of averaging here. So 72 at a 255 feels to me like a medium amount of red. 33 feels relatively little blue. But if now the computer combines those wavelengths of light, so to speak, a medium amount of red, medium amount of green, a little bit of blue, what you get is the color code for a single dot. And does anyone want to guess what color roughly this represents, these three bytes?

AUDIENCE: White.

AUDIENCE: Purple.

DAVID MALAN: Not white, not purple.

AUDIENCE: Brown.

DAVID MALAN: Not brown.

AUDIENCE: Yellow.

DAVID MALAN: Yellow, in fact, is the answer. So it represents in a single pixel roughly this shade here of yellow. Which is to say, if we look back at any of those emoji, which, again, are represented by patterns of zeros and ones, but you and I as humans perceive them as images on the screen-- let me actually go

ahead and zoom and zoom in further to one such sample emoji. And when you zoom in far enough or you put the phone close enough to your face, you can actually see all of those little dots known as pixels, all of the little squares.

And given that so many of these pixels are yellow, that is to say that that pattern of three bytes, 72, 73, 33, is used to represent this pixel. Another 3 identical bytes are used to represent this pixel, this one, this one, and so forth. So now if you've taken digital photos on your phone or a camera, you're probably generally familiar from the internet and hardware today that a photograph is, what, 1 megabyte, 10 megabytes depending on the resolution of it?

Well, megabyte means millions of bytes. Where are all of these bytes inside of these photographs or these images you're taking or downloading? They correspond to every one of the single pixels on the screen. There's at least three bytes being used to represent every one of those dots. As an aside, bit of a white lie because nowadays there's fancy compression software that can use fewer than that many bytes. But in general, that's where all of those bytes, those millions of bytes are coming from.

So how is that for an answer to how do we represent colors? Thank you. So if we agreed now that there's this way and perhaps others to represent colors, well, how do we represent not just images, but videos? Well, videos once upon a time, or movies, were called motion pictures. So motion pictures with motion. Why is that? Well, it's analogous to growing up. If you ever played with one of these picture books-- and in fact, there's memes nowadays that have made these popular again, whereby why you have a whole bunch of images on individual sheets of paper.

And if you flip through them fast enough, your human mind and eyes perceive it as actual motion, even though you're just seeing image, image, image, image, image, image. But it's so fast, it looks like motion. That's all a video is on your screen. That's all a film is on your TV. It is not in fact, continuous motion. It's maybe 30 frames or images per second, maybe 24 frames or images per second. Which is to say, we know how to represent numbers, we know how to represent letters, we know how to represent colors and thus images.

Now we kind of get videos for free because it's just more of the same. Use more and more of those patterns. Why are videos so darn large? Why are they gigabytes to download, billions of bytes? Because there's so many darn images. 30 some odd images per second in those kinds of videos. And maybe lastly, just to top off our multimedia, how could you represent sound? Maybe musicians in the room. How, using only zeros and ones, could you represent something as sonorous as music? Something analog as digital. Yeah?

AUDIENCE: So each number corresponds to a frequency.

DAVID MALAN: Yeah. So each number that we store in the computer could correspond to a certain frequency, which has a direct relationship to the sound or the pitch of a note. For instance, in the world of piano and many other instruments, you've got like your A, your B, your C, maybe you have sharps and flats as well. We could just agree, like the ASCII people did years ago, to represent the musical note A, let's use this pattern, musical note A-sharp, let's use this pattern and so forth.

But maybe pitch alone or frequency alone is not enough. Maybe we need that number, but maybe a second number for the volume, the sort of digital equivalent of how hard are you hitting the key on the piano. Maybe a third number for how long are you holding the key down. So maybe the pitch and the volume and the duration, kind of like RGB, we could use three bytes to represent every musical note in some piece.

And if we wanted to keep track of what instrument should be played by the computer to sound that music, well, maybe that's just a fourth byte or something

else as well. Which is to say, at the end of the day, all we have are these zeros and ones to throw at the problem. So for now, that's it for representing information. We've got our numbers, we've got our letters, we've got our colors and images, our videos, and now sound. Any questions on how computers, then, are representing, as promised, those inputs and outputs using just zeros and ones? Yeah, in the middle.

AUDIENCE: The computer is taking it as input.

DAVID MALAN: Correct. So the computer is taking as input at the end of the day, zeros and ones and is outputting zeros and ones. However, as we'll learn in this class, by writing software, by writing code that understands those zeros and ones we will enjoy not just literally seeing zeros and ones, we will see A, B, C, we will see colors, we will see video, we will hear sounds so long as we write the code to interpret those zeros and ones.

And indeed, it's worth noting now that same pattern I keep using for an example, 72, 73, 33, how does a computer know? Is that the message hi? Is that the color yellow? Is it a dot in a video alone? Just depends on the context. Simply put, if you're opening that pattern of zeros and ones with Excel or a calculator program, odds are the software will interpret those three bytes as numbers, of course. If, though, you open that same pattern in a text messaging program, Google Docs, Microsoft Word, that same pattern will be interpreted as a sequence of letters.

Instead, if you open Photoshop, that same pattern, you'll probably see a single dot that happens to be yellow. Conversely, once you yourself are a programmer or even better programmer, you will be able to write in code how you want the computer to treat these patterns of zeros and ones. You can essentially tell the computer, use this to store a number or a letter or a color or something else. That's the power the programmer themselves have at the end of the day. Other questions on representing things with bits? No?

All right. So lastly, then, in this middle of this black box, so to speak, is the sort of secret sauce that solves problems, that converts those inputs to outputs, those problems to solutions. So what is an algorithm? It's really just step by step instructions for solving some problem. And indeed, I think back to my own first time in CS50 where we learned the same from Professor Brian Kernighan. And as luck would have it, just had my 25th reunion where we pulled some video footage from 1996.

And so we're actually fortunate to have the very first few minutes of CS50 over 25 years ago when I myself took it. But the lessons back then, as today, are fundamentally the same. And what's important, indeed, is to not only express yourself correctly, but precisely, as we'll explore today. This then is Professor Brian Kernighan, who, years ago, very memorably introduced us and my classmates to algorithms by actually, in class, shaving his beard. If we could dim the lights here for Brian.

[VIDEO PLAYBACK]

- The other thing that we're going to talk about in this class is the notion of an algorithm. An algorithm is a very precise description of how to do something. And the operative word there is precise. It has to be very, very, very, very precise. And the task that I'm going to do is that I'm going to trim my beard, which has gotten out of whack.

[APPLAUSE]

And I brought a variety of things which one might use to trim beards with.

[LAUGHTER]

[APPLAUSE]

[END PLAYBACK]

DAVID MALAN: So suffice it to say, I don't have much of a beard. But I do have this here other technology known once upon a time as a phone book. And these phone books, of course, have lots of information in them. Happen to be storing numbers and letters in particular. For those unfamiliar, they are storing human's names from A to Z here in English and associated with everyone's name is a number. So even if you've never had occasion to physically use this kind of device, turns out it's pretty much equivalent to the contacts or the address book app on your iOS phone or your Android phone as well.

Why? Because if you pull up your contacts, of course, you see some familiar names here alphabetized by first name or last name. And if you click on any of those names, you reach the person you're presumably trying to call or text. Pictured here then is John Harvard's, whose number here is plus 1-949-468-2750, which you're welcome to call or text at your leisure. But here is John Harvard that's partway through the phone book digitally.

Well, it turns out that physically in the phone book, we might use an algorithm, step by step instructions, for finding John Harvard in pretty much the same way as iOS, Android, Mac OS, Windows, or other operating systems themselves use. So I could, when looking for John Harvard, first name, starting with J, I could start at the beginning of the phone book and start looking page by page by page for John Harvard. And if he's there, I can call. This is an algorithm.

It's indeed step by step, but that was a bug. A few pages turned. But is this algorithm correct? Step by step, assuming I'm paying attention. So yes, if John Harvard is in here, I will eventually find him once I get to the J section. Now, this is a little tedious. It's going to take a while. A few dozen, a few hundred pages. So maybe I could do things a little smarter from grade school, like 2, 4, 6, 8, 10, 12, 14, 16, and so forth, going twice as fast. Is that algorithm correct? So no, but why?

AUDIENCE: You could miss it.

DAVID MALAN: I could miss him, right? I could just get unlucky, really, with 50/50 probability, because John Harvard could be sandwiched between two pages. Now, this isn't a complete loss, this algorithm. Maybe what I could do is if I get past the J section to K, I could double back at least one page just to make sure I didn't miss John Harvard. So I can still go twice as fast plus an extra step just to make sure I didn't mess up. So the first algorithm might take as many pages as there are in the phone book.

So if this phone book has a thousand pages, in the worst case, if I'm not looking for John Harvard, but someone whose name starts with Z, might take me a thousand pages to actually get there. Second algorithm, twice as fast. Literally, it might take me 500 plus one step to get there because I'm going two at a time, so as long as I indeed fix that bug. But what we used to do back in the day and what your phone is doing today, albeit digitally, is going roughly to the middle of the phone book, looking down and realizing, oh, I'm accidentally in the M section, so halfway through the phone book.

But what do I now know about John Harvard? Is he to the left or to the right? So he's obviously to the left, because J comes before M. So what I can do literally and what your computer does figuratively is tear the problem in half, throw half of the problem away, leaving us now with the same fundamental problem, but it's half as big. So I've gone from a thousand pages suddenly to 500 pages. And compare this to the other two, 1,000 pages, 999, 998, versus 1,000 pages, 998, 996, 994.

That's still slow. I went from 1,000 to 500 in just one step of this algorithm. What do I do next? I go roughly to the middle here. Oh, I went a little too far. I'm in the E section now. So is John Harvard to the left or right now? So he's

to the right. So I can again tear the problem in half, throw the left half away, knowing now that John Harvard must alphabetically be in here.

And I can divide and divide and divide and conquer this problem again and again by using that heuristic of going left or going right. And I dare say, if I do this correctly, I'll eventually be left with one single page on which John Harvard's number actually is. Or maybe he's not in the phone book at all. So how many steps maximally might that third and final algorithm take? It's not a thousand. It's not even 500 or 501. How many times can you divide a thousand pages in half again and again and again, roughly?

AUDIENCE: I want to say nine.

DAVID MALAN: So 9, 10. So typically 10 times, give or take. There's a bit of rounding there because it's not a perfect power of two, but roughly 10 times. Like, that is fundamentally better than both of the two algorithms because I go from a thousand pages to 500 to 250 to 125 and so forth, literally halving the problem again and again. So we can actually appreciate and see this even more so graphically.

And this is among the things we'll do later in the term when we speak to not only writing correct code, but is your code well designed? Is it better than your previous code? Is it better than someone else's code? Is it better than some other product? If you have given more thought to the algorithms and the quality thereof, you can perhaps minimize the time required to solve problems but no less correctly.

So if we have a simple xy plot here, on the y-axis or vertical is the amount of time to solve in whatever unit of measure, seconds, pages, however you want to count. On the horizontal or x-axis is the size of the problem measured in, for instance, numbers of pages. So this would mean zero pages in the phone book. This would mean a lot of pages in the phone book. This would mean no time to solve. This would mean a lot of time to solve. What's the relationship then, among those three algorithms?

Well, the first one is essentially a straight line, a slope of one. And if the phone book has n pages in it, we'll describe the slope here as essentially 1 over 1 for the algorithm with the first algorithm, turning page by page by page. Which is to say, if we were to add one more page to the phone book next year, first algorithm is going to take one more step. But the second algorithm is definitely better. It's definitely faster, but it's still a straight line.

So it's going to take roughly n over 2 steps on average, because you only have to go through half of the phone book because you're going two pages at a time, instead of the whole phone book in the worst case, if someone's name is Z, to go through every page in total. So if we actually compare these-- let me just draw some dashed lines. Suppose that you have this many pages in the phone book.

If you just draw this vertical white line here, it's going to take this much time in red using the first algorithm, but it's going to literally take half as much time in yellow for the second algorithm because you're literally going two pages at once. But the third and final algorithm is a fundamentally different shape. It instead looks a little something like this. It looks like it's flatter and flatter and flatter. It's always increasing. It never gets perfectly flat. But it grows so much more slowly as a function of phone book size.

And for those who recall their logarithms, this would be described as log base 2 of n. And in fact, that's where the math came from. Log base 2 of 1,000 is roughly 10 in total, even if you need a calculator to confirm as much. But this shape is fundamentally different. Why? Well, suppose that Cambridge, where we are, and Allston, the town across the street next year, combine their two phone books. And they go from a thousand pages each to one phone book with 2,000 pages.

The first algorithm is going to literally take twice as many steps or pages. Second algorithm is going to take half as many or 50% more because you're going two at a time. But the third algorithm is going to barely miss a beat. Why? Because if this is a thousand pages here and 2,000 pages is over there, just inferring from the shape of the green line, it's not going to be much higher on the vertical axis than the other two were.

So more specifically, if you have a 2,000 page phone book next year, how many more steps will it take you using that third and final algorithm? Just one, because you'll divide and conquer a 2,000 page phone book into a 1,000 page phone book, and then you're back at the original story. And that's the sort of power of learning algorithms. That's the power of learning computer science and learning how to program is to be able to navigate big data, so to speak.

Things the size of google, things the size of artificial intelligence training data sets using better and better, more clever algorithms that perform faster, and therefore not only make the software more competitive, but also make it more usable and more favorable for people like you and me when using that software. So when it comes to implementing algorithms as programmers, as computer scientists, what you're really doing is taking these algorithms, which might be expressed in English conceptually as we just did, but really just translating them to code, be it C or C++ or Python or R or Ruby or any number of other languages that exist in the world.

But for now, let's consider how we might implement that algorithm using something that's literally still English, but pseudocode. Something that is still correct, but precise and finite, as per Professor Kernighan's advice, which is to say use your own vernacular of English and just say what you mean but very succinctly. There's no one way to write pseudocode. It's not some formal language. I'm just going to convert the steps I did intuitively to step by step instructions as follows.

Step one, what I did was pretty much pick up the phone book. Step two, what I did was pretty much open to middle of phone book for the third algorithm. Step three, look at page. Step four, if person is on page, then, step five, call person. If that does not prove to be the case, step six, else if the person is earlier in the book, then open to the middle of the left half of the book and then go back to line three.

Then, else if the person is later in the book, open to the middle of the right half of the book and, again, go to line three. Else, there's a fourth and final case. If the person like John Harvard is not on the page, is not earlier, is not later, what's the fourth scenario we'd best consider? He's just not there. Else we should do something specific like quit.

Now, as an aside, everyone in this room has probably had one of these stupid technical support issues where your phone or your laptop or your desktop computer just freeze all of a sudden, or maybe it spontaneously reboots for no reason. Odds are that's because not you but some other human made a mistake. They probably wrote code working at Microsoft or Apple or Google or somewhere else, and they didn't actually anticipate that, oh, there could be a fourth scenario that could happen in the real world.

But if there's no code that tells the computer what to do in that fourth and final scenario, who knows what the computer is going to do? It might, by default, reboot. It might, by default, freeze. That's just a hint of the bugs, the mistakes in software to come.

But even though this is just one way to write this code, a.k.a. pseudocode, there are some salient characteristics that we'll use throughout today. One, there are these verbs, these actions. And henceforth, as aspiring computer scientists or programmers, we're going to start to call these by what a more and more technical audience would. These are functions. A function is an action or a verb. It's like a bite-sized task that a computer can do for you. Those then are

functions in this here pseudocode.

But there's other types of code in here. There are these things here. If else if else if else. Those are examples of what we're going to start calling conditionals. These are sort of proverbial forks in the road where maybe you go this way, maybe you go this way, but you decide which way to go based on a question. The questions that you ask are what we'll technically call Boolean expressions named after mathematician Boole.

A Boolean expression is a question with a yes or no answer, a true or false answer, a black or white answer, a one or zero answer. There's two possibilities, and there's a hint of the binary underneath. A Boolean expression is going to tell you yes or no, you should go down that fork in the road. Notice what's important here is that indentation mattered as a result.

Notice that on line four when I first asked if the person is on page question mark, so to speak, I should only do line five per its indentation if the answer is yes or true, I should only open to the middle of the left half of the book and go back to line three if person is instead earlier in the book. So indentation in pseudocode and in many programming languages has logical significance. It tells you whether to do things or not. But there's another construct in here.

Go back to. Go back to, which literally makes me go back to line three, potentially again and again and again, creating some kind of cycle or what we'll typically call a loop instead. So even in this relatively simple real world algorithm, we have these four fundamental characteristics of most computer programs that we will write in this class, and you might write beyond this class, that we have some technical jargon now to describe them.

But what's important to note is that line 8 and line 11, even though they're saying go back to line three, go back to line three, you might think you're running the risk of what we'll call an infinite loop where you literally get stuck in a loop forever, which doesn't sound like a good thing if, at some point, you want to turn your computer off, even though it's still working. But these will not induce infinite loops. Why? What is happening in this particular algorithm every time we go back to line three that guarantees eventually we will stop going back to line three?

AUDIENCE: The person is on the page, you call it.

DAVID MALAN: Exactly. If the person is on the page, we will call them or we will quit. But more importantly, because we keep dividing and conquering the problem, in this case, having the phone book, having the phone book, eventually we're going to run out of phone book, in which case, indeed, John Harvard is either on that page or not And we will call or we will quit instead. So we'll see in time.

And in fact, allow me to promise. Odds are at some point you will write code that seems to take control over the computer for you, where it's doing something, doing something, doing something, and it literally won't respond to you anymore. That's just going to be because of a mistake, a so-called bug that you yourself will invariably have added to your code accidentally. But we'll show you ways for terminating it or breaking out of those conditions.

And indeed, what we'll do in just a little bit after a break for today's lecture is explore not just these concepts, but some of the ways you can use them to solve real and very visual and audio problems. But for now, let's at least connect it to something that's been all too germane in recent months, the past few years, namely artificial intelligence, which is a topic we'll come back to at the end of the course, too, to give you a sense of what the connection is with what everyone's been talking about in the world of AI and what it is we're going to spend the next few weeks building up to by writing code.

If you were to try to implement something like a chat bot, for instance, that just answers questions and has a conversation with you, you could do that using pseudocode, and as we'll soon see, you can use C, Python, any number of other languages too. That pseudocode might look like this when implementing a chat bot. You could tell the chat bot, if the student says hello to you, then say hello back. And the indentation, as per earlier, implies this is conditional.

Else if the student says goodbye to you, say goodbye to the student. Else if the student asks you how you are, say you are well. So you can just enumerate question after question after question and just handle all of these conditional possibilities. But things kind of escalate quickly, especially with the tools of today like ChatGPT. Are we really going to have the wherewithal as programmers to write another conditional like else if the student asks why 111 in binary is 7 in decimal-- like, this kind of hints at, oh my God, there's an infinite number of things this human could ask the chat bot.

Do we really have to write an infinite number of conditionals? That's just not possible. Like, there's not enough time in the day, there's not enough lines of code available. Artificial intelligence surely needs to be able to figure some of this out instead. And so indeed, this is not how you implement AI, but rather how you implement an AI like a chat bot is you typically train it based on lots and lots of data.

You give it lots of inputs, lots of inputs, training data, and let it figure out what it should say in response to certain questions. And it boils down to a lot of probability, a lot of statistics, otherwise known now as large language models, which, if we really peek under the hood, are actually typically implemented with what are called neural networks inspired by the world of biology, whereby we humans have all of these neurons that transmit electrical signals such that my brain tells my hand to move this way, this way, and this other way.

And so what computer scientists have been doing over the past many years is implementing in software using literally zeros and ones, graphs or networks, neural networks, that look a little something like this, where each of the circles represents a neuron, each of the arrows represents a pathway between them, and provides as input to these networks huge amounts of data like all of the internet, all of Wikipedia, all of the books that it might consume as input.

And then the goal of this neural network, as per this single final neuron right here, is to produce an answer to a question. Maybe it's simple like, yes, no, or maybe it's something like the answer to the 111 question or how are you or goodbye or hello or the like. And what these neural networks do is use statistics and probability and try to output the most probabilistically likely answer to this question that's been asked and really just hope that it is correct.

There is no programmer at OpenAI or Google or Microsoft that's trying to anticipate every one of these questions we might ask, not only in English but in other languages as well. So you might be wondering why there's this 8 foot duck on the stage. So the persona that CS50's own AI takes is in fact that of a rubber duck, because it turns out in programming circles-- and this is true long before CS50-- it has often been recommended to students and aspiring programmers that you keep literally a physical rubber duck on your desk.

The idea being, in the absence of a friend, family member, colleague, TA who could answer technical questions for you, if you're alone in your room in Mather at night, you can talk to the duck, maybe door closed, and ask the duck your questions, or, more importantly, talk the duck through what confusion you're having. And the mere act of talking through the problem, explaining logically what you're trying to do, what you're actually doing, and what the error actually is, invariably, that sort of proverbial light bulb goes off and you realize, oh, I'm an idiot.

I hear in my own words where I've gone awry. And even though this duck will never say anything back to you, that alone, rubber duck debugging or rubber ducking, tends to be a valuable programming technique, believe it or not. But thanks to these large language models, we have not only physical but virtual ducks as well. And so available to you will be in this class not tools like ChatGPT and the like, which are, through policy, disallowed. It is not reasonable to use ChatGPT and the like.

But you are allowed and encouraged to use CS50's own AI based tools, which resemble those same tools but know something about CS50 and aspire to behave akin to a good teaching fellow guiding you to solutions as opposed to handing you something outright. So this is a tool that will be available at literally this URL throughout the course. CS50.ai. It will also be embedded in the programming environment you'll soon meet, which is called Visual Studio Code, a cloud based version thereof.

The duck will live in that environment as well, as well as on stage from time to time, which is to say we'll not only talk about, but use throughout the course this thing known as AI. But this is ultimately code that we're going to start writing next week. And unfortunately, this code here is written in a language called C. This is essentially the program that I lost two points on some 25 plus years ago.

It does look admittedly cryptic. That's why today what we'll focus on is not what this code looks like, nor the zeros and ones that that code gets converted to so your computer can understand as input what you want it to do. We're going to focus on a much more visual incarnation of this. But I know thus far this has been a lot. So let's go ahead and take a five minute break here, and when we come back in five, we'll do some actual programming. So see you in five.

All right. So it's now time to solve with actual code some actual problems, albeit in a fun and visual and audio way. But recall that where we left off was this. Starting next week, you'll be writing code that ultimately looks like this, but thankfully, you will not be writing zeros and ones, and no normal person, myself included, can understand what all of these zeros and ones are at a glance. We could take out some paper, pencil, and probably figure it out very tediously.

But this is exactly the point. Computers only understand this stuff, but what we as programmers will start writing today and beyond is code at a higher level. And indeed, this is going to be-- this is going to be frequent within computer science where there's different levels of abstraction that we operate at. And the lowest level, the nittiest gritty, is like the zeros and ones that computer understand.

That's it in this class for zeros and ones. Hopefully you at least have wrapped your mind around why zeros and ones can be used in triples and as bytes to represent higher and higher numbers. But let's just now agree that computers can do that. Let's abstract away from that detail and focus on higher level languages than zeros and ones, namely a language like this. So this is an example of the very first programming language I learned back in the day as per that homework in a language called C.

It's an older language, but it remains one of the most popular languages in omnipresent languages nowadays because it's incredibly fast and it's particularly good at making devices operate quickly. For us pedagogically, the value of C is not that you're probably in Silicon Valley and other such jobs going to be using C yourself that much, but because it's going to provide a conceptual foundation on top of which we introduce other languages, like Python, which is newer and improved, so to speak, that gives you more and more functionality for free out of the box by abstracting away some of the stuff we'll focus on in the coming days first.

So at the end of the day, you should better understand languages like Python and JavaScript and SQL because of your underlying understanding of a language like C. But this is too much for the first day. Many of you will think that this is too much for the second week. But in fact, C is really only sort of scary looking because all of this darn punctuation and syntax, the semicolon, the parentheses, the double quotes, the curly braces, and the like. And I concur.

This is intellectually uninteresting, and a lot of the challenges early on when learning programming is you just don't have the muscle memory that I or some of the teaching fellows might for knowing what symbol should be where. But that's going to come with time and practice, I guarantee it. What we'll do for today, though, is just throw away all of that intellectually uninteresting detail and focus really on ideas.

And some of you might be in your comfort zone here because if back in middle school you were playing with a programming language called Scratch, you were probably using at the time just to have fun in class or out of class, making games, animations, interactive art. What you probably didn't use it for, at least in middle school, was to consider and explore programming languages themselves.

But what's wonderful about Scratch, which is this graphical programming language from down the street at MIT, where it was invented some years ago, is you can program not by using your keyboard per se, but by dragging and dropping puzzle pieces, otherwise known as blocks, that will snap together if it makes logical sense to do so. And what you won't have to deal with is parentheses and double quotes and semicolons and all of that, at least until next week.

But the nice thing about Scratch is that after this week and after the so-called problem set zero, the first assignment in which you'll use Scratch, you'll have a mental model via which it will be easier to pick up all of the subsequent syntax as well. So let's see how we can start programming in Scratch by making the simplest of programs first. You can do this at scratch.mit.edu.

You needn't do this now in the moment. Problem set zero will walk you through all of these steps. But what I've done here is opened up at scratch.mit.edu, precisely the default web page therein. This is after having clicked the Create button in Scratch, which is going to allow me to create my first program. But first, a tour of the user interface here, and what is ultimately available to you.

Well, within the Scratch environment, we'll see a few different regions of the screen. One, we have this palette of puzzle pieces at left. The blue ones relate to motion, the purple ones relate to looks, the pink ones relate to sound, and so forth. So the color of the blocks just roughly categorizes what that block's purpose in life is. We're going to be able to use those puzzle pieces by dragging and dropping them from left to right. In the right here, in the middle of the screen is where I'm going to write my actual programs.

This is where I'll drag and drop these puzzle pieces, lock them together, and actually write my code. What am I going to be coding? Well, I'm going to be controlling one or more sprites. Much like in the world of games are familiar, a sprite is like a character that you might see on the screen. The default character in the world of Scratch is, in fact, a cat that looks like this.

And if in this case, I have just one cat, I can then make that cat do things in his own little world at top right by making the cat move up, down, left, right, spinning around, or doing other things as well. But if you want to introduce a dog or a bird or any number of other custom characters, you just add more sprites and they get their own place in that same world. As for how to think about movement in this world, it's actually pretty familiar, even though it gets a little numeric for a moment.

If Scratch at the moment is in the middle of the screen, the cat is at 0, 0 if

you think about x, y-coordinates or latitude longitude. If you move the cat all the way up, this would still be x equals 0, but it would be y 180. What's the 180? 180 pixels vertically or dots on the screen. This is negative 180 pixels on the screen at the bottom. By contrast, if you go left and right, your x value might change. Negative 240, but y is 0, or positive 240 and y is 0 as well.

But most of the time you won't need to know or care about what the pixel coordinates of the cat are. All you're generally going to care about is the programmer, most likely, is do you want the cat to go relatively up, down, left, or right, and let MIT figure out the mathematics of moving this thing around in most cases.

All right. So let's go ahead and introduce the first of these programs by doing something quite simple, as we did in C there, but a little more simply by writing code as follows. I'm going to go back to scratch.mit.edu. I've already clicked, per before, the Create button. And if I click on the yellow category of blocks here at left-- and I'll zoom in-- we'll see a whole bunch of yellow puzzle pieces. And probably the most common one you will use to write code in Scratch for just this first week is literally when green flag clicked.

Why? Well, if we go back over to the cat's world at top right, notice that above the cat's rectangular world, there's not only a green flag for starting, there's a red stop sign for stopping as well. So let's do this. Let me go ahead and click and drag. When green flag clicked anywhere into the middle and let go. And now I'm going to go to looks, and it looks like there's a whole bunch of purple puzzle pieces here. I'm going to choose something simple like say hello, drag it.

And notice if I get just close enough, it's going to want to magnetically snap together. So I'll just do that and it does its thing. The fact that there's this white oval with text means that is an input to this, say, puzzle piece. I can literally then change what the input is if I want to more conventionally say hello, world. Which in fact, according to lore, was the very first program written in C, and nowadays in most every language, including in Brian Kernighan's book. So hello world is generally the first program that most any programmer first writes.

So that's it as programs go. Let me go ahead and zoom out here. Let me go over to the right and click the green flag, and somewhat excitingly, maybe underwhelmingly, we've now written a program that quite simply says hello world on the screen. Now let's make this a little more technical for just a moment. What is this here puzzle piece, as I keep calling it? It's actually a similar-- it's an incarnation one of the ideas from our pseudocode before. What did we call those actions and verbs last time in my pseudocode?

AUDIENCE: [INAUDIBLE].

DAVID MALAN: Functions. That's right. So these purple puzzle pieces here are indeed functions, and some functions, as we can see, take inputs, like hello comma world. After all, how does Scratch know what to say? You have to provide the cat with input, which is to say functions can indeed take inputs like this. In this case one input, but we'll see opportunities for passing in more input as well.

What the cat is doing though, visually on the screen here at top right, is what's generally called a side effect. Sometimes when you call a function, it does something visually. And in this case, you're seeing literally a cartoon speech bubble, hello world. That is the side effect of this function. So if we now want to map this to our world of inputs and outputs and see where this side effect is, this is the paradigm I proposed at the start of class that is computer science in a nutshell and will be the framework we use literally throughout the class, no matter how-- no matter how the languages in particular evolve.

So what's the input to this particular program? Well, this white oval, hello world is my input. The algorithm, step by step instructions for solving some problem, is implemented in code, this language called Scratch by way of this purple puzzle piece. And the output of that function, given this input, is the side effect whereby the cat indeed says hello world visually on the screen in that speech bubble. So the exact same paradigm with which we began today governs how exactly this cat here works.

Well, let's actually go back to this program and make it a little more interesting than that. Let me go ahead and click the red stop sign. And let me actually use a different type of puzzle piece, another function that does something a little different. First, I'm going to get rid of the say block. So I'm going to not only pull it away, I'm going to drag it over anywhere at left and just let go and it will delete itself automatically. Or I could right click or Control click, and from a little menu I could also explicitly say delete.

And what I'm going to do now is under sensing, which is a light blue shade of puzzle piece-- there's a whole bunch here, but I'm going to focus on this one. Ask something and wait. And the default text is, what's your name? And that's fine. But because it's a white oval, that input can be manually changed by me if I wanted to change the question. I'm going to drag it over here. It's going to magnetically snap together. And I'm OK with that question.

But what do I want to say with the answer? Well, let's go ahead and do this. I could go to looks again. I could grab another say block, let it snap in, and I could say something like, hello, David. But this is going to be the first of many bugs that I make, intentionally or otherwise. Let me click the green flag. Scratch is now, just like in a web browser, prompting me for some input here. So let me go ahead and type in my name. David. Enter. And voila. It works. Hello, David.

I'm kind of cheating, though, right? Because if I zoom out, stop, and play again. Let me type in Julia's name here, enter, and it still says hello, David. So that didn't really implement the idea that I wanted. All right, so how can I fix this? Well, it seems that this time I want more than a side effect. I want to use the value that the human types in. And for this, we need another feature of functions, which is that not only can they sometimes have side effects, something visually happens.

Some functions can hand you back a value, a so-called return value, that will allow you to actually reuse whatever the human typed in. So a return value is something that gets virtually handed back to you and you can store it in something called a variable, like x, y, and z in mathematics, and you can generally reuse it one or more times. So let me actually draw our attention then to, at left, not only the blue puzzle piece, ask what's your name and wait, but notice that there's a special puzzle piece below it, this blue oval called answer, and that represents what a computer scientist would call a return value.

So MIT has kind of bundled it together side by side to make clear that one of those pieces relates to the other. What it means is that I can do this. I can drag this oval and use this oval as the input to the save function. Now, notice it's not the same size, but it is the right shape, so that's OK. Scratch will grow or shrink things to fit properly. But this too isn't quite right. Let me go ahead and do this. Let me go ahead and stop that, click the green flag. I'll type in my name again. D-A-V-I-D. Enter.

And it's just kind of weird or rude. Like, I wanted a hello at least, and it just said David on the screen. OK, so I can fix that. Let me stop with the red stop sign. Let me just separate these temporarily. And I can leave it in the middle there, but they have no logical connection temporarily. Let me go back up to looks. Let me grab a say block, a second one, and let me go ahead and say, just to be grammatical, hello, space. And then I'll reconnect this here.

So at the moment it looks like what I want, I want a hello, comma, and then the return value printed out based on whatever the human typed in. So let me zoom out. Let me click the green flag. Again, what's your name? D-A-V-I-D. And watch the cat's side effect. Enter. It's still not greeting me properly. There's no hello. And if in case it was too fast, let's do it again. Green flag. D-A-V-I-D. Enter. It rudely just says my name, which is weird. What's the bug here, though? It's a little more subtle. Why? Yeah?

AUDIENCE: It's just quickly going.

DAVID MALAN: Yeah. It's just too quickly going over the say command or the say function, in this case. My Mac, your PC, your phone, it's just so darn fast. Both are happening, but too fast for my human eyes to even notice. So we can solve this in a number of ways. I could actually use a different puzzle piece altogether. In fact, MIT kind of anticipated this. Notice the first puzzle piece in purple is say hello for a specific number of seconds, and you can specify not just the message, but the number of seconds, ergo two inputs, otherwise now known as arguments to a function.

An input to a function is just an argument now. And that would be a fix here. I could maybe a little more explicitly do this. I could go under events, scroll down a little bit, and-- sorry, under control in orange, I could grab a wait block and I could kind insert it in the middle. And this might actually help. So I could click on the green flag. D-A-V-I-D. Enter. Hello, David. And I could change the timing to be a little more natural. But what if I want the cat to just say hello, David all in one breath, so to speak. Well, for that I'm going to need to use a slightly different technique as follows.

Let me go ahead and get rid of the wait. Let me get rid of the second say block and stop the cat with the stop sign. Let me go under operators here and let me somewhat cleverly grab this. A join block at the bottom. By default, it's using apple and banana as placeholders, but those are white ovals so I can change those. Let me drag this over the white oval for the save function and let go, and it will snap to fill. Let me go ahead here and type hello, comma, space instead of apple. And what should I do instead of banana?

AUDIENCE: Answer.

DAVID MALAN: Yeah. So it'd be answer return value-- the return value. So let me go under sensing again. Let me just drag another copy of it. And you can use these again and again and again. They don't disappear. I want to drag answer over banana so that the second input to join is actually, if you will, the output of the ask block, like that. And it snaps to fit. So now if I go ahead and click the green flag once more. D-A-V-I-D. Enter.

Now we have the behavior aesthetically that I cared about. But beyond the aesthetics of this, the goal here really was to map it to, again, this same paradigm, which we'll see here. The algorithm and the output and the input for this example are as follows. The input to the say block was, quote, unquote, what's your name? The function, of course, implementing that algorithm in code was the ask and wait block. The output, though, of the ask block recalls not some visual side effect.

It is a return value called answer, like a variable, a special variable like x, y, and z in math. But in this one, we generally in programming describe variables with actual words, not just letters. But this output of the say block, I kind of want to make room for it to pass it into the say block as a second argument. So let's do this. Let's take one step back and propose that now for the join block that I just used. It takes two inputs hello, space and answer. The function in question is indeed the join block.

The output of this had better be hello, David. What do I want to do with the output of the join block? Well, let me clear the screen. Let me move this over, because now the output of the join block is going to instantly become the input

to the say block so that the output now in this multistep process is the side effect of hello, David. So the fact that I nested these blocks on top of one another was very much deliberate. If I zoom in here, notice that hello and answer are on top of join, join is on top of the say block.

And if you think back to high school math, this is like when you had parentheses and you had to do the things inside parentheses before the things outside parentheses. It's the same idea, but I'm just visually stacking them instead. But outputs can become inputs depending on what the function there expects. Let me pause here and see if there's any questions about not so much what the cat is doing, but how the cat is doing this. Questions at hand?

All right. Well, let's make the cat more cat-like and do this. Let me throw away all the say block and just let go there. And let me introduce at bottom left a nice feature of scratch whereby there's also these extensions that tend to use the cloud, the internet, to give you even more functionality. And in fact, I'm going to click on this extension up here, text to speech. And if I click on that, I suddenly get a whole new category of blocks at the bottom. Text to speech. They happen to be green.

But what's nice here is that I can actually now have the cat say something audibly. So let me drag the speak block here instead of the say block. I don't want it to just say hello. Let me stop that. So let me go back under operators. Let me grab another join block, because I threw the other one away. Let me change apple to hello, space again. Let me go to sensing. Let me drag answer to banana again. And now let me hit the green flag and let me type in my name, D-A-V-I-D. And in a moment I'll hit enter and.

COMPUTER: Hello, David.

DAVID MALAN: All right. It's not exactly cat-like, but it was synthesized. But it turns out under these text to speech blocks, there are some others. Set voice to alto, for instance, seems to be the default. But let's change this. So notice that some puzzle pieces don't just take white ovals. They might even have drop downs.

So whoever created that puzzle piece decided in advance what the available choices are for that input per the dropdown. So I'm going to change it to squeak, which sounds-- or actually kitten sounds even more apt. Let me zoom out, click the green flag, type my name. D-A-V-I-D. Enter.

COMPUTER: Meow, meow.

DAVID MALAN: That's interesting. So it doesn't seem to matter what I type. So how about David Malan.

COMPUTER: Meow, meow, meow.

DAVID MALAN: So it seems to meow proportional to how long the phrase is that I typed in. It can get a little creepy quickly. If I change kitten to giant. Let me go ahead and hit Play. D-A-V-I-D. Enter.

COMPUTER: Hello, David.

DAVID MALAN: So you can, for very non-academic ways, start to have fun with this, but just playing around with these various inputs and outputs. But let's actually make the cat do something more cat-like and indeed meow instead of saying any words at all. So let me throw all of that away. Let me go now under sound.

Let me drag the play sound until done. And notice in the dropdown here, by default, you just get the cat sound. You can record your own sounds. There's a whole library of dogs and birds and all sorts of sounds you can import into the program. I'll keep it simple with cat. And let me click the green flag.

COMPUTER: Meow.

DAVID MALAN: All right. So the cat meowed once. If I want the cat to meow again, I could do this.

COMPUTER: Meow.

DAVID MALAN: If I want the cat to meow a third time, I could again hit play.

COMPUTER: Meow.

DAVID MALAN: So this is kind of tedious if to play this game, I have to keep clicking the button, keep clicking the button to keep the cat alive virtually in this way. So maybe I want this to happen again and again and again. Well, let me just do that. Let me sort of drag and drop. Or I could right click or Control click and then a little menu would let me Copy-Paste or duplicate blocks. But I'll just keep dragging and dropping. Let's do this.

COMPUTER: Meow. Meow. Meow.

DAVID MALAN: Cat's kind of hungry, unhappy. So let's slow things down so it's adorable again. So let me go under control. Let me grab one of those wait one second, and I'll plop this here. Another one. Let me plop it here. Click play again.

COMPUTER: Meow. Meow.

DAVID MALAN: Cuter. Less hungry. Sure. But this program is now, I daresay, correct if my goal is to get the cat's meow three times. But now, even if you've never programmed before, critique this program. It is not well-designed, even though it is correct. In other words, it could be better. How, might you think? Yeah?

AUDIENCE: A loop.

DAVID MALAN: So using a loop. And why? Why are you encouraging me to use a loop even though it works as is?

AUDIENCE: It's easier to plot.

DAVID MALAN: Yeah. So to summarize, it's just easier to use a loop because I could specify explicitly in one place how many times I want it to loop. And moreover, frankly, any time you are copying and pasting something in code or dragging the same thing again and again, odds are you're doing something foolish. Why? Because you're repeating yourself unnecessarily.

And this is a bit extreme, but suppose I want to change this program later so that the cat pauses two seconds in between meows. Well, obviously I can just go in here and do two. But what if I forget? And suppose this program isn't, like, five or six puzzle pieces. Suppose it's 50 or 60 or 500 or 600. Eventually I or a colleague I'm working with is going to screw up.

They're going to change a value in one place, forget to change it in another. So why are you inviting the probability of making a mistake? Just simplify things so that you only have to change inputs in one place. So how can I do this? Let me zoom out. Let me throw most of this duplication away, leaving me with just the play and the wait function.

Let me now, under control as well, grab one of these. I could, for instance, repeat as follows. Let me grab a repeat. I'm going to have to move these in two parts. So I'm going to move this down. It's too small, but it will grow to fit the right shape. Then let me reattach it up here. Let me change the default 10 to a 3. And now I think I've done exactly what you were encouraging, which is

simplify. And I click play now.

COMPUTER: Meow. Meow.

DAVID MALAN: Now and.

COMPUTER: Meow.

DAVID MALAN: Yeah. So still correct, but arguably better designed as a result. I can keep things simple and change things now in just one place and it will continue to work. But this is getting a little tedious now, I claim. Like, why am I implementing the idea of meowing? Wouldn't MIT have been better to have just implemented a meow puzzle piece for us? Because the whole thing is themed around a cat. Why is there not a meow puzzle piece? Why do I need to go through all of this complexity to build that functionality?

Well, what's nice about Scratch and what's nice about programming languages in general is you can generally invent your own puzzle pieces, your own functions, and then use and reuse them. So let me go ahead and do this. I'm going to go under my blocks in pink down here. I'm going to go ahead and click make a block, and I'm going to be prompted with this interface here. And I'm going to call this block literally meow, because apparently MIT forgot to implement it for us.

And I'm just going to go ahead and immediately click OK. And what you'll see now is two things. One, on the screen, I've been given this placeholder pink piece that says define meow as follows. So anything I attach to the bottom of that define block is going to define the meaning of meowing. And at top left, notice what I have under my blocks. I now have a pink puzzle piece called meow that is a new function that will do whatever that other block of code tells the cat to do.

So what do I want to do here? Well, I'm going to keep it simple for now. I'm going to move the play sound meow until done and wait two seconds. Though let's change it back to one second to move things along. And now let me drag the meow puzzle piece over to my loop such that now, what's it going to do? It's going to meow three times. And just to be dramatic, out of sight, out of mind.

Let me, for no technical reason, just drag this all the way to the bottom of the screen and then scroll back up just to make the point visually that now meowing exists. That is an implementation detail that we can abstract away, not caring how it exists, because I now know at a higher conceptual level, if I want a meow, I just use the meow puzzle piece, and I or someone else dealt with already how to implement meowing. So now let me go ahead and hit play.

COMPUTER: Meow. Meow. Meow.

DAVID MALAN: OK, so same exact code, but arguably better design because I've now given myself reusable code so I don't have to Copy-Paste those several blocks. I can just use meow again and again. But let's make one refinement. Let me actually scroll down to where I did in fact implement this. Let me Control click or right click on it and let me edit the pink block that I created a moment ago, because I want to practice what I've been preaching about inputs.

So I don't want this function just to be called meow. I want this function to also take an input, and just for consistency with our use of n earlier, which in computer science generally means number, let me meow n times. And just so that this puzzle piece is even more programmer friendly, let me add just a textual label that has no technical significance other than to make this function read left to right in a more English friendly way. Meow n times.

Let me click OK. And now notice this thing at the bottom has changed such that it's not only called meow, there's explicit mention of n, which is a circle, which is exactly the variable shape that we saw earlier when it was called

answer. This is not a return value, though. This is what, again, we're going to call an argument, an input to a function. So let me do this.

I'm going to move this back up to the top so I can see everything in one place, and I'm going to make one modification, because my goal now is to make a new and improved version of meowing that actually takes into account how many times I want the cat to meow. So instead of using a loop in my own program under when green flag clicked, I'm going to detach this temporarily. I'm going to move this away. I'm going to move this code over here, and I'm going to reattach it here.

So focusing for the moment on just the left, meow is now defined as repeating three times the following two functions. Play sound and wait. But that's not quite right. I want to get rid of the three. So what can I do? Because I created this input to the meow function myself a moment ago, I can actually drag a copy of it over right that is change the three to be generally an n.

So now I have a function called meow that will meow any number of times. And what's nice now is my actual program that is governed by that green flag, I can type in three, I can type in 10, I can type in 100, and it will just work. And henceforth, I can, again, dramatically scroll this down so we don't know or care about it anymore. Now my program is a single line whereby this notion of meowing has been abstracted away by just defining my own function or custom block.

Questions, then, about just this idea, this principle of creating your own functions to hide implementation details once you've solved a problem? Therefore, you don't want to have to think about that same problem ever again. And that's the beauty of programming, typically. Questions on what here we just did? No?

All right. Well, let's do this. Let's now make this a little more interactive in code. Let me go to this green flag. Let me scroll down and just throw all of this hard work away that we have copies on the courses website of all of these programs step by step if you want to review them in slower detail. Let's do this. Under control, turns out there's other ways to loop. There's this forever block that will just do something forever. So in the forever block, there's some place for some other code.

And I'm going to move to the control section here and grab one of these if blocks, so one of these conditionals. Let's plug that in here. And now notice if, and then there's this sort of trapezoid-like placeholder that's going to probably fit what? The if is a conditional. Forever is a loop. Say and so forth have been functions. What was the other key term we used?

So a Boolean expression. We need to put one of those yes, no or true, false questions here. So what are those? Well, I've been using Scratch for some years, so I under sensing there's one of these shapes here. Touching mouse pointer, question mark. The question mark literally evokes the whole idea of a Boolean expression being yes, no. It's way too big to fit, but it is the right shape.

So let me drag it. Let go. It's going to grow to fill. And now let me go to sound. Let me grab that play sound, meow until done, and put it inside that conditional such that what kind of program have I just implemented here, arguably? What will this program do when I click the green flag? Well, nothing at the moment.

AUDIENCE: Not touching the cat.

DAVID MALAN: But I'm not touching the cat. So if I move the mouse pointer to the cat.

COMPUTER: Meow.

DAVID MALAN: Again.

COMPUTER: Meow.

DAVID MALAN: Again.

COMPUTER: Meow.

DAVID MALAN: It's kind of implementing the idea of petting a cat, if you will, because I'm forever just waiting and waiting and waiting. Is the mouse pointer touching that sprite, touching that cat? And only if so, go ahead and play that sound meow until done. But now we can make things a little more interesting. Let me stop this and let me do something actually completely different.

Let me throw all this hard work away. Let me go under extensions. Let me go to video sensing, because lots of laptops, my own included, has a little webcam nowadays. Let me approve use of that there. And you can see me in the frame. And let me do this. Let me drag one of these when motion exceeds some measure. And through trial and error, I figured out that 50 tends to work well. Let me step out of frame here and program off to the side.

And if I go to play sound meow until done, notice that this is an alternative to using when green flag clicked. This is a category of block that's constantly waiting for what we'll call an event. An event is just something that can happen on the screen, a click, a drag, a mouse movement, and so forth. So let me zoom out here. And now, if I can do this-- here we go. No, too slow. Still too slow. Wait, did I click play? Let's see. Try again.

COMPUTER: Meow.

DAVID MALAN: There we go. OK. 50 is a little too high, apparently. So let's make this a little gentler. 10.

COMPUTER: Meow.

DAVID MALAN: OK, well.

COMPUTER: Meow.

DAVID MALAN: There we go.

COMPUTER: Meow.

DAVID MALAN: There we go.

COMPUTER: Meow.

DAVID MALAN: OK, so we've implemented now more physically the idea of actually responding to petting a cat.

COMPUTER: Meow. Meow.

DAVID MALAN: Oh, damn it. OK.

COMPUTER: Meow. Meow. Meow.

DAVID MALAN: All right. So this is a bug. Like now-- this is MIT's fault. So it's not stopping in response to the red stop sign. So what do you do in doubt? Most extreme, you reboot. For now, I'm just going to close the window. OK. So now we've seen all of those primitives that we saw in that pseudocode, but incarnated in this graphical programming language, and again, without parentheses and semicolons and double quotes and all that punctuation that we will introduce before long.

But for now, we have the mechanisms in place where we can do some really interesting things. So in fact, I thought, in the spirit of thinking back on

olden times, thought I'd open up the very first program I wrote when I actually took-- I was cross-registered in an MIT class and took a class that introduced aspiring teachers to Scratch.

And I implemented this program here called Oscartime, which was a game that used a childhood song that I was a fan of and it allows you to drag trash into a trash can. But to bring this to life and perhaps in exchange for one stress ball, could I get one brave volunteer who wants to come up and control this here keyboard? I saw your hand first. Come on up. Come on up. And you'll see, thanks to the team, we also have this amazing lamppost here, being on Quincy Street as we are. Do you want to introduce yourself to the group?

AUDIENCE: Hi, my name is Anna. I'm from Richmond, Virginia, and I'm in Weld.

DAVID MALAN: Nice. Weld.

AUDIENCE: Yes!

DAVID MALAN: All right, come on over. So here, Anna, you'll have a chance to play the very first game I wrote in Scratch, which admittedly is more complicated typically than we would expect of a student doing this for the very first time, as in problem set zero. But what I'm going to do is full screen this here. I'm going to click the green flag, and what you'll see on the screen are these instructions. Drag as much falling trash as you can to Oscar's trashcan before his song ends. And here we go.

[OSCAR THE GROUCH, "I LOVE TRASH"]

Oh, I love trash

Anything dirty or dingy or dusty

Anything ragged or rotten or rusty

Yes, I love trash

DAVID MALAN: There we go. So as Anna continues to play, let's tease this apart a little bit. So one, there's some costumes on the stage. Like that lamppost is actually never going to move. But there's a couple of sprites. There's the trash can, which seems to be a character unto itself. There's this piece of trash that keeps coming back and back. That is a sprite. There's now this sneaker, which is another sprite. And in fact, notice that Oscar, of course, keeps popping up from his sprite once in a while.

So Oscar seems to have multiple costumes. So I offer this as an example, as you keep playing, if you would. Very good job so far. The song goes on forever. This was a nightmare to implement, to listen to this all day long. But how do we implement the rest of this? Well, notice that the trash, every time she throws into the trash can, does reappear somewhere different.

So there's some kind of randomness involved. And indeed, Scratch will let you pick random numbers in a range. So maybe it could be negative 240, maybe it could be positive 240, at the 180 point on the top of the screen. So you can randomly put things on the screen. There's apparently what kind of construct that makes the trash fall again and again. I think no one's listening to me. They're all just watching you.

What's making the trash fall from top to bottom? So it's actually some kind of loop because there's a motion block inside of a forever loop, probably, that just keeps moving the trash one pixel, one pixel, one pixel, one pixel, one pixel, creating the illusion, therefore, of motion. And if we can crank the song a little bit more, you'll see that this is all synchronized now.

OSCAR THE GROUCH: (SINGING) Because they're trash

Oh, I love trash

Anything dirty or dingy or dusty

Anything ragged--

DAVID MALAN: The song keeps going forever, seemingly. And now notice more and more sprites are appearing because they waited for-- here we go. Climax.

OSCAR THE GROUCH: (SINGING) I love trash

DAVID MALAN: All right. A big round of applause for Anna. Nicely done. OK, here you go. Here you go. All right. So this is an interminable song. And indeed, I spent hours building that, and just listening to that song on loop was not the best way to program. But the goal here is to really use it as just an intellectual exercise as to how that was implemented. And we won't do the entire thing in detail, because I will say back in the day when I was younger, I didn't necessarily write the cleanest code.

And in fact, if we see inside this and we poke around the bottom of the screen here, you can see all of my different sprites. And the code is kind of complex. Like, things just kind of escalated quickly. But I did not set out and write all of these programs all at once for each sprite. I pretty much took baby steps, so to speak. And so, for instance, let me open up just a few sample building blocks here that speak to this that are written in advance.

So here's version zero. Computer scientists typically start counting at zero. And let me show you this example here that only has two sprites on the screen. We have Oscar the trashcan and we have the piece of trash. And now notice, what does Oscar do? Well, let me go ahead and zoom in on this script, as it's called. A program is a script. When the green flag is clicked, Oscar switches his costume to Oscar one. That's his default costume where the lid is closed.

Then Oscar does this forever. If Oscar is touching the mouse pointer, change the costume to Oscar two, otherwise change it back to Oscar one. So that whole idea of animation where Oscar is popping in and out is just like a quick costume change based on a loop inside of which is a conditional waiting for the cursor, like Anna did, to get near the trash can.

Meanwhile, if we look at the piece of trash here, notice that the trash is actually not doing anything in this first version because I didn't even implement falling first. So let me hit the green flag. Nothing is happening in this very first version. But notice, if I click on the trash and drag as soon as I'm touching Oscar, there comes that trash can lid. And it was just the result of making this one program respond to that input. All right. What did I do next?

Well, next, after taking that single baby step, I added one other feature. Let's see inside this version one. Again, Oscar is behaving the exact same way. But notice this time the trash is designed to do the following. First, I'm telling the program that the drag mode is draggable. That is, I want the trash to be movable when the user clicks on it. Then I tell the piece of trash to go to a random x location. x is the horizontal, so it's going somewhere between 0 and 240, but all the way at the top of the screen. 180.

Then forever, the piece of trash just changes by negative one. So it just moves down and down and down. And without looking at the second script yet, let me just hit play. And notice, without even doing anything-- and eventually, once there was lots of trash falling, like Anna was struggling to keep up with this. It's just moving one pixel at a time forever until, thankfully, MIT does stop things automatically if they hit the bottom, lest a six-year-old get upset that all of a sudden their sprite is gone forever.

So there is some special casing there. But what else is this trash doing? Let me zoom in here. The piece of trash also, when the green flag is clicked, is forever asking this question. If you are touching Oscar, then pick a new random location between 0 and 240 at positive 180 and go back to the top. So in other words, as soon as this piece of trash is dragged over to Oscar like this and I let go, it recreates itself at the top. It's just sort of teleporting to the top, and thus was born this feature.

And I won't slog through all of the individual features here, but if we do just one more and see inside this one-- so now let me go ahead and hit Play. Notice at the top left of the screen, there's a score. Currently zero. But now when I click the trash and let go, notice that the score is being incremented by one. And this, in fact, is how, Anna, your score kept going higher and higher and higher.

Every time I noticed, oh, the trash is touching Oscar, let's not only teleport, let's also increment a variable. And we didn't see this before, but if I go to this Oscar Scratch now, you'll see that it is exactly the same. But if I now go to the trash piece here and we go to when green flag clicked, you'll see that I'm initializing a variable in orange called score to zero. But if we scroll down to the bottom, Oscar is also doing another thing in parallel at the same time.

When the green flag is clicked, Oscar is forever checking, is the piece of trash touching Oscar? If so, change the score by one and then go to top, which is another location on there, that screen. So in other words, even though at a glance something like Oscar time might look very complicated and it did take me hours, the goal, especially with problem set zero, is not going to be to bite off all of that at once, but to take proverbial baby steps. Implement one tiny feature so that you feel like you're making progress.

Add another feature, another. And invariably you might run out of time and not get to the best version of your vision, but hopefully it'll be good. Hopefully it'll be better, but you'll have these sort of mental milestones, hoping that you at least get to that point. Because as you will soon discover, everything in the world of programming unfortunately takes longer than you might expect. That was true for me 25 years ago and is still true today. Well, let me introduce one final set of examples here.

This one written by one of your own predecessors, a former student. Let me go ahead and open up three baby steps, if you will, toward an end of implementing a game called Ivy's Hardest Game, whereby it's now more interactive, quite like Oscartime. So at top right here, notice-- and I'll zoom in-- we have this world that's initially very simple. Two black lines, two walls, if you will, and a Harvard sprite in the middle. But when you click the green flag, notice that nothing happens initially except that the sprite jumps to the middle.

But I can hit the up key or the down key or the left key or the right key. But if I try to go too far, even though it's not the edge of the world, it's only touching that there black line, it's still going to stop as well. So intuitively, how could you implement that type of program? How could you get a sprite from what we've seen to respond to up, down, left, right, but actually move when I touch my arrow keys? Like, what does it mean to move? Yeah?

AUDIENCE: Maybe if then.

DAVID MALAN: Exactly. So much like with representing information, at the end of the day, all we've got is zeros and ones. When it comes to algorithms, at the moment, all we have are functions and loops and conditionals and Boolean expressions and soon some more things too. But there's not all that much we have at our disposal. So let me zoom out from this and let me actually show you what the Harvard sprite is doing. It's doing this.

When I go up to the green flag here, the Harvard sprite is going to 0, 0. So

dead center in the middle. And then it's forever doing two things, listening for the keyboard and feeling for walls, left and right. Now, those are not puzzle pieces that come with Scratch. I created my own custom blocks, my own functions to implement those ideas. Let's not abstract away for now. Let's actually look at these features. And indeed, to your instincts at left here, what does it mean to listen for the keyboard?

Well, if the up arrow key is pressed, change y by one. Move up. If the down arrow key is pressed, change y by negative one. If the right arrow key is pressed, change x by one. If the left arrow key is pressed, change x by negative one. So take all the magic out of moving up, down, left, right by just quantizing it as plus, minus, this, and that. It's all numbers, indeed, at the end of the day. But what else is it doing? Notice that it did, indeed, bounce off the wall.

So my other custom function, which I chose, feel for walls to evoke this idea, it's asking two questions. If you're touching the left wall, then change x by one, so bounce in the other direction. Else if you're touching the right wall, bounce in the negative one direction. And so what are left wall and right wall? I mean, I kind of cheated. I just used two more sprites. These sprites are literally nothing except black lines.

But because they exist, I can ask that question in my conditional saying, are you touching those other sprites? And I could have colored them any way I want, but this is enough, if I zoom in, to implement this idea of going up, down, left, and right, and preventing the sprite from leaving that little world. All right. So if you'll agree that there's a way now to implement motion up, down, left, right, let's go ahead and implement this idea by adding a rival into the mix, like a Yale sprite.

And what the Yale sprite is going to do, if I click the green flag, is this. So Harvard at the moment is still going to be movable with the arrow keys, up, down, left, right. But Yale, for better or for worse, is just going to mindlessly bounce back and forth from left to right forever, it would seem. The operative word being forever. So how is that working?

Well, let's look. Here's the Yale sprite at the bottom. Let's zoom in on its actual code here. The Yale sprite starts at 0, 0. It points in direction 90 degrees, which means left, right, essentially. And then it forever does this. If touching the left wall or touching the right wall, turn around 180 degrees. So I don't want the Yale sprite to just stop by moving it one pixel to bounce off slightly. I want it to wrap around and just keep going and going and going forever.

And that's it. Everything else is the same. So one final flourish. Let's add a more formidable adversary, like MIT here, whereby if I zoom in and hit play, notice that if I move the Harvard sprite, MIT comes chasing me now. Now, how is this actually working? Yale is just kind of doing its thing, bouncing back and forth. Now MIT has really latched on to me and it's following me up, down, left, right. So how is that logic now working?

Well, again, it's probably doing something forever, because that's why it's continually doing it. Let's click on MIT. This too is pretty simple, even though it's a pretty fancy idea. Initially the MIT sprite goes to a random position, but thereafter, it forever points toward the Harvard logo outline, which is just the long name that your predecessor or former student gave the name for that sprite. And then it moves one step, one step, one step.

So suppose this were an actual game, and in games things get harder and harder, the adversary moves faster and faster. How could we make MIT even faster by changing just one thing here? Like, how do we level up? Change the one to two pixels at a time, two steps at a time. So let's see that. Let's go ahead and zoom out. Let's hit play. And now notice that MIT is coming in much faster this time.

All right. So it wasn't noticeably faster. Let's do this. Let's move 10 steps at a time. So 10 steps faster than originally. I mean, now-- and now notice it's kind of twitching back and forth in this way. Why? Well, probably, if we worked out the math, probably the MIT sprite is touching the sprite and it's bouncing off of it, but then it's realizing, oh, I went too far. Let me move back. Wait a minute. I'm still touching it. Let me move down. So you can get into these perverse situations where there is actually a bug, be it logical or aesthetical.

But in this case, we probably want to fix that. So 10 is probably too fast for this to work particularly well. But the final flourish here really is to show you the actual version of a game that one of your predecessors, a past classmate, actually implemented. Before, thereafter, we will adjourn for cake in the transept, which is the CS50 tradition. But can we get one more final volunteer to come on up to play Ivy's Hardest Game? I'm seeing your hand most enthusiastically there. Yeah, come on down. Very happily.

[APPLAUSE]

In just a moment, we will indeed adjourn. But the goal here now is going to be to navigate a maze that's a little more difficult than the last. Let's have you first, though, introduce yourselves to your classmates in front.

AUDIENCE: Hi, y'all. I'm Eric. I'm from Philadelphia and I'm also from Hollis Hall.

[CHEERS]

DAVID MALAN: One person from Hollis. Nice. OK. Welcome. All right. Eric, go ahead and take the keyboard here. It, too, will be all about up, down, left, right as soon as you click the green flag. And if we can crank the music.

[MC HAMMER, "U CAN'T TOUCH THIS"]

You Can't touch this

DAVID MALAN: So notice, the black walls are a little more involved than last time. But the goal is to get to the sprite all the way at right and just touch it, at which point you move to the next level. The next level, of course, has Yale doing its thing back and forth. You've made it to level three.

But now there's two Yale. So another sprite is in the mix that's randomly moving a little different in terms of direction. Three Yales. Next level. MIT is in. Nice.

The walls are now gone. Princeton's in the mix. Nice. Two Princetons. OK. New life. OK, another life. Nice. Nice. Oh. Nice. Second to last level. Three Princetons. Last level. Yeah! Congratulations.

[APPLAUSE]

Thank you. All right. This, then, was CS50. Welcome aboard. Cake is now served.

[MUSIC PLAYING]