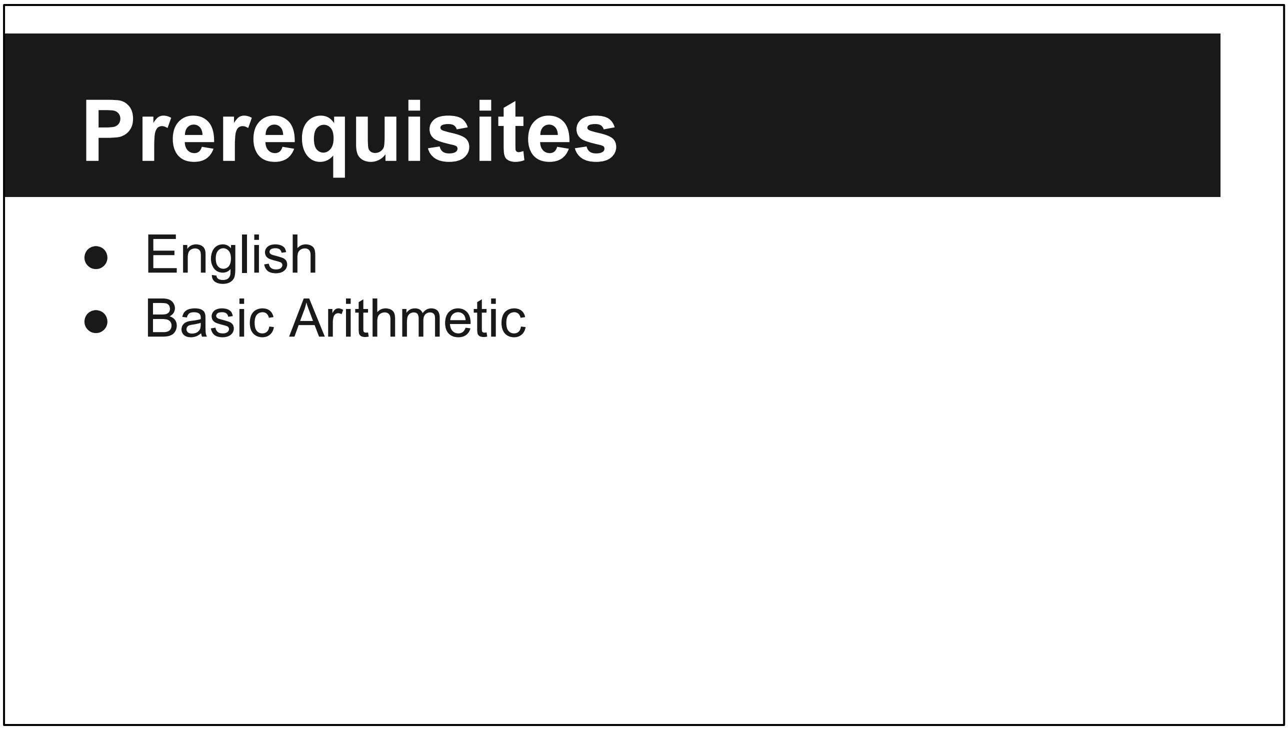
Intro to Computer Science

This is going to look familiar to anyone who’s seen the demo lecture.
It’s the same stuff, minus the “trying to convince you to come to the school” part.

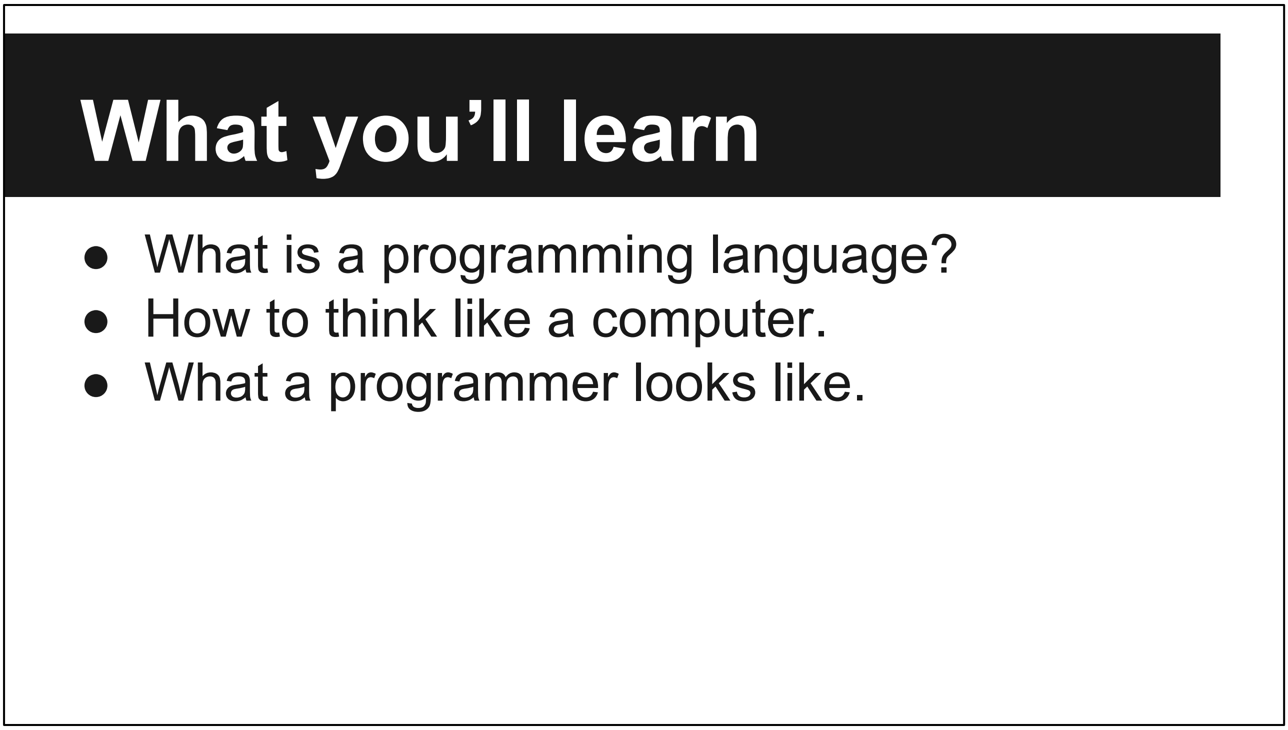


Prerequisites

- English
- Basic Arithmetic

The only thing you really *need* to know to get through this course is English. We’re going to show you that learning a programming language is like learning a foreign language. If at some point in your life you learned to read and write English, you can learn to read and write code.

Some basic arithmetic would be useful as well (e.g. addition, subtraction, etc.). But, in a pinch, we can teach you that as well. Come find me or a TA afterwards.



What you’ll learn

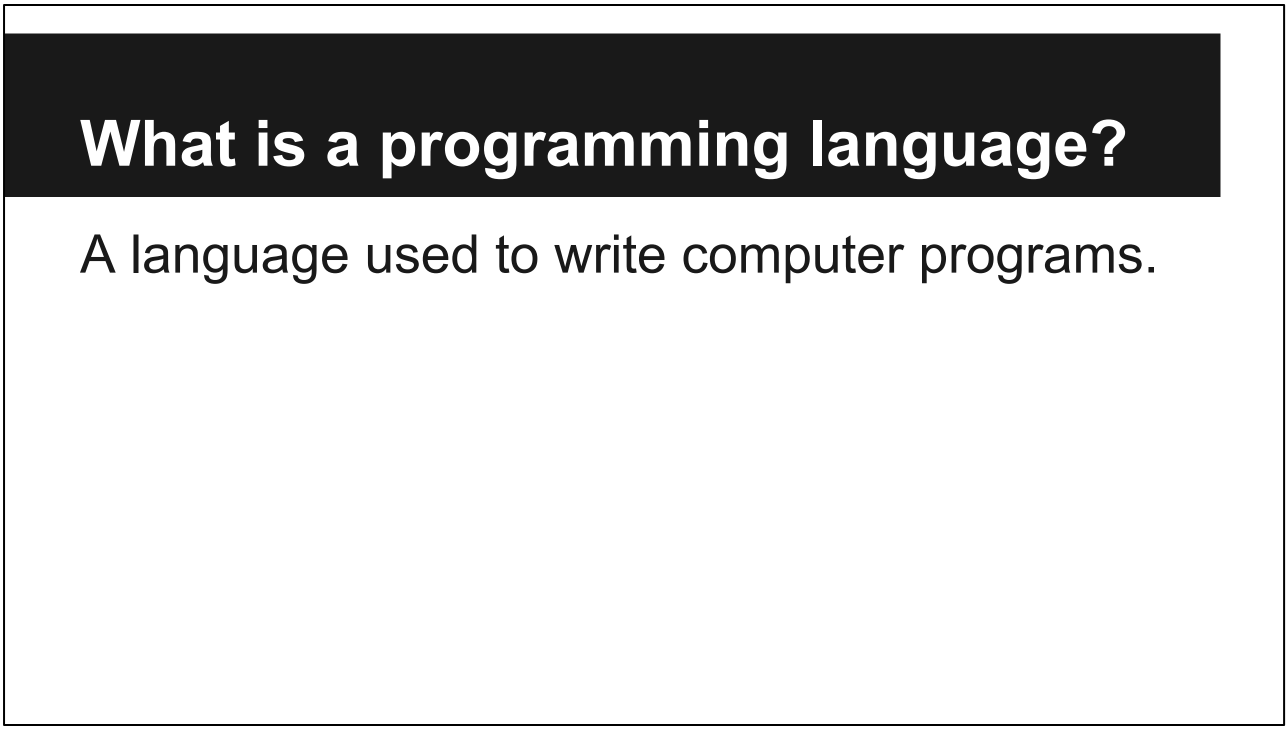
- What is a programming language?
- How to think like a computer.
- What a programmer looks like.

Here are the things you’ll learn in this lecture.

1. What is a *programming language*?
2. How to think like a computer.
3. What a programmer looks like.

To read code, you need to understand how a computer works.
You need to *think* like a computer.

We’ll also digress into a short discussion about diversity in the software industry.



What is a programming language?

A language used to write computer programs.

Let’s start at the beginning. Let’s define the term “programming language”.
It’s a language used to write computer programs. That’s not very enlightening.

I’m going to break it down into two words. We’ll talk about “language” first, then “programming”.

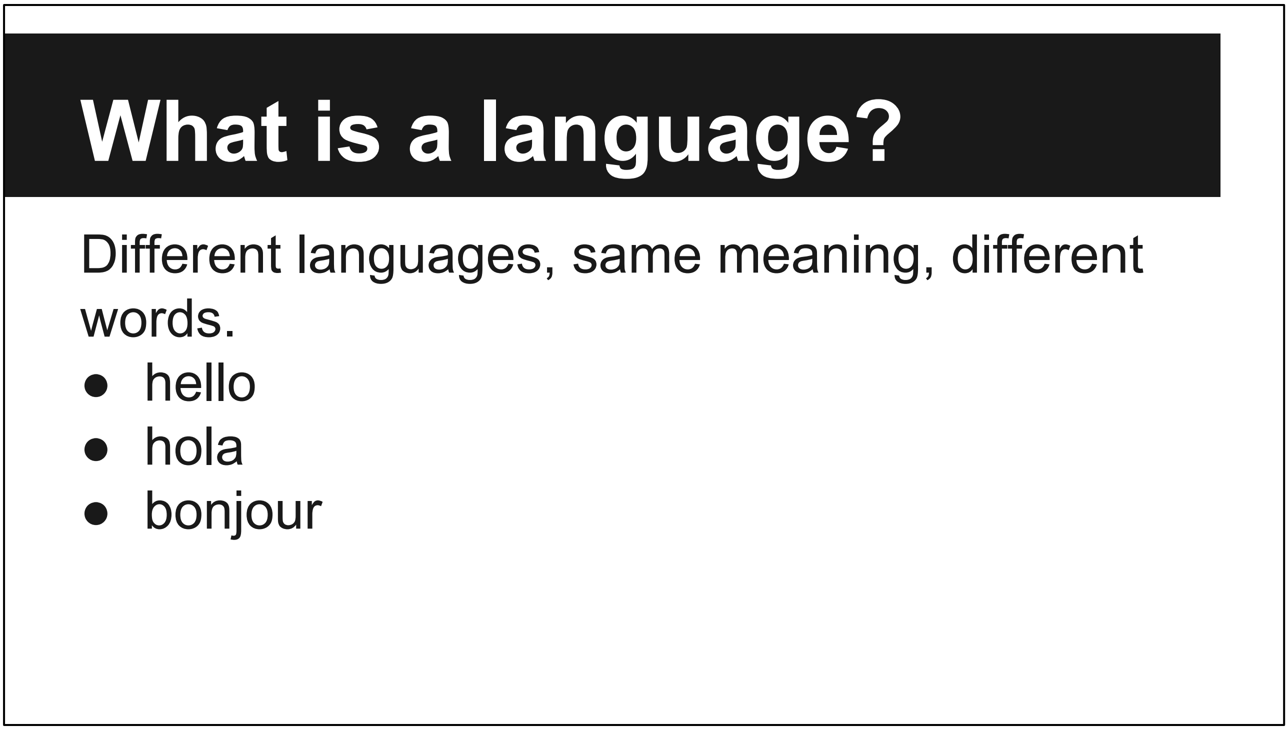
What is a language?

A language is a means of communication.

First, what’s a language?

Let’s not make this complicated. We’re not teaching linguistics.
A language is just a means of communication.

There are only two things I want to emphasize about *human* languages.



What is a language?

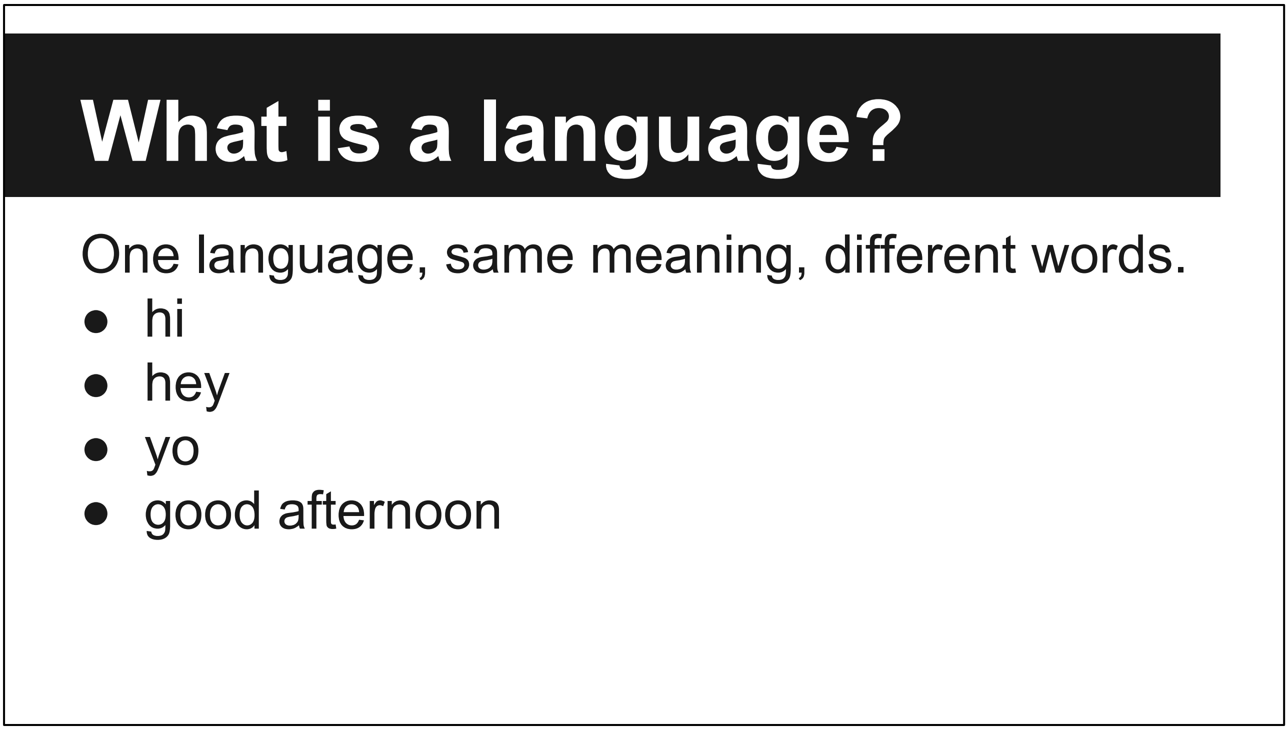
Different languages, same meaning, different words.

- hello
- hola
- bonjour

First, *different* human languages can express the *same* concept in many ways.

For example, here are three different ways to say “hello” in different languages.

Different languages, same meaning.



What is a language?

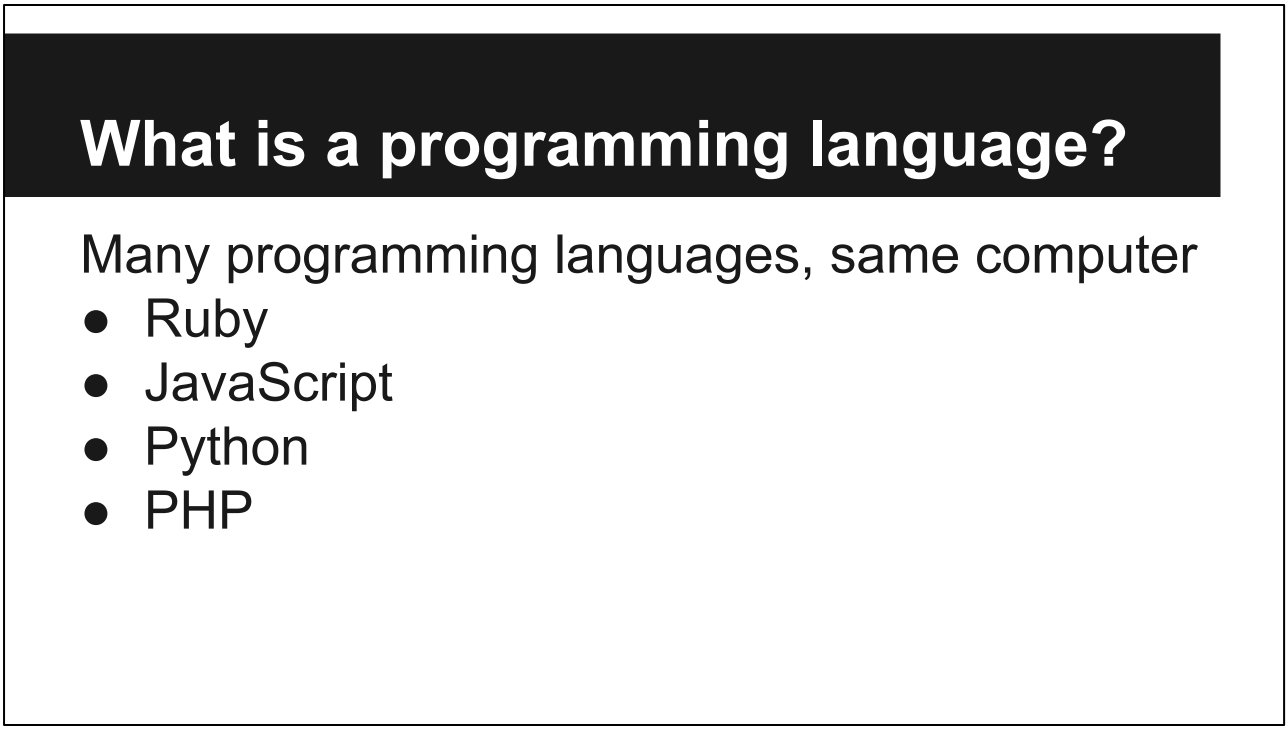
One language, same meaning, different words.

- hi
- hey
- yo
- good afternoon

And second, *one* language can express the *same* thing in many ways.

Here are 4 ways to say hello in English.

Same language, same meaning.



What is a programming language?

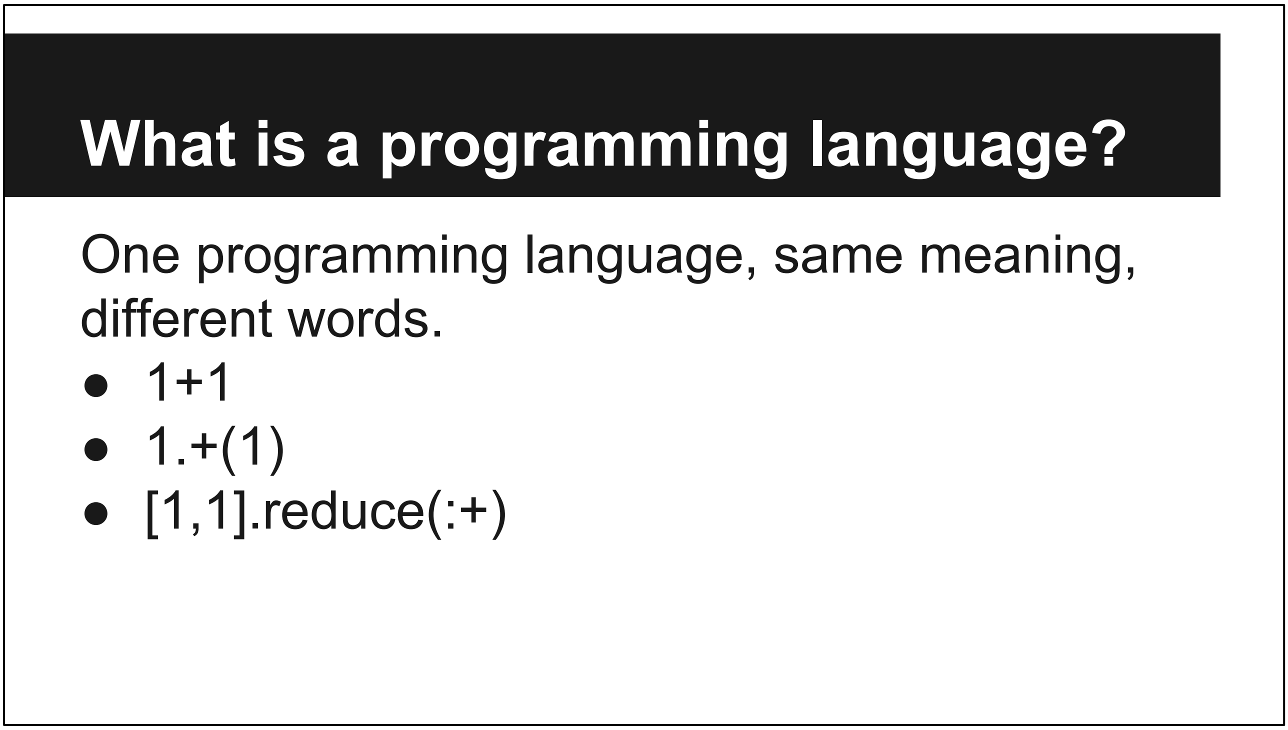
Many programming languages, same computer

- Ruby
- JavaScript
- Python
- PHP

The same rules apply to programming languages. There are *many* programming languages. For example:

Ruby, JavaScript, Python, PHP, Java, Swift, C, C++, C#, etc.

Different languages, all the same to a computer.



What is a programming language?

One programming language, same meaning, different words.

- `1+1`
- `1.+(1)`
- `[1,1].reduce(:+)`

And within each programming language there are many ways to express the same concept.

Here are three ways to express 1+1 in Ruby, from most to least readable.

Same language, all the same meaning to a computer.

The Art of Code

Programming is art.

Start by writing ugly code that works.

Done is better than perfect.

Programming, like creative writing, is an art form. There’s no one *right* way to construct an English sentence. There are just better or worse ways, depending on context. (Like the code on the previous slide.)

A junior English speaker and a junior programmer can both get the job done. They can both get their points across. They can both express what they mean.

But it takes time to learn how to do the job well. It takes time to learn how to express what you mean eloquently.

In the beginning, we’ll teach you *how* to program. And you’ll write ugly, awkward code that works, just like a new English speaker will write awkward sentences that still “work”.

Later we’ll teach you how to program *well*. But writing well-designed code takes *a lot* of practice. Even after 9 weeks you won’t be writing code that reads like Shakespeare or looks like DaVinci.

As you’ll someday learn, lots of gainfully employed devs write terrible code. And they still get paid because they can still get the job done.

We want you to be better than that. But first, remember that “Done is better than perfect”.

DD: Some people say “Perfect” is the enemy of “done”.

Complete Language

If human languages and programming languages were so similar, this would just be a foreign language class. It’s not. So how are the two languages different?

One obvious difference is that, instead of one person talking to another person, it’s one person talking to a computer. That comes with some baggage. For example, there’s something called “completeness”.

ET: There are many “incomplete” human languages, meaning there are languages that just don’t have words for certain concepts. Eskimos might not have words for “mango” or “papaya”. The Aztecs probably didn’t have words for “kangaroo” or “space shuttle”.

DD: When talking about human languages, it’s easy to think of ones that don’t have the same vocabulary as other ones. For example, there might not be native terms in Inuit for tropical fruit like a mango or a papaya, and unlike Inuit, English doesn’t have specific words to describe drifting snow, clinging snow, snow on the ground, or the crust on fallen snow. But of course I have just managed to express those concepts.

ET: “Completeness” has a different meaning for computers. We don’t talk about what a computer can “understand” (that’s a whole other bag of worms - machine learning). Instead, we talk about what a computer can do. A computer is a tool. A “complete” programming language is a language that can take *full*-advantage of that tool.

DD: “Completeness” for computers is all about that expression. We don’t talk about what a computer can “understand” (that’s another sub-discipline called Machine Learning). Instead, we talk about what a computer can do. A computer is a tool. A “complete” programming language is a language that can take *full*-advantage of that tool.

If you imagine a computer as a Swiss-army knife, a “complete” programming language can be used to access all the pieces: the scissors, the knife, the nail file, the bottle opener, and so on.

A “complete” programming language is referred to as a “Turing complete” programming language. Not all programming languages are Turing complete, and thus not all languages are capable of taking full-advantage of everything a computer can do. But most are.

Let’s make a computer!

So what does it really mean for a programming language to be “complete” (or “Turing complete”)? What’s on the list of *all* the things a computer can do?

To better understand the concept of “Turing completeness”, we need to know how a computer works. To understand how a computer works, we need to know how it’s made. So let’s make one.

We’re not going to worry about the hardware just yet: the monitor, the keyboard, the mouse, the mysterious box. We’ll talk about some of that stuff later.

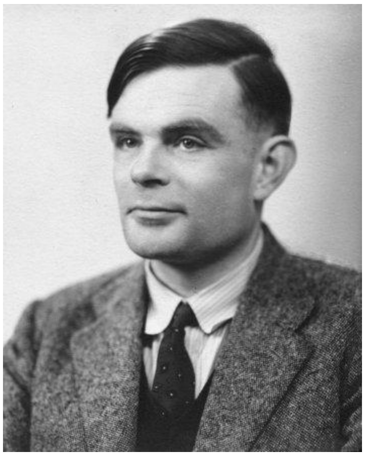
We’re going to start by inventing the *idea* of a computer.

T.A.R.D.I.S.

To do that, we’re going to have to travel back in time to meet...

Alan Turing

- b. 1912, London
- King’s College Cambridge
- Princeton
- WW2 German code breaker
- gay
- died tragically in 1954



Alan Turing, the father of computer science.

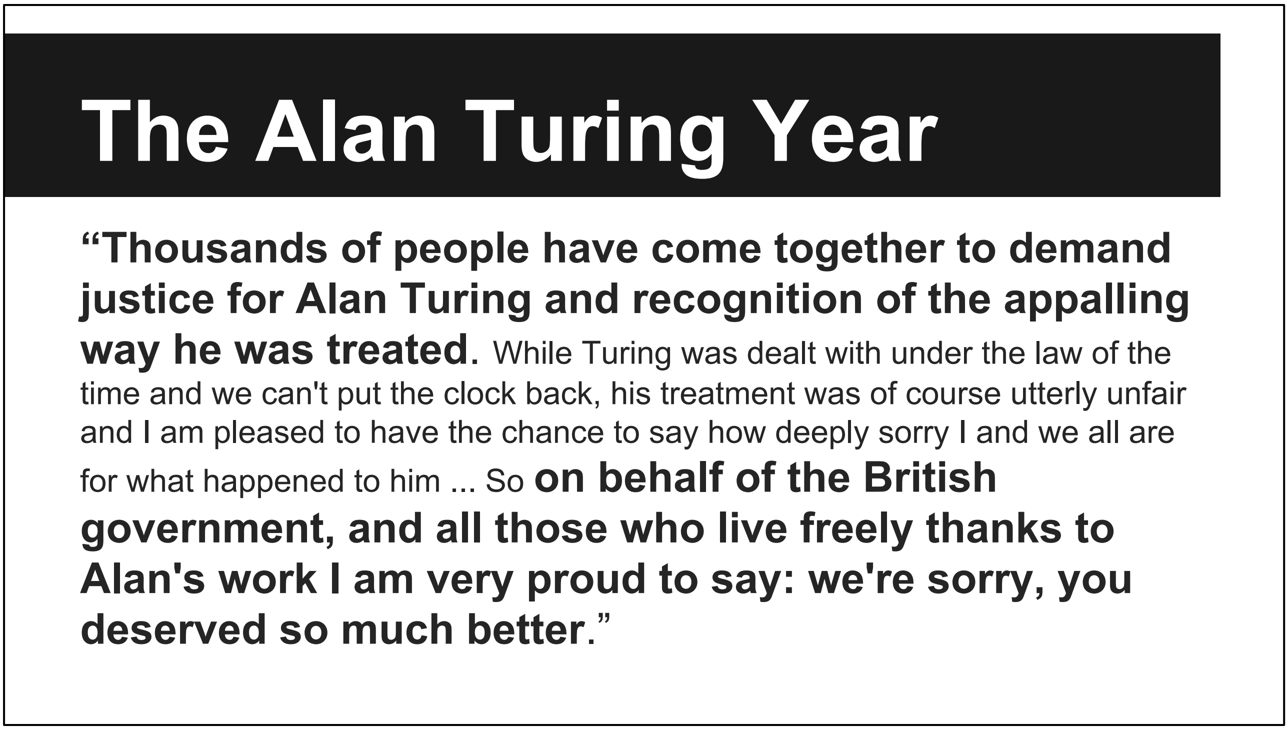
Alan Turing was a British mathematician born in 1912. He attended King’s College in Cambridge as an Undergrad and transferred to Princeton for his PhD.

He helped the allies spy on German communications in WW2 (NSA-style).

He was also gay, which wasn’t an easy thing to be in the middle of the 20th Century. It led to him being charged with “gross indecency”, forced to undergo a chemical castration, stripped of his security clearance, and denied entry into the U.S.

This was a man who helped us defeat the Nazi’s...

Alan died tragically of cyanide poisoning in 1954. It may, or may not, have been a suicide.



The Alan Turing Year

“Thousands of people have come together to demand justice for Alan Turing and recognition of the appalling way he was treated. While Turing was dealt with under the law of the time and we can't put the clock back, his treatment was of course utterly unfair and I am pleased to have the chance to say how deeply sorry I and we all are for what happened to him ... So **on behalf of the British government, and all those who live freely thanks to Alan's work I am very proud to say: we're sorry, you deserved so much better**.”

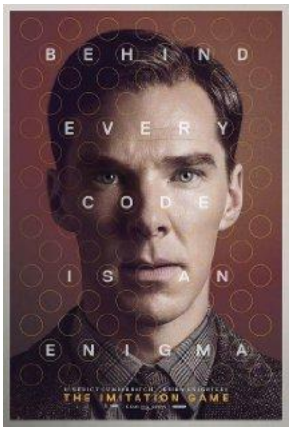
The 100th Anniversary of his birth was just a couple years ago (2012). In his honor, the British Parliament introduced and debated a bill to pardon his conviction. But before they could act, Queen Elizabeth herself granted him a royal pardon this past December.

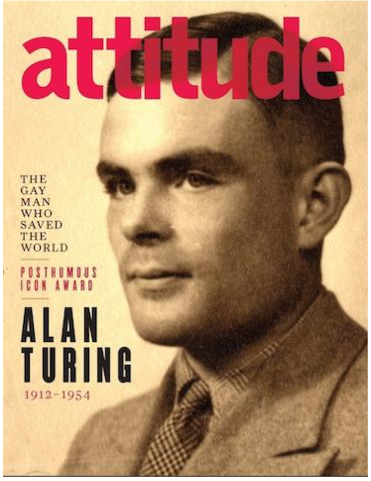
2012 was dubbed “Alan Turing Year” (https://twitter.com/alanturingyear) by the The Alan Turing Centenary Advisory committee. Over 40 countries participated with events and celebrations.

But a few years prior, in 2009, as Britain marked the 70th Anniversary since the start of World War II (“it’s finest hour” according to Winston Churchill), then British Prime Minister Gordon Brown released an official public apology for on behalf of the British government:

“on behalf of the British government, and all those who live freely thanks to Alan's work I am very proud to say: we're sorry, you deserved so much better.”

Alan Turing





Turing continues to receive press.

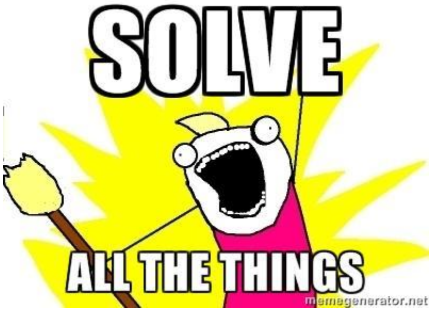
He’s the subject of a movie that came out in November called “The Imitation Game”. He’s played by Benedict Cumberbatch.

He has also inspired a musical (or maybe “concept album”) entitled “A Man from The Future” and written by the Pet Shop Boys. Some of the songs from the musical premiered this past July in London.
https://www.youtube.com/watch?v=E2yG-Uwg3Iw

Most recently, he was honored as “The Gay Man Who Saved The World” by Attitude Magazine last October.
http://www.huffingtonpost.co.uk/2013/10/16/gay-marriage-alan-turing-attitude-awards_n_4108367.html

The Turing Machine (1936)

A machine capable of solving any and all mathematical and logical problems.



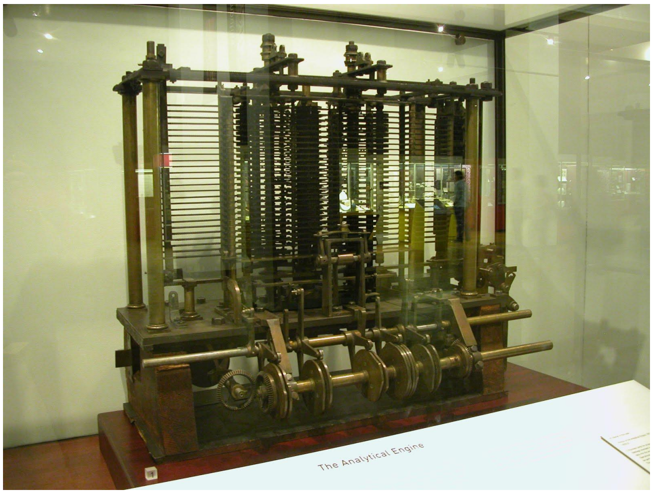
So what makes Turing so special? He worked on lots of subjects, but in particular...

He published a paper in 1936 where he introduced the concept of a “Universal Machine”, later known as a “Turing Machine”, that was capable of solving all mathematical problems.

It’s not a calculator. It could do much more than basic arithmetic. Turing’s “Universal Machine” could solve problems in Algebra, Calculus, Geometry… pretty much every problem you can imagine in math and logic.

What would such a magical machine look like?

Babbage Analytical Engine (1837-1871)

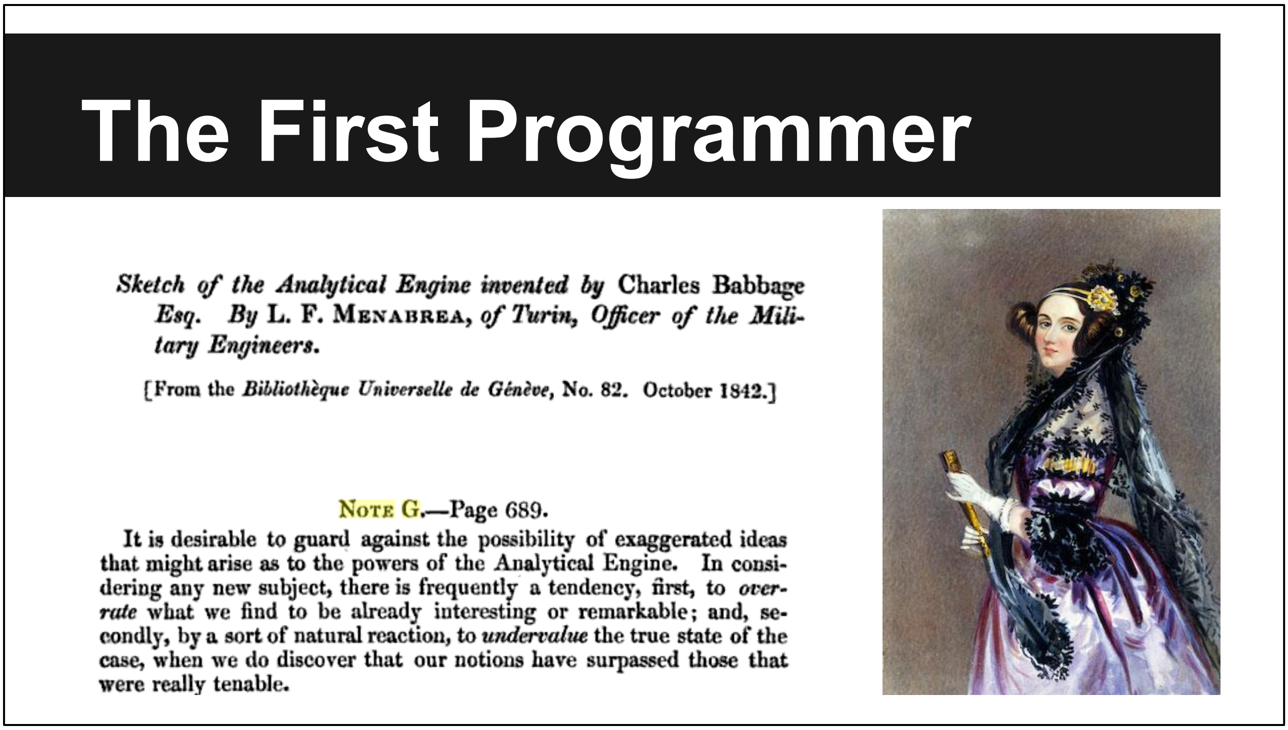


You might imagine a Turing Machine would look something like this. This is Charles Babbage’s Analytical Engine. It was first proposed in 1837 and unfinished by his death in 1871. This is the only piece of the machine he completed before he died. It currently sits in London’s Science Museum.

If you’re paying attention to the dates, you may have noticed that this device predates Turing’s 1936 paper by almost *100 years*. This design was only later described as the first “Turing machine”, the world’s *first* computer.

If anyone out there is a fan of steampunk fiction, most of that work is inspired by imagining a world in which this fully-functioning computer was actually built and working in Victorian England.

The world wouldn’t see another general purpose computer like this for another 70 years (the 1940s).



The First Programmer

Sketch of the Analytical Engine invented by Charles Babbage *Esq. By* L. F. Menabrea, *of Turin, Officer of the Military Engineers.*

[From the *Bibliothèque Universelle de Génève*, No. 82. October 1842.]

Note G.—Page 689.

It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. In considering any new subject, there is frequently a tendency, first, to *overrate* what we find to be already interesting or remarkable; and, secondly, by a sort of natural reaction, to *undervalue* the true state of the case, when we do discover that our notions have surpassed those that were really tenable.

The first computer also came with the first programmer. Meet Ada Lovelace.

Ada Lovelace was born in 1815. She’s the daughter of Lord Byron, a notable English Romantic poet.

Ada spent a year translating a French paper on the Analytical Engine into English. It sounds like a typical secretarial task, but Ada went above-and-beyond.

She extended the translation with copious notes, lettered A through G. The final note, Note G, contained instructions for how to use the engine to calculate a particular mathematical sequence of numbers, the Bernoulli numbers.

The notes were republished and rediscovered in 1953, over a century after Ada’s death. Ada’s Note G is the world’s *first* computer program.

The First Programmer

“the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent”

As the daughter of a leading Romantic-era poet, Lovelace was also the first to note the potential of a “calculating machine” to do more than simple number crunching.

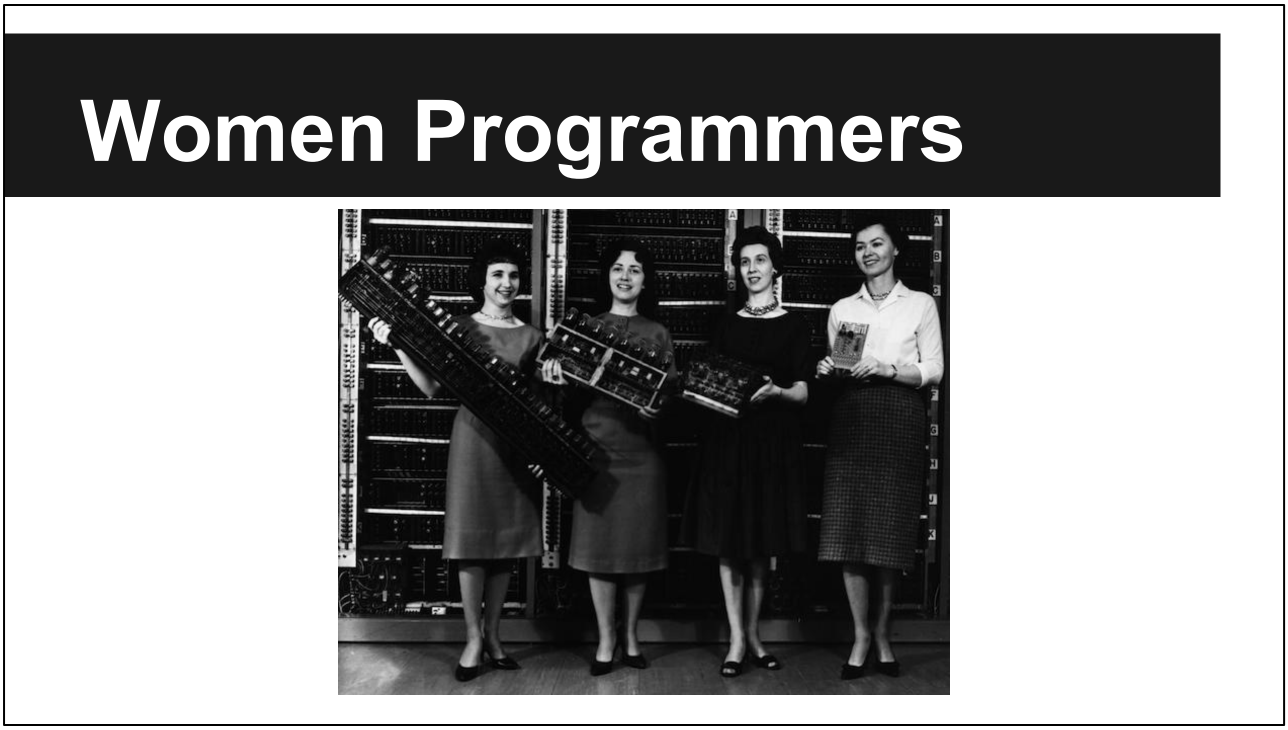
In Note A, she wrote, “[T]he engine might compose elaborate and scientific pieces of music of any degree of complexity or extent”.
http://books.google.com/books?id=qsY-AAAAYAAJ&dq=sketch%20of%20the%20analytical%20engine&lr&as_drrb_is=b&as_minm_is=0&as_miny_is=1840&as_maxm_is=0&as_maxy_is=1852&as_brr=0&pg=PA666#v=onepage&q=elaborate&f=false

Lovelace described her approach as "poetical science". This was a conceptual leap from previous ideas of the potential of a general computing machine. This was *electronic music*, circa 1843.

So next time you’re at Ultra, tell them you’re into the really really old-school stuff.

So Ada Lovelace is a big deal. In her honor, a day in mid-October is set aside as Ada Lovelace Day to celebrate the achievements of women in STEM (science, technology, engineering and math).

There are also a number of organizations, a programming language, some buildings, and even a tunnel boring machine all named in her honor (Why tunnel boring? Because England).



Women Programmers

There has been a lot of talk lately about women and programming (and diversity overall in the software industry).

Suffice to say that programming was, at one point, considered “woman’s work”.

The first general purpose electronic computer, the Electronic Numerical Integrator And Computer (or ENIAC), was programmed by young women mathematicians (pictured here). Such work “lacked prestige”, according to an NPR article from this past October.

http://www.npr.org/blogs/alltechconsidered/2014/10/06/345799830/the-forgotten-female-programmers-who-created-modern-tech

In the 1940s, programming was not considered “manly”. “Men” engineered and built the devices. “Women” simply used them, like secretaries or phone operators.

“In the 1930s female math majors were fairly common — though mostly they went off to teach.”

Jean Jennings Bartik, a member of the team of young women mathematicians who programmed the ENIAC, recalled “The ENIAC wasn't working the day before its first demo.”. Her team “worked late into the night and got it working.”

Then, the next day:

quote - "They all went out to dinner at the announcement," she says. "We weren't invited [but] there we were. People never recognized, they never acted as though we knew what we were doing. I mean, we were in a lot of pictures." - unquote

At the time, [“media outlets didn't name the women in the pictures”].

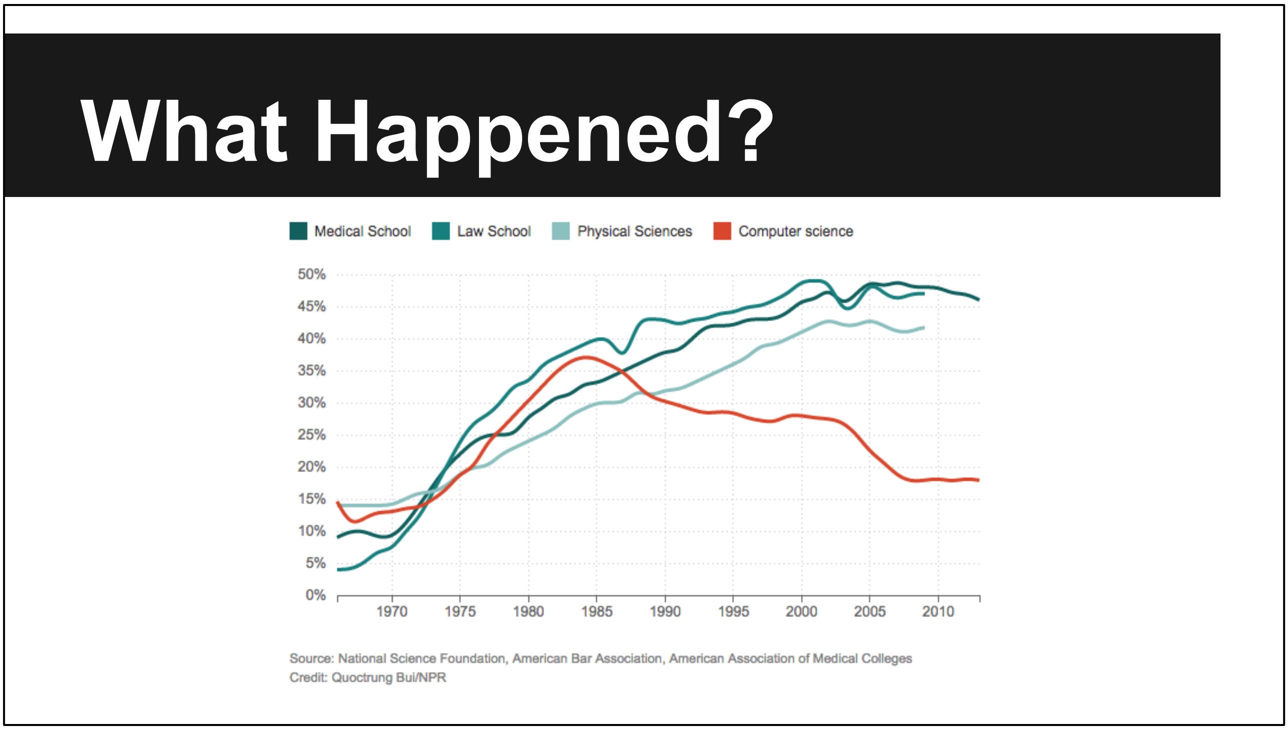
Women Programmers

After WW2, the growing demand for computer talent allowed these “keypunch girls” (as they were called) to be promoted to “system analysts” (because we didn’t use the term “programmer” back then).

United States Navy Rear Admiral Grace Hopper spent the post-war era (1940s-80s) working on early computers with the young lady mathematicians who worked on ENIAC. Before the war, she was a tenured math professor. “Amazing Grace” (as she was later referred to) created COBOL, the first programming language to use words instead of numbers.

“Amazing Grace” is also a big deal. She has a U.S. Navy Destroyer and a supercomputer named after her (the USS Hopper and the Cray XE6 "Hopper", respectively).

Grace Hopper will make another appearance a bit later in the course.



What Happened?

Source: National Science Foundation, American Bar Association, American Association of Medical Colleges
Credit: Quoctrung Bui/NPR

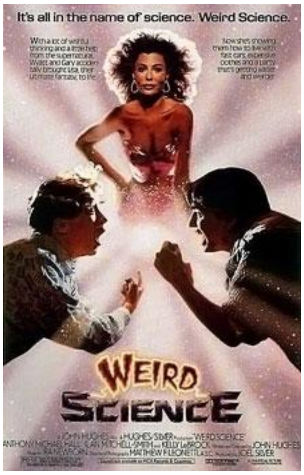
So what happened to all the women coders?

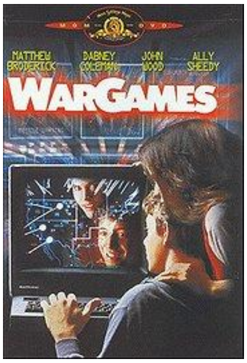
We know *when* it happened.

This chart shows the % of female majors by field. Medicine, Law, Physical Sciences: all have been on the rise since the 1970s.

That red line, on the other hand, represents “Computer Science”. Something happened in the 1980s that start turning women away from computer programming. What was it?

The 80s Happened





The 80s happened.

As computers entered the home, marketing companies decided that the home personal computer should be a “boy toy”. And the movie industry followed suit, showing time and again young boys (young *white* boys) playing with these new, expensive toys (with their girls on the side).

NPRs Planet Money podcast talked about this last October, interviewing some of the women behind the very first software consulting companies.
http://www.npr.org/blogs/money/2014/10/17/356944145/episode-576-when-women-stopped-coding
http://www.npr.org/blogs/money/2014/10/21/357629765/when-women-stopped-coding

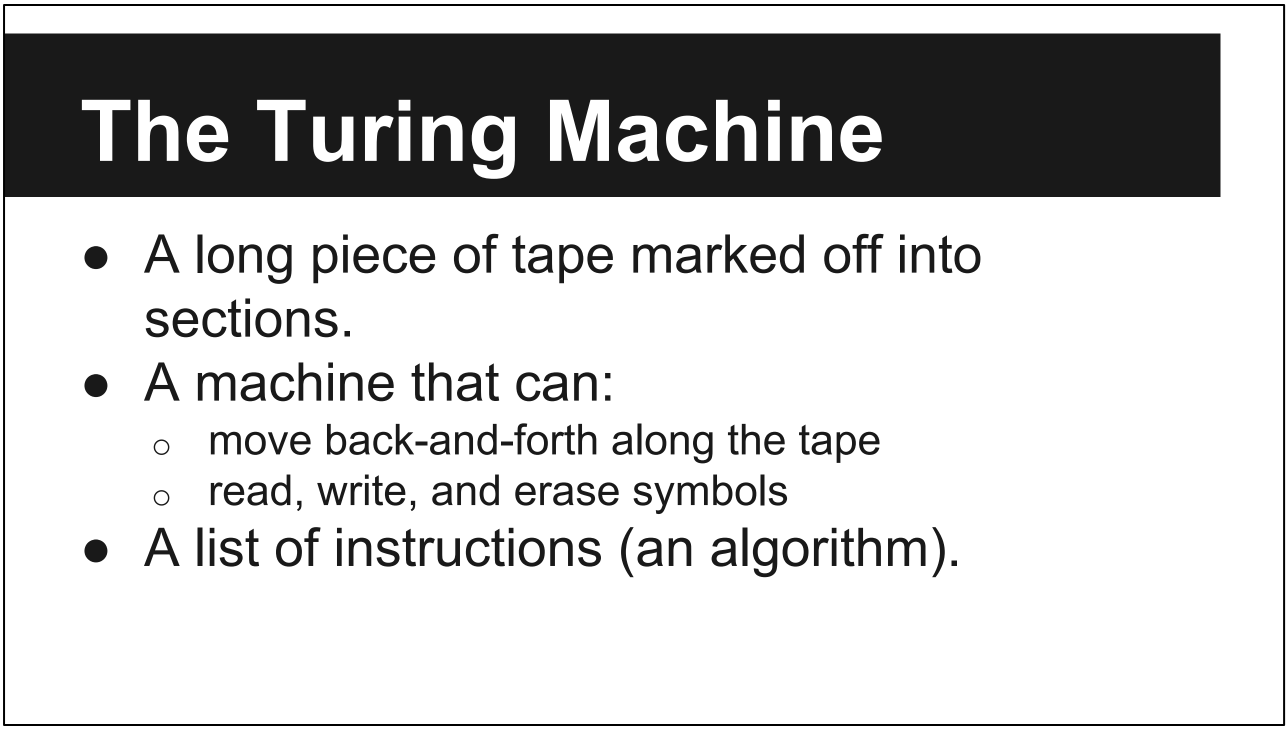
Many people believe programming is primarily a white male world. It’s all Mark Zuckerburgs and Bill Gates’s.

But it wasn’t always like that. It’s almost an accident that we ended up the way we are today. Blame it on Radio Shack.

Radio Shack color computer commercial: https://www.youtube.com/watch?v=1CDkHs4lzUo
Seriously, why wasn’t that little girl allowed to play? And why is everyone *so* white? Like Finish/Canadian white.

Wyncode wants to promote a diverse base of programmers. We have no tolerance for intolerance or anything that contributes to a hostile environment for our proudly diverse student body.

That extends to anything either I or the staff might say or do to make anyone less than completely comfortable here.



The Turing Machine

- A long piece of tape marked off into sections.
- A machine that can:
  - move back-and-forth along the tape
  - read, write, and erase symbols
- A list of instructions (an algorithm).

Back to Turing.

Babbage’s design for the Analytical Engine was so complicated he couldn’t finish building the darn thing before he died, spending over 30 years on the project.

It turns out, however, that Turing’s description of a “Universal Machine” is much simpler. All you need is...

Sound familiar?

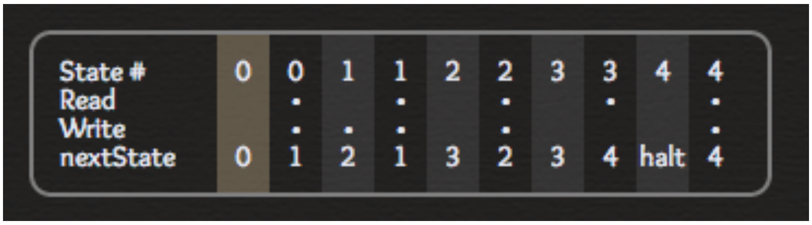
Does this sound familiar? (Is there anyone in the room too young to know what this is?)

The hardware inside a modern computer isn't actually that far off from what you see here.

A computer isn't an unimaginably complicated device. The concept is actually quite simple. The materials are complicated. To help you understand programming, you need to demystify your understanding of how a computer works. So let's do that now.

Turing Machine Algorithm

Let’s solve 1+1...



State #	0	0	1	1	2	2	3	3	4	4
Read		•		•		•		•		•
Write		•	•	•		•				•
nextState	0	1	2	1	3	2	3	4	halt	4

Here’s a Turing machine algorithm for adding 1+1. I’m going to go through it on the board right now for you.

* * => **

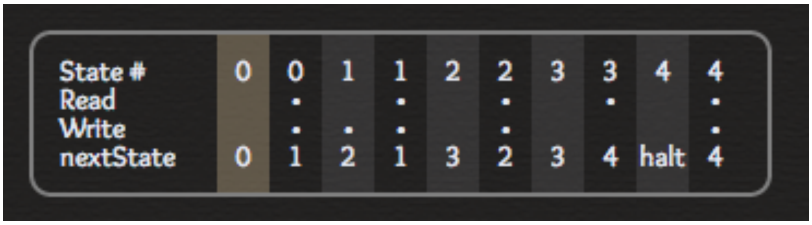
Some of you may not have been able to follow along with that. That’s ok. We won’t be asking you to create or even read these tables. But there are some important lessons hiding in here.

First, note the frequent use of “if this, then that” logic. It’s called “Boolean logic” and computers use it a lot. We’ll learn more about that later.

Second, note the concept of state, and the way a computer works by transitioning between states. A computer is a “state machine”. This is a subtle point, but we’re going to revisit it later as well.

Turing Machine Algorithm

Let’s solve ~~1+1~~ *any addition* problem...



State #	0	0	1	1	2	2	3	3	4	4
Read		▪		▪		▪		▪		▪
Write		▪	▪	▪		▪				▪
nextState	0	1	2	1	3	2	3	4	halt	4

Another important thing to note is that this algorithm actually works to add together any two (whole) numbers.

For example, it’ll turn 2+1 (** *) into 3 (***). To save time, I won’t go through that with you.

At a high level, what the computer is doing is shifting the block of dots on the right over 1 slot. It’s actually very similar to the way you’d teach a child addition.

But it’s not obvious, from looking at this table, to see what’s happening. Just like it’s not obvious, looking at a human brain, to see how it actually works.

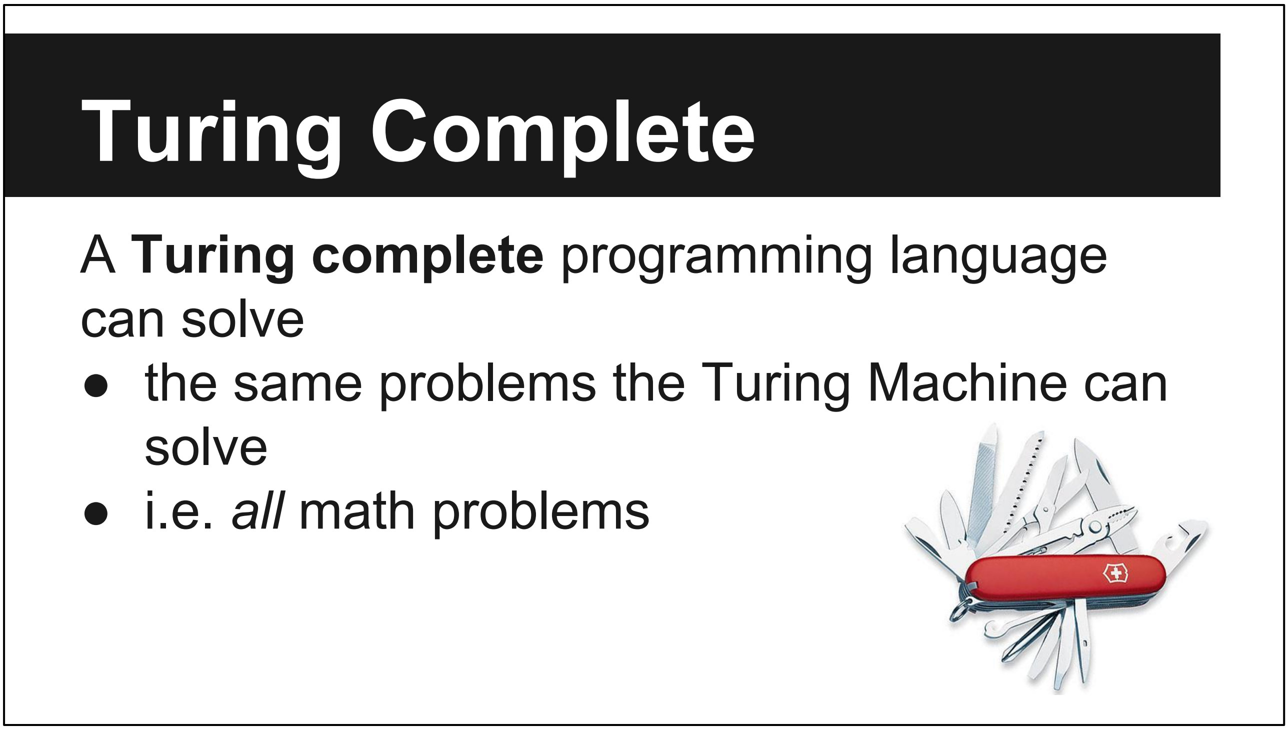
Where am I going with this? Why go through this exercise if we won’t see another Turing Table after today?

The point is that this rudimentary machine can solve *any* mathematical problem. This happens to be what simple addition looks like. You’ll just have to imagine what the table for Calculus looks like. It’s probably huge. But the problem is solvable. Turing proved it.

It’s also important to realize that programming languages, behind the scenes, are translating the words you type into these tables. This table is how a computer thinks - deep down inside that mystery box. A programming language’s job is to translate your English words into this computer-speak.

And, while the table is complicated, the machine that executes this “algorithm” isn’t. This piece of tape, these dots, that table of instructions, and that state *are* a computer. Moving my marker right and left, writing and erasing dots: this is how a computer works. This is how a computer *thinks*. Given enough time, we could solve any math problem using this process.

p.s. Keep that Turing tape in mind. While we won’t be seeing the table again, we will be seeing the tape later.



Turing Complete

A **Turing complete** programming language can solve

- the same problems the Turing Machine can solve
- i.e. *all* math problems

We started this journey talking about language completeness - how a programing language is considered “Turing complete” when it can take full advantage of everything a computer can do. What does that mean?

[slide]

So a computer (a Turing machine) is capable of solving all math problems. And a Turing complete programming language is capable of using that computer completely.

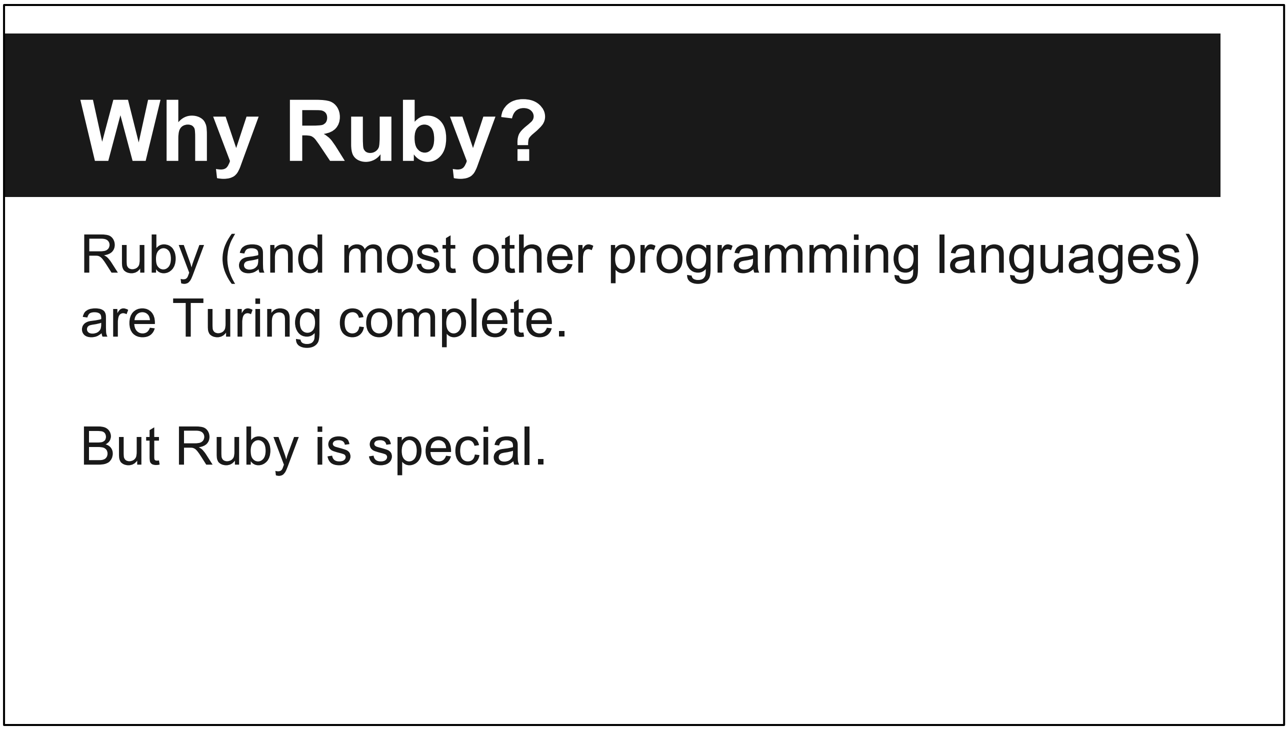
This has an important logical implication. All Turing complete programming languages are the *same*. They’re all equally powerful. They can all be used to solve the same set of problems - namely all the math problems in the world.

What that means is that every program you can write in Ruby, you can also write in Python, PHP, Java, C#, or whatever. They’ll all look different, some will work faster than others, but they can all accomplish the exact same things. You don’t get any special powers using one language vs. another. It’s only a question of efficiency for the task at hand.

This isn’t the first lesson in learning Ruby. This is the first step to learning any programming language. This the first step to computer science.

Caveat: Infinity is big thing. Technically, a Turing machine requires an infinitely long piece of tape. A real computer will run out of space eventually (for example, if I type

1+1+1+1... long enough). Nevertheless, the laptops in front of you are all still considered Turing machines.



Why Ruby?

Ruby (and most other programming languages) are Turing complete.

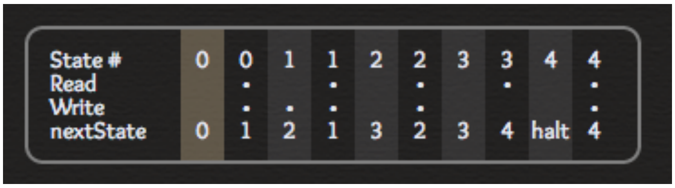
But Ruby is special.

Ruby (and most other programming languages) are Turing complete. So why learn Ruby instead of one of those other ones?

1. Because all the other coding bootcamps use it.
2. Because Ruby is special.

Ruby

Programming languages have traditionally been written so machines can easily read them.

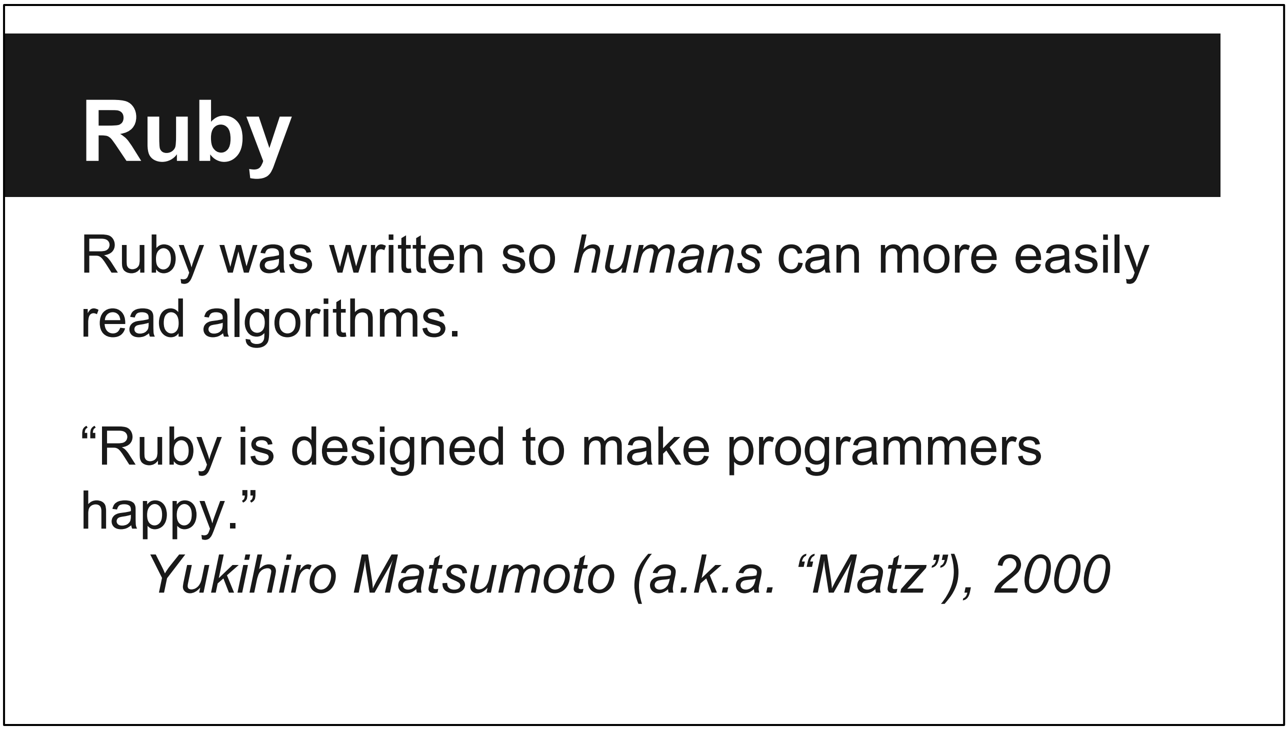


State #	0	0	1	1	2	2	3	3	4	4
Read		•		•		•		•		•
Write		•	•	•		•				•
nextState	0	1	2	1	3	2	3	4	halt	4

[slide]

Computers *love* these tables. While it takes me a few minutes to run through this algorithm by hand, a modern computer can run through *billions* of these operations *per second*: move right, read a dot, write a dot - move right, read a dot, erase a dot.

This addition algorithm is a piece of cake for a modern computer.



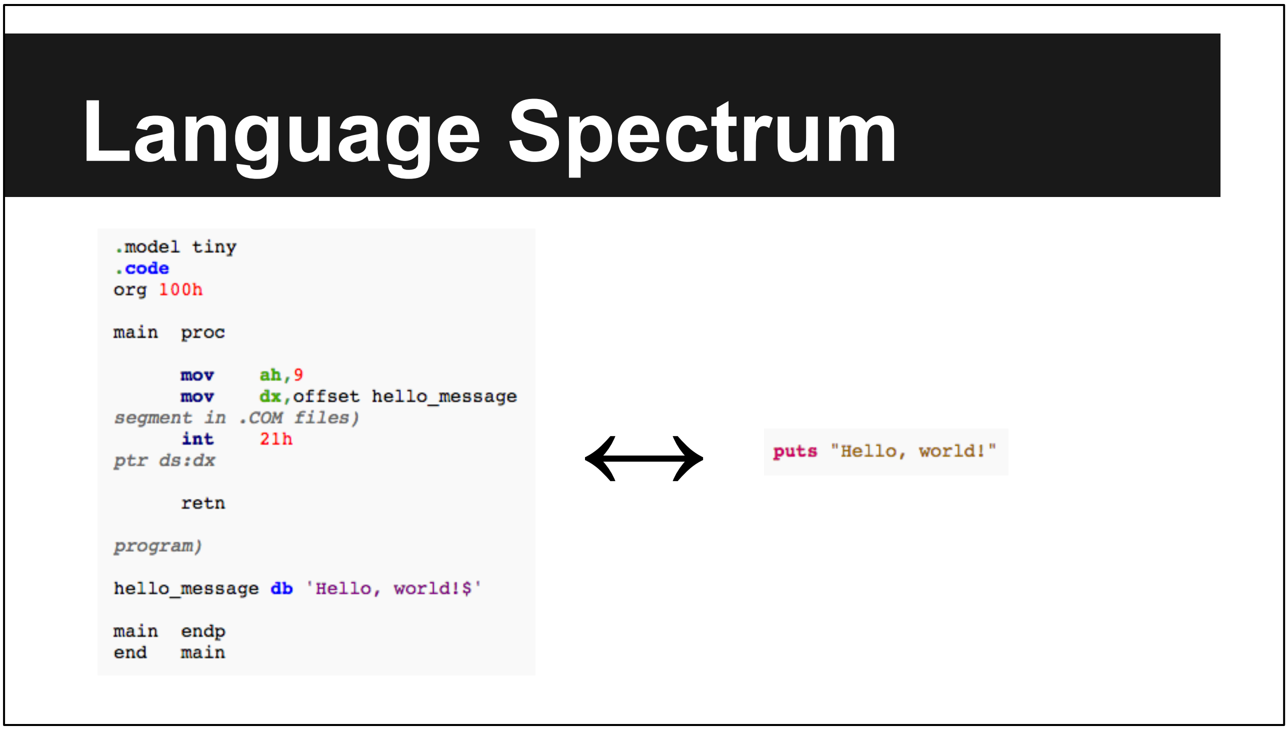
Ruby

Ruby was written so *humans* can more easily read algorithms.

“Ruby is designed to make programmers happy.”
    Yukihiro Matsumoto (a.k.a. “Matz”), 2000

[slide]

Matz created Ruby in the mid-90s in Japan.



Language Spectrum

```
.model tiny
.code
org 100h

main  proc

      mov    ah,9
      mov    dx,offset hello_message
segment in .COM files)
      int    21h
ptr ds:dx

      retn

program)

hello_message db 'Hello, world!$'

main  endp
end   main
```

↔

```
puts "Hello, world!"
```

Imagine a programming language spectrum.

At one end you've got “machine language”. Computers understand it well, but it's very hard for humans to read and recognize.

At the other end you've got “high level” languages like Ruby and Python, which are easier for humans to read, write, and understand.

Computers have gotten more powerful over the years. But humans haven't evolved as much. High-level languages like Ruby believe the computer should be doing more of the work for us.

We shouldn't be adapting to our tools. Our tools should be adapting to us.