The following is a conversation with Chris Latner, his third time on this podcast.
As I've said many times before, he's one of the most brilliant engineers in modern computing, having created LLM compiler infrastructure project, the Clang compiler, the Swift programming language,
a lot of key contributions to TensorFlow and TPUs as part of Google.
He served as vice president of Autopilot software at Tesla, was a software innovator and leader at Apple.
And now he co-created a new full stack AI infrastructure for distributed training,
inference and deployment on all kinds of hardware called modular and a new programming language called Mojo.
That is a superset of Python, giving you all the usability of Python, but with a performance of C, C++.
In many cases, Mojo code has demonstrated over 30,000 X speed up over Python.
If you love machine learning, if you love Python, you should definitely give Mojo a try.
This programming language, this new AI framework and infrastructure and this conversation with Chris is mind blowing.
I love it. It gets pretty technical at times, so I hope you hang on for the ride.
And now a quick few second mention of each sponsor.
Check them out in the description. It's the best way to support this podcast.
We've got iHerb for Health, Numeri for World's Hardest Data Science Tournament and Insight Tracker for tracking your biological data.
Choose wisely, my friends.
Also, if you want to work with our team, our amazing team, who are always hiring, could electsfreedman.com slash hiring.
And now onto the full ad reads.
As always, no ads in the middle.
I try to make this interesting, but if you skip them, if you must, my friends, please still check out the sponsors.
I enjoy their stuff.
Maybe you will too.
This show is brought to you by iHerb, a platform, a website, a place.
We can go and get high quality, selected just for you, health and wellness products for great value, inexpensive, affordable.
I get fish oil over there.
It's one of the main supplements I've taken for a long, long, long time in pill form.
Makes me feel like I'm oiling the machine that is the human body and the human mind.
Even just saying that makes me wonder, what is the power of the placebo effect in all of this?
I'm actually a big believer in the power of the human mind, coupled with the effectiveness of medication and supplements and nutrition and diet and exercise, all of it.
If you couple the belief that the thing will work with stuff that actually works, it's like a supercharge.
Because the mind, there's something about the mind allowing the thing to work and maybe the belief that it works reduces stress and has kind of secondary and tertiary effects that you can't even comprehend on the entirety of the biological system that is the human body.
It's so fascinating and it's so difficult to do good studies on that because the whole point is you want

to study the effect of the entirety of the lifestyle and diet decisions you make on the entirety of the human organism, the billions of organisms that make up a single organism that is you.
Anyway, get 22% off with promo code Lex when you go to iherb.com slash Lex.
This show is also brought to you by Numeri, a hedge fund that uses AI and machine learning to make investment decisions, they created a tournament, a challenge for all machine learning gurus to come and to compete against each other to build the best predictive models for financial markets, the stakes are high.
The stakes are high.
This is the kind of problems in the machine learning space that I really care about real world problems with high stakes, not toy problems, not ImageNet.
Now, ImageNet and all that kind of stuff is good for exploring little ideas, the nuances of architecture's training procedures of cool little ideas of the entirety of the pipeline of how to do machine learning or for education purposes.
But if you want to really develop ideas that work in the real world, you should be working on real world data where the stakes are high.
And this is probably one of the hardest tournaments for machine learning in the world.
Head over to Numeri.com slash Lex to sign up for a tournament and hone your machine learning skills.
That's Numeri.com slash Lex for a chance to play against me and win a share of the tournament prize pool.
This show is also brought to you by Insight Tracker, a service I use to track biological data.
As I was saying, the entirety of the biological organism that is you, the individual cells, all the life force that makes up the details of the cells and the proteins and the bloods and the organs and the individual systems that are interconnected to each other in these incredible
complex ways that are all asynchronously communicating with each other through chemistry, through physics, through electrical signals, through mechanical signals, through whatever the hell signals that I can't even find the right words for.
It's an incredible system.
How is it possible that this thing works at all?
I'm looking out into the distance now as I say these words and my visual cortex is processing all of it and somehow makes sense of all of it.
This is incredible.
Anyway, measuring data on this incredible organism is good and companies that allow you to measure it in order to make lifestyle decisions is obviously the future.
That's why I support Insight Tracker.
You can get special savings for a limited time when you go to insight tracker dot com slash Lex.
This is the Lex movement podcast to support it.
Please check out our sponsors in the description and now your friends.
Here's Chris Latner.
It's been, I think, two years since we last talked and in that time, you somehow went and co-created a new programming language called Mojo.
So it's optimized for AI.
It's a super set of Python.
Let's look at the big picture.

What is the vision for Mojo?
For Mojo.
Well, so I mean, I think you have to zoom out.
I've been working on a lot of related technologies for many, many years.
So I've worked on LLVM and a lot of things and mobile and servers and things like this, but the world's changing.
And what's happened with AI is we have new GPUs and new machine learning accelerators and other ASICs and things like that that make AI go real fast.
At Google, I worked on TPUs.
That's one of the biggest, largest scale deployed systems that exist for AI.
And really what you see is if you look across all of the things that are happening in the industry, there's this new compute platform coming.
And it's not just about CPUs or GPUs or TPUs or NPUs or IPUs or whatever, all the PUs, right?
It's about how do we program these things, right?
And so for software folks like us, right, it doesn't do us any good if there's this amazing hardware that we can't use.
And one of the things you find out really quick is that having the theoretical capability of programming something
and then having the world's power and the innovation of all the smart people in the world get unleashed on something can be quite different.
And so really where Mojo came from was starting from a problem of we need to be able to take machine learning,
take the infrastructure underneath it and make it way more accessible, way more usable, way more understandable
by normal people and researchers and other folks that are not themselves like experts in GPUs and things like this.
And then through that journey, we realize, hey, we need syntax for this, we need to do programming language.
So one of the main features of language, I say so fully in jest, is that it allows you to have the file extension to be an emoji or the fire emoji,
which is one of the first emojis used as a file extension I've ever seen in my life.
And then you ask yourself the question, why in the 21st century are we not using Unicode for file extensions?
I mean, it's an epic decision. I think clearly the most important decision in the most part.
But you could also just use mojo as the file extension.
Well, so, okay, so take a step back. I mean, come on, Lex, do you think that the world's ready for this?
This is a big moment in the world, right?
This is where we release this onto the world.
This is innovation.
I mean, it really is kind of brilliant.
Emojis are such a big part of our daily lives. Why is it not in programming?
Well, and like you take a step back and look at what file extensions are, right? They're basically metadata, right?

And so why are we spending all the screen space on them and all this stuff?
Also, you know, you have them stacked up next to text files and PDF files and whatever else.
Like, if you're going to do something cool, you want to stand out, right?
Emojis are colorful. They're visual. They're beautiful, right?
What's been the response so far from...
Is there support on Windows on operating system in displaying file explorer?
Yeah, the one problem I've seen is that Git doesn't escape it, right?
And so it thinks that the fire emoji is unprintable and so it prints out weird hex things if you use the command line Git tool.
But everything else as far as the word works fine.
And I have faith that Git can be improved.
And so Git Hub is fine.
Git Hub is fine. Yeah. Git Hub is fine. Visual Studio Code, Windows, like all this stuff, totally ready.
Because people have internationalization in their normal part of their paths.
So this is just like taking the next step, right?
Somewhere between... Oh, wow, that makes sense. Cool.
I like new things too.
Oh my God, you're killing my baby.
Like, what are you talking about? This can never be...
Like, I can never handle this. How am I going to type this?
Like all these things.
And so this is something where I think that the world will get there.
We don't have to bet the whole farm on this.
I think we can provide both paths.
But I think it'll be great.
When can we have emojis as part of the code? I wonder.
Yeah, so I mean lots of languages provide that.
So I think that we have partial support for that.
It's probably not fully done yet.
But yeah, you can do that.
For example, in Swift, you can do that for sure.
So an example we gave at Apple was the dog cow.
So that's a classical Mac heritage thing.
And so you use the dog and the cow emoji together,
and that could be your variable name.
But of course the internet went and made pile of poop for everything.
So if you want to name your function pile of poop,
then you can totally go to town and see how that gets through code review.
Okay, so let me just ask a bunch of random questions.
So is Mojo primarily designed for AIs,
or is it a general purpose program?
Yeah, good question. So it's AI first.
And so AI is driving a lot of the requirements.
And so Modular is building and designing and driving Mojo forward.

And it's not because it's an interesting project theoretically to build.
It's because we need it.
And so at Modular, we're really tackling the AI infrastructure landscape
and the big problems in AI and the reasons that it is so difficult to use
and scale and adopt and deploy and like all these big problems in AI.
And so we're coming out from that perspective.
Now, when you do that, when you start tackling these problems,
you realize that the solution to these problems isn't actually an AI-specific solution.
And so while we're doing this, we're building Mojo to be a fully general programming language.
And that means that you can obviously tackle GPUs and CPUs and like these AI things,
but it's also a really great way to build NumPy and other things like that.
Or, you know, just if you look at what many Python libraries are today,
they're a layer of Python for the API and they end up being C and C++ code underneath them.
That's very true in AI. That's true in lots of other domains as well.
And so anytime you see this pattern, that's an opportunity for Mojo to help simplify the world
and help people have one thing.
To optimize through simplification by having one thing.
So you mentioned Modular.
Mojo is the programming language. Modular is the whole software stack.
So just over a year ago, we started this company called Modular.
What Modular is about is it's about taking AI and up-leveling it into the next generation.
And so if you take a step back, what's gone on in the last five, six, seven, eight years
is that we've had things like TensorFlow and PyTorch and these other systems come in.
You've used them. You know this.
And what's happened is these things have grown like crazy.
They get tons of users. It's in production deployment scenarios.
It's being used to power so many systems.
AI is all around us now. It used to be controversial years ago, but now it's a thing.
But the challenge with these systems is that they haven't always been thought out with current demands in mind.
And so you think about it, where were LLMs eight years ago?
Well, they didn't exist, right?
AI has changed so much.
And a lot of what people are doing today are very different than when these systems were built.
And meanwhile, the hardware side of this has gotten into a huge mess.
There's tons of new chips and accelerators and every big company's announcing a new chip every day, it feels like.
And so between that, you have this moving system on one side, a moving system on the other side, and it just turns into this gigantic mess, which makes it very difficult for people to actually use AI, particularly in production deployment scenarios.
And so what Modular Students were helping build out that software stack to help solve some of those problems
so that then people can be more productive and get more AI research into production.
Now, what Mojo does is it's a really, really, really important piece of that.

And so that is part of that engine and part of the technology that allows us to solve these problems.
So Mojo is a programming language that allows you to do a high-level programming, the low-level programming.
They do all kinds of programming in that spectrum that gets you closer and closer to the hardware.
So take a step back.
So Lex, what do you love about Python?
Oh, boy. Where do I begin? What is love? What do I love about Python?
You're a guy who knows love. I know this.
Yes. How intuitive it is. How it feels like I'm writing natural language, English.
How when I can not just write but read other people's codes, somehow I can understand it faster.
It's more condensed than other languages, like ones I'm really familiar with, like C++ and C.
There's a bunch of sexy little features.
Yeah.
We'll probably talk about some of them, but list comprehensions and stuff like this.
Also, and don't forget the entire ecosystem of all the packages.
Oh, yeah. That's probably huge.
Because there's always something. If you want to do anything, there's always a package.
Yeah. So it's not just the ecosystem of the packages and the ecosystem of the humans that do it.
That's an interesting dynamic.
That's good.
Because I think something about the usability and the ecosystem makes the thing viral.
It grows and then it's a virtuous cycle, I think.
Well, there's many things that went into that.
So I think that ML was very good for Python.
And so I think that TensorFlow and PyTorch and these systems embracing Python really took and helped Python grow.
But I think that the major thing underlying it is that Python is like the universal connector.
It really helps bring together lots of different systems so you can compose them and build out larger systems without having to understand how it works.
But then what is the problem with Python?
Well, I guess you could say several things, but probably that it's slow.
I think that's usually what people complain about.
And so I mean, other people complain about tabs and spaces versus curly braces or whatever.
But I mean, those people are just wrong because it is actually just better to use an indentation.
Wow, strong words.
So actually, I'm a small tangent.
Let's actually take that.
Let's take all kinds of tangents.
Oh, come on, Lex, you can push me on it.
I can take it.
Design.
Design. Listen, I've recently left Emacs for VS Code and the kind of hate mail I had to receive.
Because on the way to doing that, I also said I've considered Vim and chose not to and went with VS Code.

You're touching on deep religions, right?
Anyway, tabs is an interesting design decision.
And so you've really written a new programming language here.
Yes, it is a superset of Python, but you can make a bunch of different interesting decisions here.
And you chose actually to stick with Python in terms of some of the syntax.
So let me explain why.
I mean, you can explain this in many rational ways.
I think that the indentation is beautiful, but that's not a rational explanation.
But I can defend it rationally.
So first of all, Python 1 has millions of programmers.
It is huge.
It's everywhere.
It owns machine learning.
So factually, it is the thing.
Second of all, if you look at it, C codes, C++ code, Java, whatever, Swift, curly brace languages.
Also run through formatting tools and get indented.
And so if they're not indented correctly, first of all, we'll twist your brain around.
It can lead to bugs.
There's notorious bugs that have happened across time where the indentation was wrong or misleading and it wasn't formatted, right?
And so it turned into an issue, right?
And so what ends up happening in modern large scale code bases is people run automatic formatters.
So now what you end up with is indentation and curly braces.
Well, if you're going to have the notion of grouping, why not have one thing, right?
And get rid of all the clutter and have a more beautiful thing, right?
Also, you look at many of these languages.
It's like, okay, well, you can have curly braces or you can omit them if there's one statement, or you just enter this entire world of complicated design space that objectively you don't need if you have Python-style indentation.
Yeah, I would love to actually see statistics on errors made because of indentation.
Like how many errors are made in Python versus in C++ that have to do with basic formatting, all that kind of stuff.
I would love to see.
I think it's probably pretty minor because once you get, like, you use VS Code, I do too.
So if you get VS Code set up, it does the indentation for you generally, right?
And so you don't, you know, it's actually really nice to not have to fight it.
And then what you can see is the editor is telling you how your code will work by indenting it, which I think is pretty cool.
I honestly don't think I've ever, I don't remember having an error in Python because I intend to stuff wrong.
So, I mean, I think that there's, again, this is a religious thing.
And so I can joke about it.
And I love, I love to kind of, you know, I realize that this is such a polarizing thing and everybody

wants to argue about it.
And so I like poking at the bear a little bit, right?
But frankly, right, come back to the first point, Python 1.
Like, it's huge.
It's an AI.
It's the right thing.
And I think that's, like we see Mojo as being an incredible part of the Python ecosystem.
We're not looking to break Python or change it or quote, unquote, fix it.
We love Python for what it is.
Our view is that Python is just not done yet.
And so if you look at, you know, you mentioned Python being slow.
Well, there's a couple of different things that go into that, which we can talk about if you want.
But one of them is it just doesn't have those features that you would use to do C like programming.
And so if you say, okay, well, I'm forced out of Python into C for certain use cases.
Well, then what we're doing is we're saying, okay, well, why, why is that?
Can we just add those features that are missing from Python back up to Mojo?
And then you can have everything that's great about Python, all the things that you're talking about that you love,
plus not be forced out of it when you do something a little bit more computationally intense or weird or
hardwarey or whatever it is that you're doing.
Well, a million questions.
I want to ask what high level again, is it compiled or is it interpreted language?
So Python is just in time compilation.
What's Mojo?
So Mojo, the complicated answer does all the things.
So it's interpreted, it's just compiled and it's statically compiled.
And so this is for a variety of reasons.
So one of the things that makes Python beautiful is that it's very dynamic.
And because it's dynamic, one of the things they added is that it has this powerful metaprogramming feature.
And so if you look at something like PyTorch or TensorFlow or, I mean, even a simple use case,
like you'd find a class that has the plus method, right?
You can overload the dunder methods like dunder add, for example.
And then the plus method works on your class.
And so it has very nice and very expressive dynamic metaprogramming features.
In Mojo, we want all those features come in.
Like we don't want to break Python, we want it all to work.
But the problem is, is you can't run those super dynamic features on an embedded processor or on a GPU, right?
Or if you could, you probably don't want to just because of the performance.
And so we entered this question of saying, okay, how do you get the power of this dynamic metaprogramming
into a language that has to be super efficient in specific cases?

And so what we did was we said, okay, we'll take that interpreter.
Python has an interpreter in it, right?
Take that interpreter and allow it to run at compile time.
And so now what you get is you get compile time metaprogramming.
And so this is super interesting and super powerful because one of the big advantages you get is you get Python style expressive APIs.
You get the ability to have overloaded operators.
And if you look at what happens inside of like PyTorch, for example, with automatic differentiation and eager mode and like all these things,
they're using these really dynamic and powerful features at runtime.
But we can take those features and lift them so that they run at compile time.
So you're, because C++ has metaprogramming with templates, but it's really messy.
It's super messy. It's always, it was accidentally, I mean, different people have different interpretations.
My interpretation is that it was made accidentally powerful.
It was not designed to be terrain complete, for example, but that was discovered kind of along the way accidentally.
And so there have been a number of languages in the space.
And so they usually have templates or code instantiation, code copying features of various sorts.
Some more modern languages or some more newer languages, let's say, like, you know, they're fairly unknown, like ZIG, for example,
says, okay, well, let's take all of those types that you can run it, all those things you can do at runtime and allow them to happen at compile time.
And so one of the problems with C++, I mean, which is one of one of the problems with C++.
Here we go.
Wrong words.
It's okay.
I mean, everybody hates me for a variety of reasons.
Anyways, I'm sure, right?
I've written enough C++ code to earn a little bit of grumpiness with C++.
But one of the problems with it is that the metaprogramming system templates,
it's just a completely different universe from the normal runtime programming world.
And so if you do metaprogramming and programming, it's just like a different universe, different syntax, different concepts, different stuff going on.
And so, again, one of our goals with Mojo is to make things really easy to use, easy to learn.
And so there's a natural stepping stone.
And so as you do this, you say, okay, well, I have to do programming at runtime.
I have to do programming at compile time.
Why are these different things?
How hard is that to pull it out?
Because that sounds, to me, as a fan of metaprogramming in C++ even, how hard is it to pull that off?
That sounds really, really exciting because you can do the same style programming at compile time and at runtime.

That's really, really exciting.
And so, I mean, in terms of the compiler implementation details, it's hard.
I won't be shy about that.
It's super hard.
It requires, I mean, what Mojo has underneath the covers is a completely new approach to the design of the compiler itself.
And so this builds on these technologies like MLIR that you mentioned,
but it also includes other like caching and other interpreters and jit compilers and other stuff like that.
You have like an interpreter inside the compiler.
Within the compiler, yes.
And so it really takes the standard model of programming languages and kind of twists it and unifies it with the runtime model,
which I think is really cool.
And to me, the value of that is that, again, many of these languages have metaprogramming features.
Like they grow macros or something, right?
You list, right?
Yes.
I know your roots, right?
You know, and this is a powerful thing, right?
And so, you know, if you go back to list, one of the most powerful things about it is that it said that the metaprogramming and the programming are the same, right?
And so that made it way simpler, way more consistent, way easier to understand reason about, and it made it more composable.
So if you build a library, you can use it both at runtime and compile time, which is pretty cool.
Yeah.
And for machine learning, I think metaprogramming, I think we could generally say is extremely useful.
And so you get features, I mean, I'll jump around, but there's the feature of auto tuning and adaptive compilation just blows my mind.
Yeah.
Well, so, okay, so let's come back to that.
All right.
So what is machine learning?
Or what is a machine learning model?
Like you take a PyTorch model off there, right?
It's really interesting to me because what PyTorch and what TensorFlow and all these frameworks are kind of pushing compute into is they're pushing into like this abstract specification of a compute problem, which then gets mapped in a whole bunch of different ways.
Right.
And so this is why it became a metaprogramming problem.
You want to be able to say, cool, I have this neural net, now run with batch size 1000, right?
Do a mapping across batch.

Or, okay, I want to take this problem now running across 1000 CPUs or GPUs, right?
And so like this problem of like describe the compute and then map it and do things and transform it are like actually it's very profound.
And that's one of the things that makes machine learning systems really special.
Maybe can you describe auto tuning and how do you pull off?
I mean, I guess adaptive compilation is what we're talking about as metaprogramming.
Yeah.
How do you pull off auto tuning?
Is that as profound as I think it is?
It seems like a really like, you know, we'll mention list comprehension.
To me, from a quick glance at Mojo, which by the way, I have to absolutely like dive in.
As I realize how amazing this is, I absolutely must dive in.
That looks like just an incredible feature for machine learning people.
Yeah.
Well, so what is auto tuning?
So take a step back.
Auto tuning is a feature in Mojo.
So very, very little of what we're doing is actually research.
Like many of these ideas have existed in other systems and other places.
And so what we're doing is we're pulling together good ideas, remixing them and making them into hopefully a beautiful system.
Right.
And so auto tuning, the observation is that turns out hardware systems, algorithms are really complicated.
Turns out maybe you don't actually want to know how the hardware works.
Right.
A lot of people don't.
Right.
And so there are lots of really smart hardware people.
I know a lot of them where they know everything about, okay, that the cache size is this and the number of registers is that.
And if you use this, what length of vector, it's going to be super efficient because it maps directly onto what it can do.
And like all this kind of stuff for the GPU has SMs and it has a warp size of whatever.
Right.
All the stuff that goes into these things or the title size of a TPU is 128.
Like these, these factoids.
Right.
My belief is that most normal people, and I love hardware people also, I'm not trying to fend literally everybody in the Internet.
But most programmers actually don't want to know this stuff.
Right.
And so if you come at it from perspective of how do we allow people to build both more abstracted, but also more portable code.

Because, you know, could be that the vector length changes or the cache size changes or it could be that the tile size of your matrix changes or the number, you know, an A 100 versus an H 100 versus a Volta versus a whatever GPU have different characteristics.
Right.
A lot of the algorithms that you run are actually the same.
But the parameters, these magic numbers you have to fill in end up being really fiddly numbers that an expert has to go figure out.
And so what auto tuning does it says, okay, well, guess what, there's a lot of compute out there.
Right.
So instead of having humans go randomly try all the things or do a grid search or go search some complicated multidimensional space.
We have computers do that.
Right.
And so auto tuning does is you can say, Hey, here's my algorithm.
It's a matrix operation or something like that.
You can say, Okay, I'm going to carve it up into blocks.
I'm going to do those blocks in parallel.
And I want this this with 128 things that I'm writing on.
I want to cut it this way or that way or whatever.
And you can say, Hey, go see which one's actually empirically better on the system.
And then the result of that you cash for that system.
Yep.
And so come back to twisting your compiler brain.
Right.
So not only does the compiler have an interpreter that she used to do metaprogramming.
That compiler that interpreter that metaprogramming now has to actually take your code and go run it on a target machine.
See, see which one likes the best and then stitch it in and then keep going.
Right.
So part of the compilation is machine specific.
Yeah.
Well, so I mean, this is an optional feature.
Right.
So you don't have to use it for everything.
But yeah, if you're so one of one of the things that we're in the quest of is ultimate performance.
Yes.
Right.
It's important for a couple of reasons.
Right.
So if you're an enterprise, you're looking to save costs and compute and things like this,
ultimate performance translates to, you know, fewer servers.
Like if you care about the environment, hey, better performance leads to more efficiency.
Right.
I mean, you could joke and say like, you know, Python's bad for the environment.

Right.
And so if you've moved to Mojo, it's like at least 10x better just out of the box and keep going.
Right.
But but performance is also interesting because it leads to better products.
Yeah.
So if you're in the space of machine learning, right, if you reduce the latency of a model, so it runs faster.
So every time you query the server running the model, it takes less time.
Well, then the product team can go and make the model bigger.
Well, that's actually makes it so you have a better experience as a customer.
And so a lot of people care about that.
So for auto tuning, for like tile size, you mentioned 120 here for TPU, you would specify like a bunch of options to try.
Yeah.
Just in the code.
Yep.
Just simple statement.
Yep.
Just forget and know, depending where it compiles, it'll actually be the fastest.
And yeah, exactly.
And the beauty of this is that it helps you in a whole bunch of different ways.
Right.
So if you're building, so often what will happen is that, you know, you've written a bunch of software yourself, right?
You, you wake up one day, you say, I have an idea.
I'm going to go code up some code.
I get to work.
I forget about it.
I move on with life.
I come back six months or a year or two years or three years later, you dust it off and you go use it again in a new environment.
And maybe your GPU is different.
Maybe you're running on a server instead of a laptop, maybe whatever.
Right.
And so the problem now is you say, okay, well, I mean, again, not everybody cares about performance,
but if you do, you say, okay, well, I want to take advantage of all these new features.
I don't want to break the old thing though.
Right.
And so the typical way of handling this kind of stuff before is, you know, if you're talking about C++ templates or you're talking about C with macros, you end up with if deaths, you get like all these weird things get layered in, make the code super complicated.
And then how do you test it?
Right.

It becomes this crazy complexity, multi-dimensional space you have to worry about.
And, you know, that just doesn't scale very well.
Actually, let me just jump around before I go to some specific features.
Like the increase in performance here that we're talking about can be just insane.
You write that Moja can provide a 35,000 X speed up over Python.
How does it do that?
So it can even do more, but we'll get to that.
So first of all, when we say that we're talking about what's called C Python, it's the default Python that everybody uses when you type Python three.
That's like typically the one you use, right?
C Python is an interpreter.
And so interpreters, they have an extra layer of like byte codes and things like this that they have to go read, parse, interpret and make some kind of slow from that perspective.
And so one of the first things we do is we move to a compiler.
And so just moving to a compiler, getting the interpreter out of the loop is two to five to 10 X speed up depending on the code.
So just out of the gate, just using more modern techniques, right?
Now, if you do that, one of the things you can do is you can start to look at how C Python started to lay out data.
And so one of the things that C Python did, and this isn't part of the Python spec necessarily, but this is just sets of decisions, is that if you take an integer, for example, it'll put it in an object.
Because in Python, everything's an object.
And so they do the very logical thing of keeping the memory representation of all objects the same.
So all objects have a header, they have like payload data, and what this means is that every time you pass around an object, you're passing around a pointer to the data.
Well, this has overhead.
It turns out that modern computers don't like chasing pointers very much and things like this.
It means that you have to allocate the data.
It means you have to reference count it, which is another way that Python uses to keep track of memory.
And so this has a lot of overhead.
And so if you say, okay, let's try to get that out of the heap, out of a box, out of an indirection, and into the registers.
That's another 10 X.
So it adds up if you're reference counting every single thing you create that adds up.
Yep.
And if you look at, you know, people complain about the Python Gil, this is one of the things that hurts parallelism.
That's because the reference counting.
Right.
And so the Gil and reference counting are very tightly intertwined in Python.
It's not the only thing, but it's very tightly intertwined.
And so then you lean into this and you say, okay, cool.

Well, modern computers, they can do more than one operation at a time.
And so they have vectors.
What is a vector?
Well, a vector allows you to take one, instead of taking one piece of data doing an add or add, and then pick up the next one, you can now do four or eight or 16 or 32 at a time.
Right.
Well, Python doesn't expose that because of reasons.
And so now you can say, okay, well, you can adopt that.
Now you have threads.
Now you have like additional things like you can control memory hierarchy.
And so what Mojo allows you to do is it allows you to start taking advantage of all these powerful things that have been built into the hardware over time.
And it gives the library gives very nice features.
So you can say, just parallelize this, do this in parallel.
Right.
Very powerful weapons against slowness, which is why people have been, I think having fun like just taking code and making it go fast because it's just kind of an adrenaline rush to see like how fast you can get things.
Before I talk about some of the interesting stuff with parallelization, all that, let's first talk about like the basics.
We talked the indentation.
Right.
So this thing looks like Python.
It's sexy and beautiful like Python, as I mentioned.
Is it a typed language?
So what's the role of types?
So Python has types.
It has strings, it has integers, it has dictionaries and like all that stuff, but they all live at runtime.
Right.
And so because all those types live at runtime in Python, you never, you don't have to spell them.
Python also has like this whole typing thing going on now and a lot of people use it.
I'm not talking about that.
That's kind of a different thing.
We can go back to that if you want.
But typically the, you know, you just say I take, I have a def and my def takes two parameters.
I'm going to call them A and B and I don't have to write a type.
Okay.
So that is great.
But what that does is that forces what's called a consistent representation.
So these things have to be a pointer to an object with the object header and they all have to look the same.
And then when you dispatch a method, you go through all the same different paths no matter

what the receiver or whatever that type is.
So what Mojo does is it allows you to have more than one kind of type.
And so what it does is allows you to say, okay, cool.
I have an object.
And objects behave like Python does.
And so it's fully dynamic and that's all great.
And for many things classes, like that's all very powerful and very important.
But if you want to say, hey, it's an integer and it's 32 bits or 64 bits or whatever it is, or it's a floating point value, it's 64 bits.
Well, then the compiler can take that and it can use that to do way better optimization.
And it turns out again, getting rid of the interactions, that's huge.
It means you can get better code completion because you have, because compiler knows what the type is.
But operations work on it.
And so that's actually pretty huge.
And so what Mojo does is it allows you to progressively adopt types into your program.
And so you can start again, it's compatible with Python.
And so then you can add however many types you want, wherever you want them.
And if you don't want to deal with it, you don't have to deal with it.
Right.
And so one of, one of, you know, our opinions on this is it's not that types are the right thing or the wrong thing.
It's that they're a useful thing.
Well, so it's kind of optional.
It's not strict typing.
You don't have to specify type.
Exactly.
Okay.
So it's starting from the thing that Python is kind of reaching towards right now with trying to inject types into it.
Yeah, with a very different approach, but yes.
Yeah.
What's the different approach?
I'm actually one of the people that have not been using types very much in Python.
That's okay.
Why did you say?
It just, well, because I know the importance it's like adults use strict typing.
And so I refuse to grow up in that sense.
It's a kind of rebellion, but I just know that it probably reduces the amount of errors even just for forget about performance improvements.
It probably reduces errors when you do strict typing.
Yeah.
So I mean, I think it's interesting if you look at that, right?
And the reason I'm giving a hard time is that there's this, this cultural norm, this

pressure, this like, there has to be a right way to do things like, you know, grownups only do it one way.
And if you don't do that, you should feel bad.
Yes.
Right.
Like some people feel like Python's a guilty pleasure or something.
And that's like, when it gets serious, I need to go rewrite it.
Right.
Yeah.
Exactly.
Well, I mean, cool.
I understand history and I understand kind of where this comes from, but I don't think it has to be a guilty pleasure.
Yeah.
Right.
And so if you look at that, you say, why do you have to rewrite it?
Well, you have to rewrite it to deploy.
Well, why do you want to deploy?
Well, you care about performance or you care about productivity or you want, you know, a tiny thing on the server that has no dependencies or, you know, you have objectives that you're trying to attain.
So what if Python can achieve those objectives?
So if you want types, well, maybe you want types because you want to make sure you're passing the right thing.
Sure.
You can add a type.
If you don't care, you're prototyping some stuff, you're hacking some things out, you're like pulling some RAM code off the internet.
It should just work.
Right.
And you shouldn't be like pressured.
You shouldn't feel bad about doing the right thing or the thing that feels good.
Now, if you're in a team, right, you're working at some massive internet company and you have 400 million lines of Python code.
Well, they may have a house rule that you use types.
Yeah.
Right.
Because it makes it easier for different humans to talk to each other and understand what's going on and bugs at scale.
Right.
And so there are lots of good reasons why you might want to use types.
But that doesn't mean that everybody should use them all the time.
Right.
What it does is it says, cool, well, allow people to use types.

And if you use types, you get nice things out of it.
Right.
You get better performance and things like this.
Right.
But Mojo is a full compatible superset of Python.
Right.
And so that means it has to work without types.
It has to support all the dynamic things.
It has to support all the packages.
It has to support for comprehension, list comprehensions and things like this.
Right.
And so that starting point I think is really important.
I think that, again, you can look at why I care so much about this.
And there's many different aspects of that.
One of which is the world went through a very challenging migration from Python 2 to Python 3.
Right.
And this migration took many years.
And it was very painful for many teams.
Right.
And there's a lot of things that went on in that.
I'm not an expert in all the details.
I honestly don't want to be.
I don't want the world to have to go through that.
Right.
And people can ignore Mojo and if it's not their thing, that's cool.
But if they want to use Mojo, I don't want them to have to rewrite all their code.
Yeah.
I mean, this, okay.
The superset part is just, I mean, there's so much brilliant stuff here that definitely is incredible.
We'll talk about that.
Yeah.
First of all, how's the typing implemented differently in Python versus Mojo?
Yeah.
So this heterogeneous flexibility you said is definitely implemented.
Yeah.
So I'm not a full expert in the whole backstory on types in Python.
I'll give you that.
I can give you my understanding.
My understanding is basically like many dynamic languages.
The ecosystem went through a phase where people went from writing scripts to writing a large scale, huge code bases in Python.
And at scale, it kind of helps have types.

Yeah.
People want to be able to reason about interfaces.
What do you expect to string or an int or like these basic things, right?
And so what the Python community started doing is it started saying, okay, let's have tools on the side, checker tools, right?
The go and like enforce and variance, check for bugs, try to identify things.
These are called static analysis tools generally.
And so these tools run over your code and try to look for bugs.
What ended up happening is there's so many of these things, so many different weird patterns and different approaches on specifying the types and different things going on that the Python community realized and recognized, hey, hey, hey, there's a thing here.
And so what they started to do is they started to standardize the syntax for adding types to Python.
Now, one of the challenges that they had is that they're coming from kind of this fragmented world where there's lots of different tools, they have different tradeoffs and interpretations and the types mean different things.
And so if you look at types in Python, according to the Python spec, the types are ignored.
Right.
So according to the Python spec, you can write pretty much anything in a type position.
Okay.
And technically, you can write any expression.
Okay.
So that's beautiful because you can extend it.
You can do cool things.
You can write, build your own tools.
You can build your own house, linter or something like that.
Right.
But it's also a problem because any existing Python program may be using different tools and they have different interpretations.
And so if you adopt somebody's package into your ecosystem, try to run the tool you prefer, it may throw out tons of weird errors and warnings and problems just because it's incompatible with how these things work.
Also, because they're added late and they're not checked by the Python interpreter, it's always kind of more of a hint than it is a requirement.
Also, the CPython implementation can't use them for performance.
And so it's really...
Well, you have the big one, right?
So you can't utilize the... for the compilation, for the just intact compilation.
Okay.
Exactly.
And this all comes back to the design principle of it's...
It's kind of...
They're kind of hints.
They're kind of...

The definition is a little bit murky.
It's unclear exactly the interpretation in a bunch of cases.
And so because of that, you can't actually...
Even if you want to, it's really difficult to use them to say, like, it is going to be an int and if it's not, it's a problem, right?
A lot of code would break if you did that.
So in Mojo, right?
So you can still use those kind of type annotations.
It's fine.
But in Mojo, if you declare a type and you use it, then it means it is going to be that type.
And the compiler helps you check that and enforce it and it's safe.
And it's not a best effort hint kind of a thing.
So if you try to shove a string type thing into an integer...
You get an error.
From the compiler.
Compiler, compile time.
Nice.
Okay.
What kind of basic types are there?
Yeah.
So Mojo is pretty hardcore in terms of what it tries to do in the language, which is the philosophy there is that we...
Again, if you look at Python, right?
Python is a beautiful language because it's so extensible, right?
And so all of the different things in Python like for loops and plus and like all these can be accessed through these underbar and bar methods.
Okay.
So you have to say, okay, if I make something that is super fast, I can go all the way down to the metal.
Why do I need to have integers built into the language?
And so what Mojo does is it says, okay, well, we can have this notion of structs.
So you have classes in Python.
Now you can have structs.
Classes are dynamic.
Structs are static.
Cool.
We can get high performance.
We can write C++ kind of code with structs if you want.
These things mix and work beautifully together.
But what that means is that you can go and implement strings and ints and floats and arrays and all that kind of stuff in the language, right?
And so that's really cool because, you know, to me as a idealizing compiler language type of person, what I want to do is I want to get magic out of the compiler and put it in

the libraries because if somebody can, you know, if we can build an integer that's beautiful and has an amazing API and does all the things you'd expect an integer to do, but you don't like it, maybe you want a big integer, maybe you want, like, sideways integer, I don't know, like what all the space of integers are, then you can do that and it's not a second-class citizen.
And so if you look at certain other languages like C++, one I also love and use a lot, ints, hard code in the language, but complex is not.
And so it's kind of weird that, you know, you have this STD complex class, but you have int and complex tries to look like a natural numeric type and things like this, but integers and floating point have these, like, special promotion rules and other things like that that are magic and they're hacked into the compiler.
And because of that, you can't actually make something that works like the built-in types.
Is there something provided as a standard because, you know, because it's AI first, you know, numerical types are so important here.
So is there something like a nice standard implementation of integer and float?
Yeah.
So we're still building all that stuff out.
So we provide integers and floats and all that kind of stuff.
We also provide, like, buffers and tensors and things like that that you'd expect in an ML context.
Honestly, we need to keep designing and redesigning and working with the community to build that out and make that better.
That's not our strength right now.
Give us six months or a year and I think it'll be way better.
But the power of putting in the library means that we can have teams of experts that aren't compiler engineers that can help us design and refine and drive us forward.
So one of the exciting things we should mention here is that this is new and fresh.
This cake is unbaked.
It's almost baked.
You can tell it's delicious, but it's not fully ready to be consumed.
Yep.
That's very fair.
It is very useful, but it's very useful if you're a super low-level programmer right now and what we're doing is we're working our way up the stack.
And so the way I would look at Mojo today in May and 2023 is that it's like a 0.1.
So I think that, you know, a year from now, it's going to be way more interesting to a variety of people.
But what we're doing is we decided to release it early so that people can get access to and play with them.
We can build it with the community.
We have a big roadmap fully published, being transparent about this, and a lot of people involved in the stuff.
And so what we're doing is we're really optimizing for building the thing the right way.
And building it the right way is kind of interesting working with the community because everybody

wants it yesterday.
And so sometimes it's kind of, you know, there's some dynamics there.
But I think it's the right thing.
So there's a discord also.
So the dynamics is pretty interesting.
Sometimes the community probably can be very chaotic and introduce a lot of stress.
Guido famously quit over the stress of the Walrus operator.
I mean, it's, you know, a broke, maybe that would be exactly.
And so like it could be very stressful to develop.
Can you just add tangent upon a tangent?
Is it stressful to work through the design of various features here, given that the community is so recently involved?
Well, so I've been doing open development and community stuff for decades now.
Somehow this has happened to me.
So I've learned some tricks, but the thing that always gets me is I want to make people happy.
Right.
And so this is maybe not all people all happy all the time, but generally I want people to be happy.
Right.
And so the challenge is that, again, we're tapping into some long, some deep seated long tensions and pressures, both in the Python world, but also in the AI world and the hardware world and things like this.
And so people just want us to move faster.
Right.
So again, our decision was, let's release this early.
Let's get people used to it or access to it and play with it.
And like, let's build in the open, which we could have, you know, had the language monk sitting in the cloister up on the hilltop, like be varying away trying to build something.
But in my experience, you get something that's way better if you work with the community.
Right.
And so yes, it can be frustrating, can be challenging for lots of people involved.
And, you know, if you, I mean, if you mentioned our Discord, we have over 10,000 people on the Discord, 11,000 people or something.
Keep in mind, we released Mojo like two weeks ago.
Yeah.
So.
It's very active.
So it's very cool.
But what that means is that, you know, 10, 11,000 people all will want something different.
Right.
And so what we've done is we've tried to say, okay, cool, here's our roadmap here and the roadmap isn't completely arbitrary.
It's based on here's the logical order in which to build these features or add these

capabilities and things like that.
And what we've done is we spun really fast on like bug fixes.
And so we actually have very few bugs, which is cool.
I mean, actually for a project in the state, but then what we're doing is we're dropping in features very deliberately.
I mean, this is fun to watch because you got the two gigantic communities of like hardware, like systems engineers.
And then you have the machine learning Python people that are like higher level.
Yeah.
And it's just too like army like they've been at war.
Yeah.
They've been at war.
Right.
And so here's a Tolkien novel or something.
Okay.
So here's the test again, like it's super funny for something that's only been out for two weeks.
Right.
People are so impatient.
Right.
But okay, cool.
Let's fast forward a year.
Like any year's time module will be actually quite amazing and solve tons of problems and be very good.
People still have these problems.
Right.
And so you look at this and you say, and the way I look at this at least is to say, okay, well, we're solving big longstanding problems.
To me, I again, working on many different problems, I want to make sure we do it right.
Right.
There's like a responsibility you feel because if you mess it up, right, there's very few opportunities to do projects like this and have them really have impact on the world.
If we do it right, then maybe we can take those feuding armies and actually heal some of those wounds.
Yeah.
Right.
There's like a speech by George Washington or Abraham Lincoln or something.
And you look at this and it's like, okay, well, how different are we?
Yeah.
We all want beautiful things.
We all want something that's nice.
We all want to be able to work together.
We all want our stuff to be used.
Right.

And so if we can help heal that.
Now, I'm not optimistic that all people will use Mojo and they'll stop using C++.
Like that's not my goal.
Right.
But if we can heal some of that, I think that'd be pretty cool.
Yeah.
And we start by putting the people who like braces into the gulag.
No.
So there are proposals for adding braces to Mojo and we just tell them, we tell them no.
Okay.
Politely.
Yeah.
Anyway, so there's a lot of amazing features on the roadmap and those ready to implement that.
It'd be awesome.
I could just ask you a few things.
Yeah, go for it.
So the other performance improvement comes from immutability.
So what's this var and this let thing that we got going on?
What's immutability?
Yeah.
So one of the things that is useful and it's not always required, but it's useful is knowing whether something can change out from underneath you.
Right.
And so in Python, you have a pointer to an array.
Right.
And so you pass that pointer to an array around to things.
If you pass into a function, maybe take that and scroll away in some other data structure.
And so you get your array back and you go to use it.
Now somebody else is like putting stuff in your array.
How do you reason about that?
Because to be very complicated, at least a lot of bugs, right?
And so one of the things that, you know, again, this is not something Mojo forces on you, but something that Mojo enables is a thing called value semantics.
And what value semantics do is they take collections like arrays, like dictionaries, also tensors and strings and things like this that are much higher level and make them behave like proper values.
And so it makes it look like if you pass these things around, you get a logical copy of all the data.
And so if I pass you an array, it's your array.
You can go do what you want to it.
You're not going to hurt my array.
Now that is an interesting and very powerful design principle.

It defines a way of ton of bugs.
You have to be careful to implement it in an efficient way.
Yeah, is there a performance hit that's significant?
Generally not, if you implement it the right way.
But it requires a lot of very low level getting the language right bits.
I assume there'll be a huge performance hit because the benefit is really nice because you don't get into these conflicts.
Absolutely.
Well, the trick is you can't do copies.
So you have to provide the behavior of copying without doing the copy.
Yeah.
How do you do that?
How do you do that?
It's not magic.
It's actually pretty cool.
Well, so first, before we talk about how that works, let's talk about how it works in Python.
So in Python, you define a person class or maybe a person class is a bad idea.
You define a database class and database class has an array of records, something like that.
And so the problem is that if you pass in a record or a class instance into the database, it'll take a hold of that object and then it assumes it has it.
And if you're passing an object in, you have to know that that database is going to take it and therefore you shouldn't change it after you put in the database.
You just kind of have to know that.
You just have to kind of know that.
And so you roll out version one of the database.
You just kind of have to know that.
Of course, Lex uses his own database, right?
Yeah.
Right?
Because you built it.
You understand how this works, right?
Somebody else joins the team.
They don't know this.
Yes.
Right?
And so now they suddenly get bugs.
You're having to maintain the database.
You shake your fist.
You argue the 10th time this happens, you're like, okay, we have to do something different.
Right?
And so what you do is you go change your Python code and you change your database class to copy the record every time you add it.
And so what ends up happening is you say, okay, I will do what's called a defensive copy inside the database.

And then that way, if somebody passes something in, I will have my own copy of it and they can go do whatever and they're not going to break my thing.
Okay.
This is usually the two design patterns.
If you look in PyTorch, for example, this is cloning a tensor.
Like there's a specific thing and you have to know where to call it.
And if you don't call in the right place, you get these bugs and this is state-of-the-art, right?
So a different approach.
So it's used in many languages.
So I worked with it in Swift.
As you say, okay, well, let's provide value semantics.
And so we want to provide the view that you get a logically independent copy.
But we want to do that lazily.
And so what we do is you say, okay, if you pass something into a function, it doesn't actually make a copy.
What it actually does is it just increments the reference to it.
And if you pass it around, you stick in your database, you can go into the database, you own it.
And then you come back out of the stack, nobody's copied anything.
You come back out of the stack and then the caller lets go of it.
Well, then you've just handed it off to the database, you've transferred it, and there's no copies made.
Now, on the other hand, if your coworker goes and hands you a record and you pass it in, you stick it in the database, and then you go to town and you start modifying it, what happens is you get a copy lazily on demand.
And so what this does is it gives you copies only when you need them.
And so it defines the way the bugs, but it also generally reduces the number of copies in practice.
But the implementation details are tricky here.
Yes.
So this is, yes.
Something with reference counting, but to make it performant across a number of different kinds of objects.
Yeah.
So you need a couple of things.
So this concept has existed in many different worlds, and so again, it's not novel research at all.
Right?
Because getting the design right so that you can do this in a reasonable way, right?
And so there's a number of components that go into this.
One is when you're passing around, so we're talking about Python and reference counting and the expense of doing that.
When you're passing values around, you don't want to do extra reference counting for no

good reason.
And so you have to make sure that you're efficient and you transfer ownership instead of duplicating references and things like that, which is a very low level problem.
You also have to adopt this and you have to build these data structures.
And so if you say Mojo has to be compatible with Python, so of course the default list is a reference semantic list that works the way you'd expect in Python, but then you have to design a value semantic list.
And so you just have to implement that and then you implement the logic within.
And so the role of the language here is to provide all the low level hooks that allow the author of the type to be able to get and express this behavior without forcing it into all cases or hard coding this into the language itself.
There's ownership, so you're constantly tracking who owns the thing.
Yes, and so there's a whole system called ownership, and so this is related to work done in the Rust community.
Also the Swift community has done a bunch of work and there's a bunch of different other languages that involve kind of C++ actually has copy constructors and destructors and things like that.
And C++ has everything, so it has move constructors and it has this whole world of things.
And so this is a body of work that's kind of been developing for many, many years now, and so Mojo takes some of the best ideas out of all these systems and remixes it in a nice way so that you get the power of something like the Rust programming language, but you don't have to deal with it when you don't want to, which is a major thing in terms of teaching and learning and being able to use and scale these systems.
How does that play with argument conventions?
What are they?
Why are they important?
How does the value semantics?
How does the transfer ownership work with the arguments when they're passed into functions?
So if you go deep into systems programming land, so this isn't, again, this is not something for everybody, but if you go deep into systems programming land, what you encounter is you encounter these types that get weird.
So if you're used to Python, you think about everything, I can just copy it around, I can go change it and mutate it and do these things and it's all cool.
If you get into systems programming land, you get into these things like I have an atomic number or I have a mutex or I have a uniquely owned database handle, things like this, right?
So these types, you can't necessarily copy.
Sometimes you can't necessarily even move them to a different address.
And so what Mojo allows you to do is it allows you to express, hey, I don't want to get a copy of this thing, I want to actually just get a reference to it.
And by doing that, well, you can say, as you can say, okay, if I'm defining something weird, like an atomic number or something, it has to be, so an atomic number is an area in memory that multiple threads can access at a time without locks, right?
And so the definition of atomic numbers, multiple different things have to be poking it, therefore they have to agree on where it is.

And so you can't just move it out from underneath one because it kind of breaks what it means.
And so that's an example of a type that you can't copy, you can't move.
Once you create it, it has to be where it was, right?
Now if you look at many other examples, like a database handle, right?
So okay, well, what happens, how do you copy a database handle?
Do you copy the whole database?
That's not something you necessarily want to do.
There's a lot of types like that where you want to be able to say that they are uniquely owned.
So there's always one of this thing.
And or if I create a thing, I don't copy it.
And so what Mojo allows you to do is it allows you to say, hey, I want to pass around a reference to this thing without copying it.
And so it has borrowed conventions, so you can say you can use it, but you don't get to change it.
You can pass it by mutable reference.
And so if you do that, then you can, you get a reference to it, but you can change it.
And so it manages all that kind of stuff.
So it's just a really nice implementation of, like C++ has, you know, the different kinds of pointers.
Yeah.
Smart, smart.
Smart pointers that you can explicitly define this allows you.
But you're saying that's more like the weird case versus the common case.
Well, it depends on where, I mean, I mean, I don't think I'm a normal person.
So I mean, I've got one to call other people weird, but the, but you know, if you talk to a normal Python, a typical Python programmer, you're typically not thinking about this, right?
This is a lower level of abstraction.
Now, if you talk to a C++ programmer, certainly if you talk to a Rust programmer, again, they're not weird.
They're delightful.
Like these are all good people, right?
Those folks will think about all the time, right?
And so I look at this as, there's a spectrum between very deep low level systems.
I'm going to go poke the bits and care about how they're laid out in memory all the way up to application and scripting and other things like this.
And so it's not that anybody's right or wrong.
It's about how do we build one system that scales?
By the way, the idea of an atomic number has been something that always brought me deep, happiness, because the flip side of that, the idea that threads can just modify stuff asynchronously, just the whole idea of concurrent programming is a source of infinite stress for me.
Well, so this is where you jump into, you know, again, you zoom out and get out of program

languages or compilers and you just look at what the industry has done.
My mind is constantly blown by this, right?
And you look at what, you know, Moore's law, Moore's law has this idea that like computers for a long time, single thread performance just got faster and faster and faster and faster for free.
But then physics and other things intervened and power consumption, like other things started to matter.
And so what ended up happening is we went from single core computers to multicore, then we went to accelerators, right?
And this trend towards specialization of hardware is only going to continue.
And so for years, us programming language nerds and compiler people have been saying, okay, well, how do we tackle multicore, right?
For a while it was like, multicore is the future, we have to get on top of this thing.
And then it was multicores to default.
What are we doing with this thing?
And then it's like, there's chips with hundreds of cores in them, what will happen, right?
And so I'm super inspired by the fact that, you know, in the face of this, you know, those machine learning people invented this idea of a tensor, right, and what is a tensor?
A tensor isn't like an arithmetic and algebraic concept.
It's like an abstraction around a gigantic, parallelizable data set, right?
And because of that, and because of things like TensorFlow and PyTorch, we're able to say, okay, we'll express the math of the system.
This enables you to do automatic differentiations, enables you to do like all these cool things.
And it's an abstract representation.
Because you have that abstract representation, you can now map it onto these parallel machines without having to control, okay, put that right here, put that right there, put that right there.
And this has enabled an explosion in terms of AI, compute, accelerators, like all the stuff.
And so that's super, super exciting.
What about the deployment, the execution across multiple machines?
So you write that the modular compute platform dynamically partitions models with billions of parameters and distributes their execution across multiple machines, enabling unparalleled efficiency.
By the way, the use of unparalleled in that sentence, anyway, enabling unparalleled efficiency scale and reliability for the largest workloads.
So how do you do this abstraction of distributed deployment of large models?
Yeah, so one of the really interesting tensions, so there's a whole bunch of stuff that goes into that.
I'll pick a random walkthrough.
If you go back and replay the history of machine learning, the brief, the most recent history of machine learning, because this is, as you know, very deep, I knew Lex when he had an AI podcast.
So if you look at just TensorFlow and PyTorch, which is pretty recent history in the big picture,

but TensorFlow is all about graphs, PyTorch, I think, pretty unarguably ended up winning and why did it win, mostly because of usability, and the usability of PyTorch is, I think, huge.
And I think, again, that's a huge testament to the power of taking abstract theoretical, technical concepts and bring it to the masses.
Now the challenge with what the TensorFlow versus the PyTorch design points was that TensorFlow is kind of difficult to use for researchers, but it was actually pretty good for deployment.
PyTorch is really good for researchers, it's not super great for deployment.
And so I think that we as an industry have been struggling, and if you look at what deploying a machine learning model today means is that you'll have researchers who are, I mean, wicked smart, of course, but they're wicked smart at model architecture and data and calculus and they're wicked smart in various domains, they don't want to know anything about the hardware or deployment or C++ or things like this.
And so what's happened is you get people who train the model, they throw it over the fence and then you have people that try to deploy the model.
Well, every time you have a team A does X, they throw it over the fence and team B does Y, you have a problem because, of course, it never works the first time.
And so you throw it over the fence, they figure out, OK, it's too slow, it won't fit, doesn't use the right operator, the tool crashes, whatever the problem is, then they have to throw it back over the fence.
And every time you throw a thing over a fence, it takes three weeks of project managers and meetings and things like this.
And so what we've seen today is that getting models in production can take weeks or months. It's not atypical.
I talked to lots of people and you talk about VP of software at some internet company trying to deploy a model and they're like, why do I need a team of 45 people?
It's so easy to train a model, why can't I deploy it?
And if you dig into this, every layer is problematic.
So if you look at the language piece, I mean, this is tip of the iceberg, it's a very exciting tip of the iceberg for folks, but you've got Python on one side and C++ on the other side.
Python doesn't really deploy, I mean, it can theoretically, technically in some cases, but often a lot of production teams will want to get things out of Python because they get better performance and control and whatever else, so Mojo can help with that.
If you look at serving, so you talk about gigantic models, well, a gigantic model won't fit on one machine, right?
And so now you have this model, it's written in Python, it has to be rewritten in C++.
Now it also has to be carved up so that half of it runs on one machine, half of it runs on another machine, or maybe it runs on 10 machines.
Well, so now suddenly the complexity is exploding, right?
And the reason for this is that if you look into TensorFlow PyTorps, these systems, they weren't really designed for this world, right?
They were designed for, you know, back in the day when we were starting and doing things where it was a different, much simpler world, like you want to run ResNet 50 or some ancient

model architecture like this, it was just a completely different world.
Trained on 1GPU, doing for some 1GPU.
Yeah, AlexNet, right, in the major breakthrough.
And the world has changed, right?
And so now the challenge is that TensorFlow PyTorps, these systems, they weren't actually designed for LLMs, like that was not a thing.
And so where TensorFlow actually has amazing power in terms of scale and deployment and things like that, and I think Google is, I mean, maybe not unmatched, but they're like incredible in terms of their capabilities and gigantic scale.
Many researchers using PyTorps, right?
And so PyTorps doesn't have those same capabilities, and so what Modular can do is it can help with that.
Now if you take a step back and you say like, what is Modular doing, right?
So Modular has like a bitter enemy that we're fighting against in the industry, and it's one of these things where everybody knows it, but nobody is usually willing to talk about it.
The bitter enemy.
The bitter thing that we have to destroy, that we're all struggling with, and it's like fish can't see water, it's complexity.
Sure, yes, it's complexity.
Right.
That was very close, I'll tell you, I was very excited.
And so if you look at it, yes, it is on the hardware side, yes, all these accelerators, all these software stacks that go with the accelerator, all these, like there's massive complexity over there.
You look at what's happening on the modeling side, massive amount of complexity, like things are changing all the time.
People are inventing, turns out the research is not done, right?
And so people want to be able to move fast.
Transformers are amazing, but there's a diversity even within transformers, and what's the next transformer, right?
And you look into serving, also huge amounts of complexity, it turns out that all the cloud providers have all their very weird, but very cool hardware for networking and all this kind of stuff, and it's all very complicated, people aren't using that.
You look at classical serving, right?
There's this whole world of people who know how to write high-performance servers with zero-copy networking and like all this fancy, asynchronous IO and like all these fancy things in the serving community, very little that has pervaded into the machine learning world, right?
And why is that?
Well, it's because, again, these systems have been built up over many years, they haven't been rethought.
There hasn't been a first principles approach to this, and so what Modular's doing is we're saying, okay, we've built many of these things, like so I've worked on TensorFlow and TPUs

and things like that, other folks on our team have like worked on PyTorch core, we've worked on Onyx one time, we've worked on many of these other systems, and so built systems like the Apple accelerators and all that kind of stuff, like our team is quite amazing.
And so one of the things that roughly everybody at Modular's grumpy about is that when you're working on one of these projects, you have the first-order goal, get the hardware to work, get the system to enable one more model, get this product out the door, enable the specific workload or solve this problem for this product team, right?
And nobody's been given a chance to actually do that step back.
And so we as an industry, we didn't take two steps forward, we took like 18 steps forward in terms of all this really cool technology across compilers and systems and runtimes and heterogeneous computing, like all this kind of stuff, and like all this technology has been, you know, I wouldn't say beautifully designed, but it's been proven in different quadrants.
Like, you know, you look at Google with TPUs, massive, huge exoflops of compute strapped together into machines that researchers are programming in Python in a notebook.
That's huge.
That's amazing.
That's incredible.
Right?
It's incredible.
And so you look at the technology that goes into that, and the algorithms are actually quite general.
And so lots of other hardware out there, and lots of other teams out there don't have the sophistication or maybe the years working on it or the budget or whatever that Google does, right?
And so they should be getting access to the same algorithms, but they just don't have that, right?
So what we're just doing is we're saying, cool, this is not research anymore, right?
We've built auto-tuning in many systems.
We've built programming languages, right?
And so like have, you know, implemented C++, have implemented Swift, have implemented many of these things.
And so, you know, it's hard, but it's not research.
And you look at accelerators, well, we know there's a bunch of different weird kind of accelerators, but they actually cluster together, right?
And you look at GPUs.
Well, there's a couple of major vendors of GPUs, and they maybe don't always get along, but their architectures are very similar.
You look at CPUs.
CPUs are still super important for the deployment side of things, and you see new architectures coming out from all the cloud providers and things like this, and they're all super important to the world, right?
But they don't have the 30 years of development that the entrenched people do, right?
And so what modular can do is we're saying, okay, all this complexity, like it's not

bad complexity.
It's actually innovation, right?
And so it's innovation that's happening, and it's for good reasons.
But I have sympathy for the poor software people, right?
I mean, again, I'm a generally a software person too, I love hardware.
But software people want to build applications and products and solutions that scale over many years.
They don't want to build a solution for one generation of hardware with one vendor's tools, right?
And because of this, they need something that scales with them.
They need something that works on cloud and mobile, right?
Because their product manager said, hey, I want to have lower latency and it's better for personalization or whatever they decide, right?
Products evolve.
And so the challenge with the machine learning technology and the infrastructure that we have today in the industry is that it's all these point solutions.
And because they're all these point solutions, it means that as your product evolves, you have to like switch different technology stacks or switch to different vendor.
And what that does is that slows down progress.
So basically, a lot of the things we've developed in those little silos for machine learning tasks, you want to make that the first class citizen of a general purpose programming language that can then be compiled across all these kinds of hardware.
Well, so it's not really about a programming language.
I mean, the programming language is a component of the mission, right?
And the mission is not literal, but our joking mission is to save the world from terrible AI software.
Excellent.
Okay.
So, you know, if you look at this mission, you need a syntax.
So yes, you need a programming language, right?
And we wouldn't have to build the programming language if one existed, right?
So if Python was already good enough, then cool, we would just use it, right?
We're not just doing very large scale, expensive engineering projects for the sake of it.
It's to solve a problem, right?
It's also about accelerators.
It's also about exotic numerics and B-flit 16 and matrix multiplications and convolutions and like this kind of stuff.
Within the stack, there are things like kernel fusion.
It's an esoteric, but really important thing that leads to much better performance and much more general research hackability together, right?
And that's enabled by the ASICs.
That's enabled by certain hardware.
So it's like, where's the dance between, I mean, there's several questions here, like how do you add a piece of hardware to this stack?

If I have this genius invention of a specialized accelerator, how do I add that to the modular framework?
And also, how does modular as a standard start to define the kind of hardware that should be developed?
Yeah.
So let me take a step back and talk about status quo.
And so if you go back to TensorFlow 1, PyTorch 1, this kind of timeframe, and these have all evolved and gotten way more complicated.
So let's go back to the glorious simple days, right?
These things basically were CPUs and CUDA.
And so what you do is you say, go do a dense layer, and a dense layer has a matrix multiplication at it, right?
And so when you say that, you say, go do this big operation, a matrix multiplication.
And if it's on a GPU, kick off CUDA kernel, if it's on a CPU, go do like an Intel algorithm or something like that with the Intel MKL, okay?
Now that's really cool if you're either video or Intel, right?
But then more hardware comes in, right?
And on one axis, you have more hardware coming in.
On the other hand, you have an explosion of innovation in AI.
And so what happened with both TensorFlow and PyTorch is that the explosion of innovation in AI has led to, it's not just about matrix multiplication and convolution, these things have now like 2,000 different operators.
And on the other hand, you have, I don't know how many pieces of hardware there are there, it's a lot.
It's not even hundreds, it's probably thousands, okay?
And across all of Edge and across all the different things that exist.
Did I use that scale?
Yeah, exactly.
I mean, yeah.
Also, it's not just like...
Yeah, it's everywhere, yeah.
No, it's not a handful of TPU alternatives.
Correct.
It's every phone, often with many different chips inside of it from different vendors.
Right.
Like, AI is everywhere, it's a thing, right?
Why are they all making their own chips?
Like, why is everybody making their own thing?
Well, so because...
Was that a good thing, Priscilla?
So Chris's philosophy on hardware, right?
So my philosophy is that there isn't one right solution, right?
And so I think that, again, we're at the end of Moore's Law, specialization happens.
If you're building...

If you're training GPT-5, you want some crazy supercomputer data center thingy.
If you're making a smart camera that runs on batteries, you want something that looks very different.
If you're building a phone, you want something that looks very different.
If you have something like a laptop, you want something that looks maybe similar, but a different scale, right?
And so AI ends up touching all of our lives, robotics, right?
And like lots of different things.
And so as you look into this, these have different power envelopes.
There's different trade-offs in terms of the algorithms.
There's new innovations in sparsity and other data formats and things like that.
And so hardware innovation, I think, is a really good thing, right?
And what I'm interested in is unlocking that innovation.
There's also like analog and quantum and like all the really weird stuff, right?
And so if somebody can come up with a chip that uses analog computing and it's 100x more power efficient, I think what that would mean in terms of the daily impact on the products we use.
That would be huge.
Now, if you're building an analog computer, you may not be a compiler specialist, right?
These are different skill sets, right?
And so you can hire some compiler people if you're running a big company, maybe.
But it turns out these are really like exotic new generation of compilers.
Like this is a different thing, right?
And so if you take a step back out and come back to what is the status quo, the status quo is that if you're intel or you're in video, you continue to keep up with the industry and you chase and okay, there's 1900 now, there's 2000 now, there's 2100 and you have a huge team of people that are like trying to keep up and tune and optimize.
And even when one of the big guys comes out with a new generation of their chip, they have to go back and rewrite all these things, right?
So really it's only powered by having hundreds of people that are all like frantically trying to keep up.
And what that does is that keeps out the little guys.
And sometimes they're not so little guys, the big guys that are also just not in those dominant positions.
And so what has been happening, and so a lot of, you talk about the rise of new exotic crazy accelerators, is people have been trying to turn this from a let's go write lots of special kernels problem into a compiler problem.
And so we and I contributed to this as well, we as an industry went into it like let's go make this compiler problem phase, let's call it, and much of the industry is still in this phase by the way, so I wouldn't say this phase is over.
And so the idea is to say, look, okay, what a compiler does is it provides a much more general, extensible, hackable interface for dealing with the general case, right?
And so within machine learning algorithms, for example, people figured out that, hey, if I do a matrix multiplication, I do a ReLU, the classic activation function, it is way

faster to do one pass over the data and then do the ReLU on the output where I'm writing out the data, because ReLU is just a maximum operation, right, max is zero.
And so it's an amazing optimization, take Matmore ReLU, squish together one operation, now I have Matmore ReLU.
Well, wait a second, if I do that, now I just went from having, you know, two operators to three.
But now I figure out, okay, well, there's a lot of activation functions.
What about leaky ReLU?
Like a million things that are out there, right?
And so as I start fusing these in, now I get permutations of all these algorithms, right?
And so what the compiler people said is they said, hey, cool, well, I will go enumerate all the algorithms and I will enumerate all the pairs and I will actually generate a kernel for you.
And I think that this has been very, very useful for the industry.
This is one of the things that powers Google TPUs, PyTorch 2s, like rolling out really cool compiler stuff with Triton, this other technology and things like this.
And so the compiler people are kind of coming into their fore and saying like, awesome, this is a compiler problem, we'll compiler it.
Here's the problem.
Not everybody is a compiler person.
I love compiler people, trust me, right?
But not everybody can or should be a compiler person.
It turns out that there are people that know analog computers really well, or they know some GPU internal architecture thing really well, or they know some crazy, sparse, numeric, interesting algorithm that is the cusp of research, but they're not compiler people.
And so one of the challenges with this new wave of technology trying to turn everything into a compiler is again, it is excluded a ton of people.
And so you look at what does Mojo do, what does the modular stack do, it brings programmability back into this world.
Like it enables, I wouldn't say normal people, but a different kind of delightful nerd that cares about numerics or cares about hardware or cares about things like this to be able to express that in the stack and extend the stack without having to actually go hack the compiler itself.
So extend the stack on the algorithm side, and then on the hardware side.
Yeah, so again, go back to like the simplest example of int, right?
And so both Swift and Mojo and other things like this did is we said, okay, pull magic out of the compiler and put it in the standard library.
And so what modular is doing with the engine that we're providing and like this very deep technology stack, right, which goes into heterogeneous runtimes and like a whole bunch of really cool, really cool things.
This whole stack allows that stack to be extended and hacked and changed by researchers and by hardware innovators and by people who know things that we don't know, because you know, modular has some smart people, but we don't have all the smart people, it turns out.
What are heterogeneous runtimes?

Yeah, so what is heterogeneous, right?
So heterogeneous means many different kinds of things together.
And so the simplest example you might come up with is a CPU and a GPU.
And so it's a simple heterogeneous computer to say, I will run my data loading and preprocessing and other algorithms on the CPU.
And then once I get it into the right shape, I shove it into the GPU, I do a lot of matrix multiplications and convolutions and things like this.
And I get it back out and I do some reductions and summaries and they shove it across the wire to across the network to another machine, right?
And so you've got now what are effectively two computers, a CPU and a GPU talking to each other, working together in a heterogeneous system.
But that was 10 years ago.
Okay.
You look at a modern cell phone, modern cell phone, you've got CPUs and they're not just CPUs, there's like big dot little CPUs and so there's multiple different kinds of CPUs that are working together, they're multi-core, you've got GPUs, you've got neural network accelerators, you've got dedicated hardware blocks for media, so for video decode and JPEG decode and things like this.
And so you've got this massively complicated system and this isn't just cell phones, every laptop these days is doing the same thing and all of these blocks can run at the same time and need to be choreographed, right?
And so again, one of the cool things about machine learning is it's moving things to like data flow graphs and higher level of abstractions and tensors and these things that it doesn't specify, here's how to do the algorithm, it gives the system a lot more flexibility in terms of how to translate or map or compile it onto the system that you have.
And so what you need, the bottom part of the layer there is a way for all these devices to talk to each other.
And so this is one thing that I'm very passionate about, I'm a nerd, but all these machines and all these systems are effectively parallel computers running at the same time, sending messages to each other and so they're all fully asynchronous.
Well this is actually a small version of the same problem you have in a data center, right?
In a data center you now have multiple different machines, sometimes very specialized, sometimes with GPUs or TPUs and one node and sometimes with disks and other nodes and so you get a much larger scale heterogeneous computer and so what ends up happening is you have this like multi-layer abstraction of hierarchical parallelism, hierarchical asynchronous communication
and making that, again, my enemy is complexity by getting that away from being different specialized systems at every different part of the stack and having more consistency and uniformity, I think we can help lift the world and make it much simpler and actually get used.
But how do you leverage the strengths of the different specialized systems?
So looking inside the smartphone, I don't know, five, six computers essentially inside a smartphone, without trying to minimize the explicit, making it explicit, which computer is supposed to be used for which operation?

There's a pretty well-known algorithm and what you're doing is you're looking at two factors.
You're looking at the factor of sending data from one thing to another, because it takes time to get it from that side of the chip to that side of the chip and things like this and then you're looking at what is the time it takes to do an operation on a particular block.
So take CPUs.
CPUs are fully general, they can do anything, right?
But then you have a neural net accelerator that's really good at matrix multiplications.
And so you say, okay, well, if my workload is all matrix multiplications, I start up, I send the data over the neural net thing, it goes and does matrix multiplications, when it's done, it sends me back the result, all is good, right?
And so the simplest thing is just saying, do matrix operations over there, right?
But then you realize you get a little bit more complicated because you can do matrix multiplications on a GPU, you can do it on a neural net accelerator, you can do it on CPU, and they'll have different trade-offs and costs, and it's not just matrix multiplication.
And so what you actually look at is you look at, I have generally a graph of compute, I want to do a partitioning, I want to look at the communication, the bisection bandwidth and like the overhead and the sending of all these different things and build a model for this and then decide, okay, it's an optimization problem, where do I want to place this compute?
So the old school theoretical computer science problem of scheduling.
And then how does presumably it's possible to somehow magically include auto-tune into this?
Absolutely.
So, I mean, in my opinion, this is an opinion, this is not everybody would agree with this, but in my opinion, the world benefits from simple and predictable systems at the bottom that you can control.
But then once you have a predictable execution layer, you can build lots of different policies on top of it, right?
And so one policy can be that the human programmer says, do that here, do that here, do that here, do that here, and like fully manually controls everything, and the systems just do it, right?
Then you quickly get in the mode of like, I don't want to have to tell it to do it.
And so the next logical step that people typically take is they write some terrible heuristic.
Oh, if it's a matrix multiplication, do it over there, or if it's floating point, do it on the GPU, if it's integer, do it on the CPU, like something like that, right?
And then you then get into this mode of like, people care more and more and more, and you say, okay, well, let's actually like make the heuristic better.
Let's get into auto-tune.
Let's actually do a search of the space to decide, well, what is actually better, right?
Well, then you get into this problem where you realize this is not a small space.
This is a many-dimensional, hyper-dimensional space that you cannot exhaustively search.
So do you know of any algorithms that are good at searching very complicated spaces?
Don't tell me you're going to turn this into a machine learning problem.

So then you turn into a machine learning problem, and then you have a space of genetic algorithms and reinforcement learning and like all these, all these cool things.
Can you include that into the stack, into the module stack?
Yeah.
Yeah.
Where does it sit?
Where does it live?
Is it a separate thing or is it part of the compilation?
So you start from simple and predictable models, and so you can have full control, and you can have coarse-grain knobs like nudge systems, so you don't have to do this.
But if you really care about getting the last ounce out of a problem, then you can use additional tools.
And there are the cool things.
You don't want to do this every time you run a model.
You want to figure out the right answer and then cache it.
And once you do that, you can say, okay, cool, I can get up and running very quickly.
I can get good execution out of my system.
I can decide if something's important, and if it's important, I can go through a bunch of machines at it and do a big, expensive search over the space using whatever technique I feel like it's really up to the problem.
And then when I get the right answer, cool, I can just start using it.
And so you can get out of this trade-off between, okay, am I going to spend forever doing a thing or do I get up and running quickly and is the quality a result?
These are actually not in contention with each other if the system's designed to scale.
You started and did a little bit of a whirlwind overview of how you get the 35,000X speedup or more over Python.
Jeremy Howard did a really great presentation about the basic, look at the code, here's how you get the speedup.
Like he said, that's something we could, probably developers can do for their own code to see how you can get these gigantic speedups.
But can you maybe speak to the machine learning task in general?
How do you make some of this code fast and specific?
What would you say is the main bottleneck for machine learning tasks?
So are we talking about metmall, matrix multiplication, how do you make that fast?
So I mean, if you just look at the Python problem, right, you can say, how do I make Python faster?
And there's been a lot of people that have been working on the, okay, how do I make Python 2x faster, 10x faster or something like that, right?
And there have been a ton of projects in that vein, right?
So it started from the, what can the hardware do?
Like what is the limit of physics?
Yeah.
What is the speed of light?
What is it?

Yeah.
Like how fast can this thing go?
And then how do I express that?
Yeah.
Right.
And so it wasn't anchored relatively on make Python a little bit faster.
It's saying, cool, I know what the hardware can do, let's unlock that, right?
Now, when you-
You can just say how gutsy that is to be in the meeting and as opposed to trying to see how do we get the improvement, it's like, what can the physics do?
I mean, maybe I'm a special kind of nerd, but you look at that, what is the limit of physics?
How fast can these things go, right?
When you start looking at that, typically it ends up being a memory problem, right?
And so today, particularly with these specialized accelerators, the problem is that you can do a lot of math within them, but yet you get bottleneck sending data back and forth to memory, whether it be local memory or distant memory or disk or whatever it is.
And that bottleneck, particularly as the training sizes get large, as you start doing tons of inferences all over the place, that becomes a huge bottleneck for people, right?
So again, what happened is we went through a phase of many years where people took the special case and hand-tuned it and tweaked it and tricked it out and they knew exactly how the hardware worked and they knew the model and they made it fast, didn't generalize.
And so you can make, you know, ResNet 50 or some, or AlexNet or something, Inception V1.
You can do that, right?
Because the models are small, they fit in your head, right?
But as the models get bigger, more complicated, as the machines get more complicated, it stops working, right?
And so this is where things like kernel fusion come in.
So what is kernel fusion?
This is this idea of saying, let's avoid going to memory.
And let's do that by building a new hybrid kernel and a numerical algorithm that actually keeps things in the accelerator instead of having to write it all the way out to memory.
What's happened with these accelerators now is you get multiple levels of memory.
Like in a GPU, for example, you'll have global memory and local memory and like all these things.
If you zoom way into how hardware works, the register file is actually a memory.
So the registers are like an L0 cache.
And so a lot of taking advantage of the hardware ends up being fully utilizing the full power in all of its capability.
And this has a number of problems, right?
One of which is, again, the complexity of disaster, right?
There's too much hardware.
Even if you just say, let's look at the chips from one line of vendor like Apple or Intel

or whatever it is, each version of the chip comes out with new features and they change things so that it takes more time or less time to do different things.
And you can't rewrite all the software whenever a new chip comes out, right?
And so this is where you need a much more scalable approach.
And this is what Mojo and what the modular stack provides is it provides this infrastructure and the system for factoring all this complexity and then allowing people to express algorithms.
You talk about auto tuning, for example, express algorithms in a more portable way so that when a new chip comes out, you don't have to rewrite it all.
So to me, like, you know, I kind of joke like, what is a compiler?
Well, there's many ways to explain that.
You convert thing A into thing B and you convert source code to machine code.
Like you can talk about many, many things that compilers do.
But to me, it's about a bag of tricks.
It's about a system and a framework that you can hang complexity.
It's a system that can then generalize and it can work on problems that are bigger than fit in one human's head, right?
And so what that means, what a good stack and what the modular stack provides is the ability to walk up to it with a new problem and it'll generally work quite well.
And that's something that a lot of machine learning infrastructure and tools and technologies don't have.
Typical state of the art today is you walk up, particularly if you're deploying, if you walk up with a new model, you try to push it through the converter and the converter crashes.
That's crazy.
The state of ML tooling today is not anything that a C programmer would ever accept, right?
And it's always been this kind of flaky set of tooling that's never been integrated well and it's been never worked together because it's not designed together.
It's built by different teams.
It's built by different hardware vendors.
It's built by different systems.
It's built by different internet companies that are trying to solve their problems, right?
And so that means that we get this fragmented, terrible mess of complexity.
So I mean, the specifics of, I mean, Jeremy showed this, there's the vectorize function which I guess is built into Mojo.
Vectorize as he showed is built into the library.
Into the library instead of library.
Vectorize, paralyze, which vectorizes more low level, paralyzes higher level.
There's the tiling thing which is how he demonstrated the autotune, I think.
So think about this in like levels, hierarchical levels of abstraction, right?
And so at the very, if you zoom all the way into a compute problem, you have one floating point number, right?
And so then you say, okay, I want to be, I can do things one at a time in an interpreter.
It's pretty slow, right?
So I can get to doing one at a time in a compiler, I can see.

Then I can get to doing four or eight or 16 at a time with vectors.
That's called vectorization.
Then you can say, hey, I have a whole bunch of different, you know, what a multi-core computer is, is this basically a bunch of computers, right?
So they're all independent computers that can talk to each other and they share memory.
And so now what Paralyzed does, it says, okay, run multiple instances on different computers.
And now they can all work together on a problem, right?
And so what you're doing is you're saying, keep going out to the next level out.
And as you do that, how do I take advantage of this?
So tiling is a memory optimization, right?
It says, okay, let's make sure that we're keeping the data close to the compute part of the problem instead of sending it all back and forth through memory every time I load a block.
The size of the block size is all that's how you get to the auto tune to make sure it's optimized.
Right.
Yeah.
Well, so all of these, the details matter so much to get good performance.
This is another funny thing about machine learning and high performance computing that is very different than C compilers we all grew up with where, you know, if you get a new version of GCC or a new version of Clang or something like that, you know, maybe something will go 1% faster, right?
And so compiler insurers will work really, really, really hard to get half a percent out of your C code, something like that.
But when you're talking about an accelerator or an AI application, or you're talking about these kinds of algorithms, you know, these are things people used to write in Fortran, for example, right?
If you get it wrong, it's not 5% or 1%, it could be 2x or 10x, right?
If you think about it, you really want to make use of the full memory you have, the cache, for example, but if you use too much space, it doesn't fit in the cache, now you're going to be thrashing all the way back out to main memory.
And these can be 2x, 10x, major performance differences.
And so this is where getting these magic numbers and these things right is really actually quite important.
So you mentioned that Mojo is a superset of Python.
Can you run Python code as if it's Mojo code?
Yes.
And this has two sides of it.
So Mojo is not done yet.
So I'll give you a disclaimer, Mojo is not done yet, but already we see people that take small pieces of Python code, move it over, they don't change it, and you can get 12x speedups.
Like somebody was just tweeting about that yesterday, which is pretty cool, right?
And again, interpreters, compilers, right?

And so without changing any code, also this is not JIT compiling or doing anything fancy, because it's just basic stuff, move it straight over.
Now Mojo will continue to grow out, and as it grows out, it will have more and more and more features, and our North Star is to be a full superset of Python, and so you can bring over basically arbitrary Python code and have it just work.
And it may not always be 12x faster, but it should be at least as fast and way faster in many cases.
This is the goal, right?
Now it'll take time to do that, and Python is a complicated language.
There's not just the obvious things, but there's also non-obvious things that are complicated, like we have to be able to talk to CPython packages, to talk to the CAPI, and there's a bunch of pieces to this.
So you have to, just to make explicit, the obvious may not be so obvious until you think about it.
So to run Python code, that means you have to run all the Python packages and libraries.
So that means what?
It's the relationship between Mojo and CPython, the interpreter that presumably would be tasked with getting those packages to work.
So in the fullness of time, Mojo will solve for all the problems, and you'll be able to move Python packages over and run them in Mojo.
Without the CPython.
Without CPython.
Someday.
Right.
Not today, but someday.
And that'll be a beautiful day, because then you'll get a whole bunch of advantages, and you'll get massive speed-ups and things like this.
So you can do that one at a time, right?
You can move packages one at a time.
Exactly.
But we're not willing to wait for that.
Python is too important.
The ecosystem is too broad.
We want to be able to build Mojo out.
We also want to do it the right way, without intense time pressure.
We're obviously moving fast.
And so what we do is we say, okay, well, let's make it so you can import an arbitrary existing package, arbitrary, including like you write your own on your local disk, you know, whatever.
It's not like a standard, like an arbitrary package.
And import that using CPython, because CPython already runs all the packages, right?
And so what we do is we built an integration layer where we can actually use CPython.
Again, I'm practical, and to actually just load and use all the existing packages as they are, the downside of that is you don't get the benefits of Mojo for those packages.
And so the run as fast as they do in the traditional CPython way.

But what that does is that gives you an incremental migration path.
And so if you say, hey, cool, well, here's a, you know, the Python ecosystem is vast.
I want all of it to just work.
But there's certain things that are really important.
And so if I, if I'm doing weather forecasting or something, well, I want to be able to load all the data.
I want to be able to work with it.
And then I have my own crazy algorithm inside of it.
Well, normally I'd write that in C++, if I can write in Mojo and have one system that scales, well, that's way easier to work with.
Is it hard to do that to have that layer that's running CPython, because is there some communication back and forth?
Yes, it's complicated.
Okay.
I mean, this is what we do.
So, I mean, we make it look easy, but it is, it is complicated.
But what we do is we use the CPython existing interpreter.
So it's running its own bytecodes, and that's how it provides full compatibility.
And then it gives us CPython objects.
And we use those objects as is.
And so that way we're fully compatible with all the CPython objects and all the, you know, it's not just the Python part, it's also the C packages, the C libraries underneath them because they're often hybrid.
And so we can fully run and we're fully compatible with all that.
And the way we do that is that we have to play by the rules, right?
And so we, we keep objects in that representation when they're coming from that world.
Just the representation that's being used in memory.
We'd have to know a lot about how the CPython interpreter works.
It has, for example, reference counting, but also different rules on how to pass pointers around and things like this, super low level fiddly.
And it's not like Python, it's like how the interpreter works, okay?
And so that gets all exposed out.
And then you have to define wrappers around the low level C code, right?
And so what this means is you have to know not only C, which is a different rule from Python, obviously, not only Python, but the wrappers, but the interpreter and the wrappers and the implementation details and the conventions, and it's just this really complicated mess.
And when you do that, now suddenly you have a debugger that debug Python, they can't step into C code.
So you have this two world problem, right?
And so by pulling this all into Mojo, what you get is you get one world.
You get the ability to say, cool, I have untyped, very dynamic, beautiful, simple code.
Okay, I care about performance for whatever reason, right?
There's lots of reasons you could, you might care.
And so then you add types, you can parallelize things, you can vectorize things, you can

use these techniques, which are general techniques to solve a problem, and then you can do that by staying in the system.
And if you're, you have that one Python package that's really important to you, you can move it to Mojo, you get massive performance benefits on that, and other advantages, if you like SAC types, it's nice if they're enforced.
Some people like that, right, rather than being hints, so there's other advantages too.
And then you can do that incrementally as you go.
So one different perspective on this will be why Mojo, instead of making C Python faster or redesigning C Python?
Yeah.
You could argue Mojo is redesigning C Python, but why not make C Python faster and better and other things like that?
There's lots of people working on that.
So actually there's a team at Microsoft that is really improving, I think C Python 3.11 came out in October or something like that, and it was 15% faster, 20% faster across the board, which is pretty huge given how mature Python is and things like this.
And so that's awesome, I love it, it doesn't run on GPU, it doesn't do AI stuff, it doesn't do vectors, it doesn't do things, 20% is good, 35,000 times is better.
So they're definitely, I'm a huge fan of that work, by the way, and it composes well with what we're doing, and so it's not like we're fighting or anything like that, it's actually just goodness for the world, but it's just a different path, and again, we're not working forwards from making Python a little bit better, we're working backwards from what is the limit of physics.
What's the process of porting Python code to Mojiz?
What's involved in that process, is there tooling for that?
Not yet, so we're missing some basic features right now, and so we're continuing to drop out new features like on a weekly basis, but at the fullness of time, give us a year and a half, maybe two years.
Is it an automatable process?
When we're ready, it will be very automatable, yes.
Is it possible to automate, in the general case, the Python to Mojiz conversion, as you're saying it's possible?
Well, so, and this is why, I mean, among other reasons why we use tabs, right?
So first of all, by being a superset, it's like C versus C++, can you move C code to C++?
Yes.
Yeah, right, and you can move C code to C++, and then you can adopt classes, you can add adopt templates, you can adopt other references or whatever C++ features you want, after you move C code to C++, like you can't use templates in C, right, and so if you leave it to C, fine, you can't use the cool features, but it still works, right, and C and C++ could work together, and so that's the analogy, right?
Here, right, there's not a Python is bad and a Mojo is good, right, Mojo just gives you superpowers, right, and so if you want to stay with Python, that's cool, but the tooling should be actually very beautiful and simple because we're doing the hard work

of defining a superset.
Right, so you're right, so there's several things to say there, but also the conversion tooling should probably give you hints as to how you can improve the code.
And then, yeah, exactly, once you're in the new world, then you can build all kinds of cool tools to say like, hey, should you adopt this feature, or like, and we haven't built those tools yet, but I fully expect those tools will exist, and then you can like, you know, quote unquote modernize your code or however you want to look at it, right?
So I mean, one of the things that I think is really interesting about Mojo is that there have been a lot of projects to improve Python over the years.
Everything from, you know, getting Python around on the Java virtual machine, PyPy, which is a JIT compiler, there's tons of these projects out there that have been working on improving Python in various ways.
They followed one of two camps.
So PyPy is a great example of a camp that is trying to be compatible with Python.
Even there, not really, it doesn't work with all the C packages and stuff like that, but they're trying to be compatible with Python.
There's also another category of these things where they're saying, well, Python is too complicated.
And you know, I'm going to cheat on the edges and, you know, like integers in Python can be an arbitrary size integer, if you care about it, fitting in a, going fast on a register and a computer, that's really annoying, right?
And so you can, you can choose to pass on that, right?
You can say, well, people don't really use big integers that often, therefore, I'm going to just not do it and it will be fine.
Not a Python superset, or you can do the hard thing and say, okay, this is Python, you can't be a superset of Python without being a superset of Python.
And that's a really hard technical problem, but it's, in my opinion, worth it, right?
And it's worth it because it's not about any one package, it's about this ecosystem.
It's about what Python means for the world.
And it also means we don't want to repeat the Python 2 to Python 3 transition.
Like we want, we want people to be able to adopt this stuff quickly.
And so by doing that work, we can help lift people.
Yeah, the challenge, it's really interesting, technical, philosophical challenge of really making a language a superset of another language.
It's breaking my brain a little bit.
Well, it paints you into corners.
So again, I'm very happy with Python.
So joking, all joking aside, I think that the indentation thing is not the actual important part of the problem, right?
But the fact that Python has amazing dynamic metaprogramming features and they translate it to beautiful static metaprogramming features, I think it's profound.
I think that's huge, right?
And so Python, I've talked with Guido about this, it's like, it was not designed to do what we're doing.
That was not the reason they built it this way, but because they really cared and they

were very thoughtful about how they designed the language.
It scales very elegantly in the space.
But if you look at other languages, for example, C and C++, right?
If you're building a superset, you get stuck with the design decisions of the subset, right?
And so C++ is way more complicated because of C in the legacy than it would have been if they would have theoretically designed a from scratch thing.
And there's lots of people right now that are trying to make C++ better and re-syntax C++.
It's going to be great.
We'll just change all the syntax.
But if you do that, now suddenly you have zero packages.
You don't have compatibility.
So what are the, if you could just linger on that, what are the biggest challenges of keeping that superset status?
What are the things you're struggling with?
Is it all boiled down to having a big integer?
No, I mean, what are the other things like?
Usually it's the long tail weird things.
So let me give you a war story.
So war story in the space is, you go way back in time, the project I worked on is called Clang.
Clang, what it is, is a C C++ parser, right?
And when I started working on Clang, it must have been like 2006 or something, 2007, 2006 when I first started working on it, right?
It's funny how time flies.
I started that project and I'm like, okay, well, I want to build a C parser, C++ parser for LLVM.
It's going to be the work, GCC is yucky.
This is me in earlier times.
It's yucky.
It's unprincipled.
It has all these weird features, like all these bugs, like it's yucky.
So I'm going to build a standard compliant C and C++ parser.
It's going to be beautiful.
It'll be amazing, well engineered, all the cool things an engineer wants to do.
And so I started implementing and building it out and building it out and building it out.
And then I got to include standard IO.h.
And all of the headers in the world use all the GCC stuff.
And so again, come back away from theory back to reality, right?
I was at a fork on the road.
I could have built an amazingly beautiful academic thing that nobody would ever use.
Or I could say, well, it's yucky in various ways.
All these design mistakes, accents of history, the legacy at that point, GCC was like over

20 years old, which by the way, now LLVM is over 20 years old.
That's funny how time catches up to you, right?
And so you say, okay, well, what is easier?
Right?
I mean, as an engineer, it's actually much easier for me to go implement long tail compatibility weird features, even if they're distasteful, and just do the hard work and like figure it out, reverse engineer, understand what it is, write a bunch of test cases, like try to understand the behavior.
It's way easier to do all that work as an engineer than it is to go talk to all C programmers and get, argue with them and try to get them to rewrite their code.
Yeah.
Right.
And...
Because that breaks a lot more things.
Yeah.
And you have realities like nobody actually understands how the code works, because it was written by the person who quit 10 years ago, right?
And so this software is kind of frustrating that way, but it's, that's how the world works, right?
Yeah, unfortunately, it can never be this perfect, beautiful thing.
Well, there are occasions in which you get to build, like, you know, you invent a new data structure or something like that, or there's this beautiful algorithm that just like makes you super happy.
I love that moment, but when you're working with people, and you're working with code and dusty deck code bases and things like this, right?
It's not about what's theoretically beautiful, it's about what's practical, what's real, what people actually use.
And I don't meet a lot of people that say, I want to rewrite all my code just for the sake of it.
By the way, there could be interesting possibilities and we'll probably talk about it where AI can help rewrite some code.
That might be farther out future, but it's a really interesting one, how that could create more, be a tool in the battle against this monster of complexity that you mentioned.
Yeah.
You mentioned Guido, the benevolent dictator for life of Python.
What does he think about Mojo?
Have you talked to him much about it?
I have talked with him about it.
He found it very interesting.
We actually talked with Guido before it launched, and so he was aware of it before it went public.
I have a ton of respect for Guido for a bunch of different reasons.
You talk about Walrus operator, and Guido is pretty amazing in terms of steering such a huge and diverse community, and like driving it forward, and I think Python is what it

is thanks to him, right?
To me, it was really important starting to work on Mojo to get his feedback and get his input and get his eyes on this, right?
Now, a lot of what Guido was and is, I think, and thought about is, have we not fragment the community?
Yeah.
We don't want to Python 2 to Python 3 thing.
That was really painful for everybody involved, and so we spent quite a bit of time talking about that and some of the tricks I learned from Swift, for example.
In the migration from Swift, we managed to not just convert Objective C into a slightly prettier Objective C, which we did, we then converted not entirely, but almost an entire community to a completely different language, right?
And so there's a bunch of tricks that you learn along the way that are directly relevant to what we do.
And so this is where, for example, you leverage C Python while bringing up the new thing.
That approach is, I think, proven and then comes from experience.
And so Guido was very interested in, like, okay, cool, like, I think that Python is really his legacy.
It's his baby.
I have tons of respect for that.
Incidentally, I see Mojo as a member of the Python family, not trying to take Python away from Guido and from the Python community.
And so to me, it's really important that we're a good member of that community.
And so I think that, again, you would have to ask Guido this, but I think that he was very interested in this notion of, like, cool, Python gets beaten up for being slow.
Maybe there's a path out of that, right?
And that, you know, if the future is Python, right, I mean, look at the far outside case on this, right?
And I'm not saying this is Guido's perspective, but, you know, there's this path of saying, like, okay, well, suddenly Python can suddenly go all the places it's never been able to go before.
Right.
And that means that Python can go even further and can have even more impact on the world.
So in some sense, Mojo could be seen as Python 4.0.
I would not say that.
I think that would drive a lot of people really crazy.
Because of the PTSD of the 3.02.
I'm willing to annoy people about Emacs versus Vim versus Spaces.
That's that one.
I don't know.
That might be a little bit far even for me.
Like, my skin may not be that thick.
But the point is the step to being a super set and allowing all of these capabilities, I think, is the evolution of a language.

It feels like an evolution of a language.
So he's interested by the ideas that you're playing with, but also concerned about the fragmentation.
So how, what are the ideas you've learned?
What are you thinking about?
How do we avoid fragmenting the community?
Where the Pythonistas and the, I don't know what to call them, Mojo people, magicians, magicians.
I like it.
What can coexist happily and share code and basically just have these big code bases that are using C-Python and more and more moving towards Mojo.
Yeah.
Well, so again, these are lessons I learned from Swift.
And here we face very similar problems, right?
In Swift you have Objective C, Super Dynamic, they're very different syntax, right?
But you're talking to people who have large scale code bases.
I mean, Apple's got the biggest, largest scale code base of Objective C code, right?
And so none of the companies, none of the iOS developers, none of the other developers want to rewrite everything all at once and so you want to be able to adopt things piece at a time.
And so a thing that I found that worked very well in the Swift community was saying, okay, cool.
And this is when Swift was very young, as you say, okay, you have a million line of code Objective C app, don't rewrite it all.
But when you implement a new feature, go implement that new class using Swift, right?
And so now this turns out is a very wonderful thing for an app developer, but it's a huge challenge for this compiler team and the systems people that are implementing this, right?
And this comes back to what is this trade off between doing the hard thing that enables scale versus doing the theoretically pure and ideal thing, right?
And so Swift had adopted and built a lot of different machinery to deeply integrate with the Objective C runtime.
And we're doing the same thing with Python, right?
Now what happened in the case of Swift is that Swift's language got more and more and more mature over time, right?
And incidentally, Mojo is a much simpler language than Swift in many ways.
And so I think that Mojo will develop way faster than Swift for a variety of reasons.
But as the language gets more mature and parallel with that, you have new people starting new projects, right?
And so when the language is mature and somebody's starting a new project, that's when they say, okay, cool, I'm not dealing with a million lines of code, I'll just start and use the new thing for my whole stack.
Now the problem is, again, you come back to where communities and where people that work together, you build new subsystem or new feature or new thing in Swift or you build new thing in Mojo, then you want to be end up being used on the other side, right?

And so then you need to work on integration back the other way.
And so it's not just Mojo talking to Python, it's also Python talking to Mojo, right?
And so what I would love to see, and I don't want to see this next month, right?
But what I want to see over the course of time is I would love to see people that are building these packages, like NumPy or TensorFlow or these packages that are half Python, half C++.
And if you say, okay, cool, I want to get out of this Python C++ world into a unified world and so I can move to Mojo, but I can't give up all my Python clients because they're like, these libraries get used by everybody and they're not all going to switch all once and maybe never, right?
Well, so the way we should do that is we should vend Python interfaces to the Mojo types.
And that's what we did in Swift and worked great.
I mean, it was a huge implementation challenge for the compiler people, right?
But there's only a dozen of those compiler people and there are millions of users.
And so it's a very expensive, capital intensive, like skill set intensive problem, but once you solve that problem, it really helps adoption and really helps the community progressively adopt technologies.
And so I think that this approach will work quite well with the Python and the Mojo world.
So for a package, port it to Mojo and then create a Python interface.
So how do you just link on these packages, NumPy, PyTorch, and TensorFlow?
How do they play nicely together?
So is Mojo supposed to be, let's talk about the machine learning ones.
Is Mojo kind of vision to replace PyTorch intensive flow, to incorporate it?
What's the relationship?
All right, so take a step back.
So I wear many hats.
So you're angling it on the Mojo side.
Mojo is a programming language.
And so it can help solve the C++ Python feud that's happening.
The fire emoji got me.
I'm sorry.
We should be talking modular.
Yes.
Yes.
Okay.
So the fire emoji is amazing.
I love it.
It's a big deal.
The other side of this is the fire emoji is in service of solving some big AI problems.
And so the big AI problems are again this fragmentation, this hardware nightmare, this explosion of new potential, but that's not getting felt by the industry.
And so when you look at how does the modular engine help TensorFlow and PyTorch, it's not replacing them.
In fact, when I talk to people, again, they don't like to rewrite all their code, you

have people that are using a bunch of PyTorch, a bunch of TensorFlow.
They have models that they've been building over the course of many years.
And when I talk to them, there's a few exceptions, but generally they don't want to rewrite all their code.
And so what we're doing is we're saying, okay, well, you don't have to rewrite all your code.
What happens is the modular engine goes in there and goes underneath TensorFlow and PyTorch.
It's fully compatible and just provides better performance, better productivity, better tooling.
It's a better experience that helps lift TensorFlow and PyTorch and make them even better.
I love Python.
I love TensorFlow.
I love PyTorch, right?
This is about making the world better because we need AI to go further.
But if I have a process that trains a model and I have a process that performs inference on that model and I have the model itself, what should I do with that in the long arc of history in terms of if I use PyTorch to train it?
Should I rewrite stuff in Mojo with that if I care about performance?
Oh, so, I mean, again, it depends.
If you care about performance, then writing in Mojo is going to be way better than writing in Python.
But if you look at LLM companies, for example, if you look at OpenAI, rumored, and you look at many of the other folks that are working on many of these LLMs and other innovative machine learning models, on the one hand, they're innovating in the data collection and the model billions of parameters and the model architecture and the RLHF and all the cool things that people are talking about.
But on the other hand, they're spending a lot of time writing cuda curls, right?
And so you say, wait a second, how much faster could all this progress go if they were not having to handwrite all these cuda curls?
And so there are a few technologies that are out there, and people have been working on this problem for a while, and they're trying to solve subsets to the problem again, kind of fragmenting the space.
And so what Mojo provides for these kinds of companies is the ability to say, cool, I can have a unifying theory.
And again, the better together, the unifying theory, the two-world problem or the three-world problem or the n-world problem, this is the thing that is slowing people down.
And so as we help solve this problem, I think it'll be very helpful for making this whole cycle go faster.
So obviously we've talked about the transition from Objective C to Swift, if design this programming language.
And you've also talked quite a bit about the use of Swift for machine learning context.
Why have you decided to move away from maybe an intense focus on Swift for the machine learning context versus sort of designing a new programming language that happens to be a super simple one?
You're saying this is an irrational set of life choices I make?
Did you go to the desert and did you meditate on it?

Okay.
All right.
No, it was bold and needed.
And I think, I mean, it's just bold and sometimes to take those leaps is a difficult leap to take.
Yeah.
Well, so, okay, I mean, I think there's a couple of different things.
So actually, I left Apple back in 2017, like January 2017.
So it's been a number of years that I left Apple.
And the reason I left Apple was to do AI.
Okay.
So, and again, I won't comment on Apple and AI, but at the time, right, I wanted to get into and understand and understand the technology, understand the applications, the workloads.
And so I was like, okay, I'm going to go dive deep into applied and AI and then the technology underneath it.
Right.
I found myself at Google.
And that was like when TPUs were waking up.
Yep.
Exactly.
And so I found myself at Google and Jeff Dean, who's a rock star, as you know, right?
And in 2017, TensorFlow is like really taking off and doing incredible things.
And I was attracted to Google to help them with the TPUs.
Right.
And TPUs are an innovative hardware accelerator platform have now, I mean, I think proven massive scale and like done incredible things, right?
And so one of the things that this led into is a bunch of different projects, which I'll skip over, right?
One of which was the Swift for TensorFlow project, right?
And so that project was a research project.
And so the idea of that is say, okay, well, let's look at innovative new programming models where we can get a fast programming language, we can get automatic differentiation into the language, let's push the boundaries of these things in a research setting, right?
Now that project, I think lasted two, three years, there's some really cool outcomes of that.
So one of the things that's really interesting is I published a talk at an LLVM conference in 2018.
Again, this seems like so long ago about graph program abstraction, which is basically the thing that's in PyTorch 2.
And so PyTorch 2 with all this Dynamo real thing, it's all about this graph program abstraction thing from Python byte codes.
And so a lot of the research that was done ended up pursuing and going out through the industry and influencing things.
And I think it's super exciting and awesome to see that.

But the Swift for TensorFlow project itself did not work out super well.
And so there's a couple of different problems with that.
One of which is that you may have noticed Swift is not Python.
There's a few people that write Python code.
Yes.
And so it turns out that all of ML is pretty happy with Python.
It's actually a problem that other programming languages have as well, that they're not Python.
We'll probably maybe briefly talk about Julia, who's a very interesting, beautiful programming language, but it's not Python.
Exactly.
Well, and so like if you're saying, I'm going to solve a machine learning problem where all the programmers are Python programmers.
And you say the first thing you have to do is switch to a different language.
Well, your new thing may be good or bad or whatever, but if it's a new thing, the adoption barrier is massive.
It's still possible.
Still possible?
Yeah, absolutely.
The world changes and evolves and there's definitely room for new and good ideas, but it just makes it so much harder.
And so lesson learned, Swift is not Python and people are not always in search of learning a new thing for the sake of learning a new thing.
And if you want to be compatible with all the world's code, it turns out, meet the world where it is.
One thing is that lesson learned is that Swift as a very fast and efficient language, kind of like Mojo, but a different take on it still, really worked well with Eager mode.
And so Eager mode is something that PyTorch does and it proved out really well and it enables really expressive and dynamic and easy to debug programming.
TensorFlow at the time was not set up for that, let's say.
The timing is also important in this world.
Yeah, TensorFlow is a good thing and it has many, many strengths, but you could say Swift for TensorFlow is a good idea except for the Swift and except for the TensorFlow part.
Because it's not Python and TensorFlow because it's not Eager mode at the time.
That is 1.0.
Exactly.
And so one of the things about that is in the context of it being a research project, I'm very happy with the fact that we built a lot of really cool technology, we learned a lot of things.
I think the ideas went on to have influence in other systems like PyTorch, a few people use that right here.
And so I think that's super cool.
And for me personally, I learned so much from it.
And I think a lot of the engineers that worked on it also learned a tremendous amount.
And so I think that that's just really exciting to see and I'm sorry that the project didn't

work out.
It did, of course, but it's a research project and so you're there to learn from it.
It's interesting to think about the evolution of programming as we come up with these whole new set of algorithms in machine learning and artificial intelligence and what's going to win out.
Because it could be a new programming language.
It could be, I just mentioned Julia.
I think there's a lot of ideas behind Julia that Mojo shares.
What are your thoughts about Julia in general?
So I will have to say that when we launched Mojo, one of the biggest things I didn't predict was the response from the Julia community.
And so I was not, okay, let me take a step back.
I've known the Julia folks for a really long time.
They were an adopter of LLVM a long time ago.
They've been pushing state of the art in a bunch of different ways.
Julia is a really cool system.
I had always thought of Julia as being mostly a scientific computing focused environment.
And I thought that was its focus.
I neglected to understand that one of their missions is to help make Python work end to end.
And I think that was my error for not understanding that.
And so I could have been maybe more sensitive to that.
But there's major differences between what Mojo is doing and what Julia is doing.
So as you say, Julia is not Python.
And so one of the things that a lot of the Julia people came out and said is like, okay, well, if we put a ton more energy and ton more money or engineering or whatever into Julia, maybe that would be better than starting Mojo.
Maybe that's true, but it still wouldn't make Julia into Python.
So if you've worked backwards from the goal of let's build something for Python programmers without requiring them to relearn syntax, then Julia just isn't there.
I mean, that's a different thing.
And so if you anchor on, I love Julia and I want Julia to go further, then you can look at it from a different lens.
But the lens we were coming at was, hey, everybody is using Python.
Syntax isn't broken.
Let's take what's great about Python and make it even better.
And so it's just a different starting point.
So I think Julia is a great language.
The community is a lovely community.
They're doing really cool stuff, but it's just a slightly different angle.
But it does seem that Python is quite sticky.
Is there some philosophical, almost thing you could say about why Python by many measures seems to be the most popular programming language in the world?
Well, I can tell you things.

I love about it.
Maybe that's one way to answer the question.
So huge package ecosystem, super lightweight and easy to integrate, it has very low startup time.
So at what startup time, you mean like learning curve or what?
Yeah.
So if you look at certain other languages, you say go, and it just takes Java, for example, it takes a long time to compile all the things, and then the VM starts up and the garbage clusters kicks in and then it revs its engines and then it can plow through a lot of internet stuff or whatever.
Python is like scripting.
It just goes.
Python has very low compile time, so you're not sitting there waiting.
Python integrates into notebooks in a very elegant way that makes exploration super interactive and it's awesome.
Python is also, it's like almost the glue of computing because it has such a simple object representation, a lot of things plug into it.
That dynamic metaprogramming thing we were talking about also enables really expressive and beautiful APIs.
So there's lots of reasons that you can look at technical things that Python has done and say like, okay, wow, this is actually a pretty amazing thing and any one of those you can neglect, people all just talk about indentation and ignore the fundamental things.
But then you also look at the community side.
So Python owns machine learning.
Machine learning is pretty big.
Yeah, and it's growing.
It's growing.
It's growing in importance.
And there's a reputation of prestige to machine learning to where if you're a new programmer and you're thinking about which programming language do I use, well, I should probably care about machine learning.
Therefore, let me try Python and it builds and builds and builds.
And you even go back before that.
My kids learn Python, not because I'm telling them to learn Python, but because that-
Were they rebelling against you or what?
No, no, no, they also learn Scratch and things like this too.
But it's because Python is taught everywhere because it's easy to learn and because it's pervasive.
Back to my day, we learned Java and C++ uphill both directions, but yes, I guess Python is the main language of teaching software engineering in schools now.
Yeah, well, if you look at this, there's these growth cycles.
If you look at what causes things to become popular and then gain in popularity, there's reinforcing feedback loops and things like this.
And I think Python has done, again, the whole community has done a really good job of building

those growth loops and help propel the ecosystem.
And I think that, again, you look at what you can get done with just a few lines of code.
It's amazing.
So this kind of self-building loop is interesting to understand because when you look at Mojo, what it stands for, some of the features, it seems sort of clear that this is a good direction for programming languages to evolve in the machine learning community.
But it's still not obvious that it will because of this, whatever the engine of popularity, of virality.
Is there something you could speak to, like how do you get people to switch?
Yeah.
Well, I mean, I think that the viral growth loop is to switch people to Unicode.
I think the Unicode file extensions are what I'm betting on.
I think that's going to be the thing.
Yeah.
Tell the kids that you could use the FireModule and they'd be like, what?
Exactly.
Well, in all seriousness, I think there's really, I'll give you two opposite answers.
One is, I hope if it's useful, if it solves problems and people care about those problems being solved, they'll adopt the tech.
That's kind of the simple answer.
And when you're looking to get tech adopted, the question is, is it solving an important problem people need solved?
And is the adoption cost low enough that they're willing to make the switch and cut over and do the pain up front so that they can actually do it, right?
And so hopefully Mojo will be that for a bunch of people and people building these hybrid packages are suffering and it's really painful.
And so I think that we have a good shot of helping people.
But the other side is like, it's okay if people don't use Mojo.
It's not my job to say like, everybody should do this.
Like I'm not saying Python is bad.
Like I hope Python see Python, like all these implementations because Python ecosystem is not just see Python.
It's also a bunch of different implementations with different tradeoffs.
And this ecosystem is really powerful and exciting as our other programming languages.
It's not like TypeScript or something is going to go away, right?
And so there's not a winner take all thing.
And so I hope that Mojo is exciting and useful to people, but if it's not, that's also fine.
But I also wonder what the use case for why you should try Mojo would be.
So practically speaking, it seems like, so there's entertainment.
There's the dopamine hit of saying, holy shit, this is 10 times faster.
This little piece of code is 10 times faster in Mojo.
Out of the box before he gets to 35,000.
Exactly.

Just even that, I mean, that's the dopamine hit that every programmer sort of dreams of is the optimization.
It's also the drug that can pull you in and have you waste way too much of your life optimizing and over optimizing, right?
But so what do you see that would be like comedy?
It's very hard to predict, of course, but if you look 10 years from now on Mojo is super successful.
What do you think would be the thing where people like try it and then use it regularly and it kind of grows and grows and grows and grows?
Well, so you talk about dopamine hit.
And so again, humans are not one thing.
And some people love rewriting their code and learning new things and throwing themselves in the deep end and try out a new thing.
In my experience, most people don't like they're too busy.
They have other things going on.
By number, most people don't like this.
I want to rewrite all my code.
But even those people, the two busy people, the people that don't actually care about the language that just care about getting stuff done, those people do like learning new things, right?
And so you talk about the dopamine rush of 10x faster.
Wow, that's cool.
I want to do that again.
Well, it's also like, here's the thing I've heard about in a different domain and I don't have to rewrite all my code.
I can learn a new trick, right?
That's called growth.
And so one thing that I think is cool about Mojo, and again, those will take a little bit of time for, for example, the blog posts and the books and like all that kind of stuff to develop and the language needs to get further along.
But what we're doing, you talk about types, like you can say, look, you can start with the world you already know, and you can progressively learn new things and adopt them where it makes
sense.
If you never do that, that's cool, you're not a bad person.
If you get really excited about it and want to go all the way in the deep end and write everything and look, whatever, that's cool, right?
But I think the middle path is actually the more likely one where it's, you know, you come out with a new, a new idea and you discover, wow, that makes my code way simpler, way more beautiful, way faster, way whatever.
And I think that's what people like.
Now, if you fast forward and you said like 10 years up, right, I can give you a very different answer on that, which is, I mean, if you go back and look at what computers look like 20 years ago, every 18 months, they got faster for free, right, 2x faster every

18 months.
It was like clockwork.
It was, it was free, right?
You go back 10 years ago and we entered in this world where suddenly we had multi-core CPUs and we had GPUs.
And if you squint and turn your head, what a GPU is, it's just a many core, very simple CPU thing kind of, right?
And so, and 10 years ago, it was CPUs and GPUs and graphics.
Today, we have CPUs, GPUs, graphics and AI because it's so important because the compute is so demanding because of the smart cameras and the watches and all the different places that AI needs to work in our lives.
It's caused this explosion of hardware.
And so, part of my thesis, part of my belief of where computing goes, if you look at 10 years from now, is it's not going to get simpler.
Physics isn't going back to where we came from.
It's only going to get weirder from here on out, right?
And so, to me, the exciting part about what we're building is it's about building that universal platform which the world can continue to get weird because, again, I don't think it's avoidable.
It's physics.
But we can help lift people scale, do things with it and they don't have to rewrite their code every time a new device comes out.
And I think that's pretty cool.
And so, if Mojo can help with that problem, then I think that it will be hopefully quite interesting and quite useful to a wide range of people because there's so much potential and maybe analog computers will become a thing or something, right?
And we need to be able to get into a mode where we can move this programming model forward but do so in a way where we're lifting people and growing them instead of forcing them to rewrite all their code and exploding them.
Do you think there'll be a few major libraries that go Mojo first?
Well, so, I mean, the modular engine is all Mojo.
So, again, come back to, like, we're not building Mojo because it's fun.
We're building Mojo because we had to dissolve these accelerators.
That's the origin story.
But I mean, ones that are currently in Python.
Yeah.
So, I think that a number of these projects will.
And so, one of the things, again, this is just my best guess, like, each of the package maintainers also has, I'm sure, plenty of other things going on.
People don't, like, really don't like rewriting code just for the sake of rewriting code.
But sometimes, like, people are excited about, like, adopting a new idea.
Yeah.
And it turns out that while rewriting code is generally not people's first thing, turns out that redesigning something while you rewrite it and using a rewrite as an excuse to redesign

can lead to the 2.0 of your thing that's way better than the 1.0, right?
And so, I have no idea.
I can't predict that.
But there's a lot of these places where, again, if you have a package that is half C and half Python, right, you just solve the pain, make it easier to move things faster, make it easier to debug and evolve your tech.
Adopting Mojo kind of makes sense to start with, and then it gives you this opportunity to rethink these things.
So the two big gains are that there's a performance gain, and then there's the portability to all kinds of different devices.
And there's safety, right?
So you talk about real types.
I mean, not saying this is for everybody, but that's actually a pretty big thing, right?
Yeah, types are.
And so there's a bunch of different aspects of what value Mojo provides.
And so, I mean, it's funny for me, like, I've been working on these kinds of technologies and tools for too many years now, but you look at Swift, right?
And we talked about Swift for TensorFlow, but Swift as a programming language, right?
Swift's now 13 years old from when I started it.
Yeah.
I started in 2010, if I remember.
And so that project, and I was involved with it for 12 years or something, right?
That project has gone through its own really interesting story arc, right?
And it's a mature, successful, used by millions of people's system, right?
It's certainly not dead yet, right?
But also, going through that story arc, I learned a tremendous amount about building languages, about building compilers, about working with the community, and things like this.
And that experience, like, I'm helping channel and bring directly into Mojo and, you know, other systems, same thing.
Like, apparently, building and iterating and evolving things.
And so you look at this LLVM thing I worked on 20 years ago, and you look at MLIR, right?
And so a lot of the lessons learned in LLVM got fed into MLIR, and I think that MLIR is a way better system than LLVM was.
And, you know, Swift is a really good system, and it's amazing, but I hope that Mojo will take the next step forward in terms of design.
In terms of running Mojo, people can play with it.
What's Mojo Playground?
And from the interface perspective and from the hardware perspective, what's this incredible thing running on?
Yeah, so right now, so here we are two weeks after launch.
We decided that, okay, we have this incredible set of technology that we think might be good, but we have not given it to lots of people yet.
And so we were very conservative and said, let's put it in a workbook so that if it crashes,

we can do something about it.
We can monitor and track that, right?
And so, again, things are still super early, but we're having, like, one person a minute sign up with over 70,000 people two weeks in is kind of crazy.
So you can sign up to Mojo Playground and you can use it in the cloud, in your browser.
And so what that's running on is that's running on cloud VMs, and so you share a machine with a bunch of other people, but it turns out there's a bunch of them now because there's a lot of people.
And so what you're doing is you're getting free compute and you're getting to play with this thing in kind of a limited controlled way so that we can make sure that it doesn't totally crash and be embarrassing, right?
So now a lot of the feedback we've gotten is people want to download it locally, so we're working on that right now.
So that's the goal to be able to download locally?
Yeah, that's what everybody expects.
And so we're working on that right now.
And so we just want to make sure that we do it right.
And I think this is one of the lessons I learned from Swift also, by the way, is that when we launched Swift, it feels like forever ago, it was 2014.
And it was super exciting.
I and we, the team, had worked on Swift for a number of years in secrecy.
We four years into this development, roughly, of working on this thing.
At that point, about 250 people at Apple knew about it.
So it was secret.
Apple's good at secrecy, and it was a secret project.
And so we launched this at WWC, a bunch of hoopla and excitement, and said, developers, you're going to be able to develop and submit apps to the App Store in three months.
Well, several interesting things happened, right?
So first of all, we learned that, A, it had a lot of bugs, and it was not actually production quality.
And it was extremely stressful in terms of trying to get it working for a bunch of people.
And so what happened was we went from zero to, I don't know how many developers Apple had at the time, but a lot of developers overnight, and they ran into a lot of bugs, and it was really embarrassing, and it was very stressful for everybody involved, right?
It was also very exciting, because everybody was excited about that.
The other thing I learned is that when that happened, roughly every software engineer who did not know about the project at Apple, their head exploded when it was launched, because they didn't know it was coming.
And so they're like, wait, what is this?
I signed up to work for Apple because I love Objective C. Why is there a new thing?
And so now what that meant practically is that the push from launch to, first of all, the fall, but then to 2.0 and 3.0 and all the way forward was super painful for the engineering team and myself.
It was very stressful.

The developer community was very grumpy about it, because they're like, okay, well, wait a second, you're changing and breaking my code, and we have to fix the bugs.
And it was just a lot of tension and friction on all sides.
There's a lot of technical debt in the compiler, because we have to run really fast, and you have to go implement the thing and unblock the use case and do the thing, and you know it's not right, but you never have time to go back and do it, right?
And I'm very proud of the Swift team because they've come, I mean, we, but they came so far and made so much progress over this time since launch.
It's pretty incredible, and Swift is a very, very good thing, but I just don't want to do that again, right?
So iterate more through the development process.
And so what we're doing is we're not launching it when it's hopefully 0.9 with no testers.
We're launching it and saying it's 0.1, right?
And so we're saying expectations of saying, like, okay, well, don't use this for production, right?
If you're interested in what we're doing, we'll do it in an open way, and we can do it together, but don't use it in production yet, like we'll get there, but let's do it the right way.
And I'm also saying we're not in a race.
The thing that I want to do is build the world's best thing, right?
Because if you do it right and it lifts the industry, it doesn't matter if it takes an extra two months.
Like two months is worth waiting.
And so doing it right and not being overwhelmed with technical debt and things like this is like, again, war wounds, lessons learned, whatever you want to say, I think is absolutely the right thing to do, even though right now people are very frustrated that you can't download it or it doesn't have feature X or something like this.
What have you learned in a little bit of time since it's been released into the wild that people have been complaining about feature X or Y or Z, what have they been complaining about, what they have been excited about, almost like detailed things versus a big vision.
I think everyone would be very excited about the big vision.
Yeah.
Yeah.
Well, so I mean, I've been very pleased.
In fact, I mean, we've been massively overwhelmed with response, which is a good problem to have.
It's kind of like a success disaster, in a sense, right?
And so, I mean, if you go back in time, when we started Modular, which is just not yet a year and a half ago, so it's still a pretty new company, new team, small but very good team of people, like we started with extreme conviction that there's a set of problems that we need to solve.
And if we solve it, then people will be interested in what we're doing, right?
But again, you're building in basically secret, right?
You're trying to figure it out.
It's the creation is a messy process.
You're having to go through different paths and understand what you want to do and how

to explain it.
Often, when you're doing disruptive and new kinds of things, just knowing how to explain it is super difficult, right?
And so, when we launched, we hoped people would be excited, but I'm an optimist, but I'm also like, don't want to get ahead of myself.
And so, when people found out about Mojo, I think their heads exploded a little bit, right?
And here's, I think, a pretty credible team that has built some languages and some tools before.
And so, they have some lessons learned and are tackling some of the deep problems in the Python ecosystem and giving it the love and attention that it should be getting.
And I think people got very excited about that.
And so, if you look at that, I mean, I think people are excited about ownership and taking a step beyond rust, right?
There's people that are very excited about that.
There's people that are excited about, you know, just like, I made Game of Life go 400 times faster, right?
And things like that.
And that's really cool.
And they're really excited about the, okay, I really hate writing stuff in C++.
Save me.
Like systems in your, they're like stepping up like, oh, yes.
Yeah.
And so, that's me, by the way, also, I really want to stop writing C++.
But the...
I get third-person excitement when people tweet, yeah, I made this code, Game of Life or whatever, faster, and you're like, yeah.
And also, like, I would also say that let me cast blame out to people who deserve it.
Sure.
These terrible people who convinced me to do some of this.
Yes.
Jeremy Howard.
Yes.
That guy.
Well, he's been pushing for this kind of thing.
He's been pushing for more than...
He's wanted this for years.
Yeah.
He's wanted this for a long, long time.
He's wanted this for years.
And so...
For people who don't know Jeremy Howard, he's like one of the most legit people in the machine learning community.
He's a grassroots.

He really teaches.
He's an incredible educator, he's an incredible teacher, but also legit in terms of a machine learning engineer himself.
Yeah.
And he's been running the fast.ai and looking, I think, for exactly what you've done.
Exactly.
And so, I mean, the first time...
So I met Jeremy pretty early on.
But the first time I sat up and I'm like, this guy is ridiculous, is when I was at Google and we were bringing up TPUs and we had a whole team of people and there was this competition called Don Bench of who can train ImageNet fastest.
And Jeremy and one of his researchers crushed Google by not through sheer force of the amazing amount of compute and the number of TPUs and stuff like that, that he just decided that progressive imagery sizing was the right way to train the model.
And if you were at epochs faster and make the whole thing go vroom, right?
And I'm like, this guy is incredible.
And so you can say, anyways, come back to where's Mojo coming from.
Chris finally listened to Jeremy.
It's all his fault.
Well, there's a kind of very refreshing, pragmatic view that he has about machine learning that I don't know if it's this mix of a desire for efficiency, but ultimately grounded in desire to make machine learning more accessible to a lot of people.
I don't know what that is.
I guess that's coupled with efficiency and performance, but it's not just obsessed about performance.
So a lot of AI and AI research ends up being that it has to go fast enough to get scale.
So a lot of people don't actually care about performance, particularly on the research side, until it allows them to have a bigger data set.
And so suddenly now you care about distributed compute and all these exotic HPC.
You don't actually want to know about that.
You just want to be able to do more experiments faster and do so with bigger data sets.
And so Jeremy has been really pushing the limits.
And one of the things I'll say about Jeremy, and there's many things I could say about Jeremy because I'm a fanboy of his, but he fits in his head.
And Jeremy actually takes the time, where many people don't, to really dive deep into why is the beta parameter of the atom optimizer equal to this?
And he'll go survey and understand what are all the activation functions in the trade-offs and why is it that everybody that does this model pick that thing?
So the why, not just trying different values, like really what is going on here.
And so as a consequence of that, like he's always, again, he makes time, but he spends time to understand things at a depth that a lot of people don't.
And as you say, he then brings it and teaches people.
And his mission is to help lift, you know, his website says, making AI uncool again.
Like, it's about, like, forget about the hype.

It's actually practical and useful.
Let's teach people how to do this, right?
Now the problem Jeremy struggled with is that he's pushing the envelope, right?
Research isn't about doing the thing that is staying on the happy path or the well-paid road, right?
And so a lot of the systems today have been these really fragile, fragmented things or special case in this happy path.
And if you fall off the happy path, you get eaten by an alligator.
So what about, so Python has this giant ecosystem of packages and is a package repository.
Do you have ideas of how to do that well for Mojo?
Yeah.
How to do a repository of packages?
Well, so that's another really interesting problem that I knew about, but I didn't understand how big of a problem it was.
Python packaging, a lot of people have very big pain points and a lot of scars with Python packaging.
Oh, you mean, so there's several things to say.
Building and distributing and managing dependencies and versioning and all this stuff.
So from the perspective of if you want to create your own package.
Yes.
And then, or you want to build on top of a bunch of other people's packages and then they get updated and things like this.
Now I'm not an expert in this, so I don't know the answer.
I think this is one of the reasons why it's great that we work as a team and there's other really good and smart people involved.
But one of my, one of the things I've heard from smart people who've done a lot of this is that the packaging becomes a huge disaster when you get the Python and C together.
And so if you have this problem where you have code split between Python and C, now not only do you have to package the C code, you have to build the C code.
C doesn't have a package manager, right?
C doesn't have a dependency versioning management system, right?
And so I'm not experienced in the state of the art and all the different Python package managers, but I might be saying is that's a massive part of the problem.
And I think Mojo solves that part of the problem directly heads on.
Now one of the things I think we'll do with the community, and this isn't, again, we're not solving all the world's problems at once.
We have to be kind of focused to start with, is that I think that we will have an opportunity to reevaluate packaging, right?
And so I think that we can come back and say, okay, well, given the new tools and technologies and the cool things we have that we've built up, because we have not just syntax, we have an entirely new compiler stack that works in a new way, maybe there's other innovations we can bring together and maybe we can help solve that problem.
So almost the tangent to that question from the user perspective of packages, it was always surprising to me that it was not easier to sort of explore and find packages with pip install.

And it just, it feels, it's an incredible ecosystem.
It's just interesting that it wasn't made, it's still, I think, not made easier to discover packages to do like search and discovery, as YouTube calls it.
Well, I mean, it's kind of funny because this is one of the challenges of these like intentionally decentralized communities.
And so I don't know what the right answer is for Python.
I mean, there are many people that were, I don't even know the right answer for Mojo.
So there are many people that would have much more informed opinions than I do, but, but it's interesting if you look at this, right, open source communities, you know, there's Git, Git is a fully decentralized, they can do it any way they want, but then there's GitHub, right, and GitHub centralized, commercial in that case, right, thing, really help pull together and help solve some of the discovery problems and help build a more consistent community.
And so maybe there's opportunities for something like a GitHub for, yeah, although even GitHub, I might be wrong on this, but the, the search and discovery for GitHub is not that great.
Like I still use Google search.
Yeah.
Well, I mean, maybe that's because GitHub doesn't want to replace Google search.
Right.
I think there is room for specialized solutions to specific problems.
Sure.
I don't know.
I don't know the right answer for GitHub either.
That's, I think they can go figure that out.
But the point is to have an interface that's usable, that's accessible to people of all different skill levels.
Well, and again, like what, what are the benefit of standards, right?
Standards allow you to build these next level up ecosystem, next level up infrastructure, next level up things.
And so again, come back to, I hate complexity.
See, C plus Python is complicated.
It makes everything more difficult to deal with.
It makes it difficult to port, move code around, work with all these things get more complicated.
And so, I mean, I'm not an expert, but maybe Mojo can help a little bit by helping reduce the amount of C in this ecosystem and make it therefore scale better.
So you kind of package is that a hybrid in nature would be a natural fit to move to Mojo.
Which is a lot of them, by the way.
Yeah.
A lot of them, especially they're doing some interesting stuff, computation wise.
Let me ask you about some features.
Yeah.
So we talked about, obviously, the indentation that it's the type language or optionally typed.

Is that the right way to say it?
It's either optionally or progressively.
Progressively.
I think so, so, so people have very strong opinions on the right word to use.
Yeah.
I don't know.
I look forward to your letters.
So there's the var versus let, but let is for constants, var is an optional.
Yeah, var makes it mutable, so you can reassign.
Okay.
Then there's function overloading.
Oh, okay.
Yeah.
I mean, there's a lot of source of happiness for me, but function overloading that's, I guess, is that for performance or is that, why does Python not have function overloading?
So I can speculate.
So Python is a dynamic language.
The way it works is that Python and Objective C are actually very similar worlds if you ignore syntax.
And so Objective C is straight line derived from small talk, a really venerable, interesting language that much of the world has forgotten about, but the people that remember it, love it generally.
And the way that small talk works is that every object has a dictionary in it, and the dictionary maps from the name of a function or the name of a value within an object to its implementation.
And so the way you call a method in Objective C is you say, go look up the way I call foo is I go look up foo, I get a pointer to the function back and then I call it.
Okay.
That's how Python works.
Right.
And so now the problem with that is that the dictionary within a Python object, all the keys are strings, and it's a dictionary.
So you can only have one entry per name.
You think it's as simple as that?
I think it's as simple as that.
And so now why do they never fix this?
Like why do they not change it to not be a dictionary?
Like I do other things.
Well, you don't really have to in Python because it's dynamic.
And so you can say I get into the function.
Now if I got past an integer, do some dynamic test for it.
If it's a string, go do another thing.
There's another additional challenge, which is even if you did support overloading, you're saying, okay, well, here's a version of a function for integers and a function for strings.

Well, you'd have, even if you could put it in that dictionary, you'd have to have the caller do the dispatch.
And so every time you call the function, you'd have to say, like, is an integer a string?
And so you'd have to figure out where to do that test.
And so in a dynamic language, overloading is something you don't have to have.
So, but now you get into a type language.
And, you know, in Python, if you subscript with an integer, then you get typically one element out of a collection.
If you subscript with a range, you get a different thing out, right?
And so often in type languages, you'll want to be able to express the fact that, cool, I have different behavior, depending on what I actually pass into this thing.
If you can model that, it can make it safer and more predictable and faster and like all these things.
It somehow feels safer, yes, but also feels empowering, like in terms of clarity, like you don't have to design, hold different functions.
Yeah.
Well, this is also one of the challenges with the existing Python typing systems is that in practice, like you take subscript, like in practice, a lot of these functions, they don't have one signature, right?
They actually have different behavior in different cases.
And so this is why it's difficult to like retrofit this into existing Python code and make it play well with typing.
You can have to design for that.
Okay.
So there's a interesting distinction that people, the program Python might be interested in is def versus fn.
So it's two different ways to define a function.
And fn is a stricter version of def.
What's the coolness that comes from the strictness?
So here you get into what is the trade off with the superset?
Yes.
Okay.
So superset, you have to, or you really want to be compatible.
You've, like, if you're doing a superset, you've decided compatibility with existing code is the important thing, even if some of the decisions they made were maybe not what you'd choose.
Yeah.
Okay.
So that means you put a lot of time in compatibility, and it means that you get locked into decisions of the past, even if they may not have been a good thing, right?
Now, systems programmers typically like to control things, right?
And they want to make sure that, you know, not all cases, of course, and even systems programmers are not one thing, right?
But, but often you want predictability.

And so one of the things that Python has, for example, as you know, is that if you define a variable, you just say x equals four.
I have a variable named x.
Now I say some long method, some, some long name equals 17.
Print out some long name.
Oops, by typo, right?
Well, the compiler, the Python compiler doesn't know, in all cases, what you're defining and what you're using.
And did you typo the use of it or the definition, right?
And so for people coming from type languages, again, I'm not saying the right wrong, but that drives them crazy because they want the compiler to tell them you typo the name of this thing, right?
And so what fn does is it turns on, as you say, it's a strict mode.
And so it says, okay, well, you have to actually declare, intentionally declare variables before you use them, that gives you more predictability, more error checking and things like this, but you don't have to, you don't have to use it.
And this is a way that Mojo is both compatible because defs work the same way that defs have already always worked, but it provides a new alternative that gives you more control and it allows certain kinds of people that have a different philosophy to be able to express that and get that.
But usually if you're writing Mojo code from scratch, you'll be using fn.
It depends, again, it depends on your mentality, right?
It's not that def is Python and fn is Mojo.
Mojo has both and it loves both, right?
And it really depends on...
Time is just strict.
Yeah, exactly.
Are you playing around and scripting something out?
Is it a one-off throwaway script?
Cool, like Python is great at that.
I will still be using fn, but yeah.
Well, so...
I love strictness.
Okay, well, so...
Control.
Power.
You also like suffering, right?
Yes, I go hand in hand.
How many pull-ups?
I've lost count at this point.
So, and that's cool.
I love you for that.
Yeah.
And I love other people who like strict things, right?

But I don't want to say that that's the right thing because Python's also very beautiful for hacking around and doing stuff and research and these other cases where you may not want that.
You see, I just feel like...
Maybe I'm wrong with that, but it feels like strictness leads to faster debugging.
So, in terms of going from even on a small project from zero to completion, it just...
I guess it depends how many bugs you generate usually.
Well, so, I mean, it's, again, lessons learned in looking at the ecosystem.
It's really...
I mean, I think it's...
If you study some of these languages over time, like the Ruby community, for example.
Now, Ruby is a pretty well-developed, pretty established community, but along their path, they really invested in unit testing.
So, I think that the Ruby community has really pushed forward the state of the art of testing because they didn't have a type system that caught a lot of bugs at compile time, right?
And so, you can have the best of both worlds.
You can have good testing and good types and things like this, but I thought that it was really interesting to see how certain challenges get solved.
And in Python, for example, the interactive notebook kind of experiences and stuff like this are really amazing.
If you typo something, it doesn't matter.
It just tells you, that's fine, right?
And so, I think that the trade-offs are very different if you're building a large-scale production system versus you're building and exploring in a notebook.
Speaking of control, the hilarious thing, if you look at code I read just for myself for fun, it's like littered with asserts everywhere.
It's a kind of...
Well, yeah, you'd like that.
It's basically saying, in a dictatorial way, this should be true now.
Otherwise, everything stops.
And that is the sign.
I love you, man.
That is the sign of somebody who likes control.
And so, yes, I think that you'll like...
I think you'll like mojo.
Therapy session.
Yes, I definitely will.
Speaking of asserts, exceptions are called errors.
Why is it called errors?
So, I mean, we use the same, we're the same as Python, right?
But we implement it in a very different way, right?
And so, if you look at other languages, like we'll pick on C++, our favorite,

right, C++ has a thing called zero-cost exception handling.
Okay, and this is, in my opinion, something to learn lessons from.
It's a nice polite way of saying it.
And so, zero-cost exception handling, the way it works is that it's called zero-cost because if you don't throw an exception, there's supposed to be no overhead for the non-error code.
And so, it takes the error path out of the common path.
It does this by making throwing an error extremely expensive.
And so, if you actually throw an error with a C++ compiler using exceptions, it has to go look up in tables on the side and do all this stuff.
And so, throwing an error could be like 10,000 times more expensive than returning from a function, right?
Also, it's called zero-cost exceptions, but it's not zero-cost.
By any stretch of the imagination, because it massively blows out your code, your binary, it also adds a whole bunch of different paths because of destructors and other things like that that exist in C++.
And it reduces the number of optimizations and adds like all these effects.
And so, this thing that was called zero-cost exceptions, it really ain't.
Now, if you fast forward to newer languages, and then this includes Swift and Rust and Go and now Mojo, well, and Python's a little bit different because it's interpreted, and so, it's got a little bit of a different thing going on.
But if you look at it, if you look at compiled languages, many newer languages say, okay, well, let's not do that zero-cost exception handling thing.
Let's actually treat throwing an error the same as returning a variant, returning either the normal result or an error.
Now, programmers generally don't want to deal with all the typing machinery and like pushing around a variant.
And so, you use all the syntax that Python gives us, for example, try and catch.
Functions that raise and things like this, you can put a raises decorator on your function, stuff like this.
And if you want to control that, and then the language can provide syntax for it.
But under the hood, the way the computer executes it, throwing errors basically as fast as returning something.
So, it's exactly the same way, from a compiled perspective.
And so, this is actually, I mean, it's a fairly nerdy thing, right, which is why I love it, but this has a huge impact on the way you design your APIs, right?
So, in C++, huge communities turn off exceptions because the cost is just so high, right?
And so, the zero-cost cost is so high, right?
And so, that means you can't actually use exceptions in many libraries.
Right?
And even for the people that do use it, well, okay, how and when do you want to pay the cost?
If I try to open a file, should I throw an error?

Well, what if I'm probing around looking for something, right?
I'm looking it up and making a different pass.
Well, if it's really slow to do that, maybe I'll add another function that
doesn't throw an error, returns an error code instead.
And I have two different versions of the same thing.
And so, it causes you to fork your APIs.
And so, one of the things I learned from Apple and I so love is the art of API
design is actually really profound.
I think this is something that Python's also done a pretty good job at in
terms of building out this large-scale package ecosystem.
It's about having standards and things like this.
And so, we wouldn't want to enter a mode where there's this theoretical feature
that exists in language, but people don't use it in practice.
Now, I'll also say one of the other really cool things about this implementation
approach is that it can run on GPUs and it can run on accelerators and things
like this.
And that standard zero-cost exception thing would never work on an accelerator.
And so, this is also part of how Mojo can scale all the way down to like little
embedded systems and to running on GPUs and things like that.
Can you actually say about the, maybe, is there some high-level way to describe
the challenge of exceptions and how they work in code during compilation?
So, it's just this idea of percolating up a thing, an error.
Yeah.
Yeah.
So, the way to think about it is, think about a function that doesn't
return anything, just as a simple case, right?
And so, you have function one calls function two, calls function three, calls
function four, along that call stack that are tri-blocks, right?
And so, if you have function one calls function two, function two has a tri-block
and then within it, it calls function three, right?
Well, what happens if function three throws?
Well, actually, it starts simpler.
What happens if it returns?
Well, if it returns, it's supposed to go back out and continue executing and
then fall off the bottom of the tri-block and keep going and it all's good.
If the function throws, you're supposed to exit the current function and then
get into the accept clause, right?
And then do whatever code's there and then keep falling on and going on.
And so, the way that a compiler like Mojo works is that the call to that function,
which happens in the accept block, calls a function and then instead of
returning nothing, it actually returns, you know, a variant between nothing and an error.
And so, if you return normally, fall off the bottom or do return, you return nothing.
And if you throw an error, you return the variant that is, I'm an error, right?
So, when you get to the call, you say, okay, cool, I called a function.

Hey, I know locally I'm in a tri-block, right?
And so, I call the function and then I check to see what it returns.
Ah-ha, if it's that error thing, jump to the accept block.
And that's all done for you behind the scenes.
Exactly.
And so, the compiler does all this for you.
And I mean, one of the things, if you dig into how this stuff works in Python,
it gets a little bit more complicated because you have finally blocks,
which now you need to go into, do some stuff, and then those can also throw and return.
Wait, what, nested?
Yeah, and like the stuff matters for compatibility.
Like, there's, there's, there's, there's with clauses.
And so, with clauses are kind of like finally blocks of some special stuff going on.
And so, there's, nesting in general, nesting of anything, nesting of functions
should be illegal.
Well, it just feels like it adds a level of complexity.
Alex, I'm merely an implementer.
And so, this is again, one, one, one of, one of the, one of the trade-offs you get
when you decide to build a superset is you get to implement a full fidelity
implementation of the thing that you decided is good.
And so, yeah, I mean, we can, we can complain about the reality of the
world and shake our fists, but.
It always feels like you shouldn't be allowed to do that, like to declare
functions and sudden functions inside functions.
Oh, wait, wait, wait, what happened to Lex, the, the Lisp guy?
No, I understand that, but Lisp is what I used to do in college.
So now you've grown up.
You know, we've all done things in college.
We're not part of, no, I love, I love.
Lisp, I love Lisp, I love Lisp.
Okay, well, yeah, I was going to say, you're afraid of me.
You're taking the whole internet and playing.
Yeah, I love Lisp.
It's, it's, uh, it worked, it worked as a joke on my head and.
Yeah, yeah, so, so, so, so nested functions are joking aside, actually
really great and for certain things, right?
And so these are also called closures.
Closures are pretty cool and you can pass callbacks.
There's a lot of good patterns and so.
Uh, so speaking of which, I don't think you have, uh, nested functions
implemented yet in Mojo.
Uh, we don't have Lambda syntax, but we do have a nested functions.
Yeah, there's a few things on the roadmap they have that it'd be cool to sort
of just fly through because it's interesting to see, you know, how many

features there are in a language, small and big, they have to implement.
Yeah.
So first of all, there's tuple support and that has to do with some very specific aspects of it, like the parentheses or not parentheses, that.
Yeah, this is just a totally a syntactic thing.
A syntactic thing.
Okay, there's, but it's cool.
It's still, uh, so keyword arguments and functions.
Yeah.
So this is where in Python, you can say call a function X equals four.
Yeah.
And X is the name of the argument.
That's a nice sort of documenting self-documenting feature.
Yeah.
I mean, and again, this isn't rocket science to implement.
That's just the longer it's just on the list.
Uh, the bigger features are things like traits.
Mm hmm.
So traits are when you want to define abstract.
So when you get into typed languages, you need the ability to write generics.
And so you want to say, I want to write this function.
And now I want to work on all things that are arithmetic like.
Well, what does arithmetic like mean?
Well, arithmetic like is a categorization of a bunch of types.
And so it's, again, you can define many different ways and I'm not going to go into ring theory or something, but the, uh, you know, you can say it's arithmetic like, if you can add subtract, multiply, divide it, for example.
Right.
And so what you're saying is you're saying there's a set of traits that apply to a broad variety of types.
And so there are all these types of arithmetic, like all these tensors and floating point integer, and like there's this category of, of types.
And then I can define on an orthogonal access algorithms that then work against types that have those properties.
And so this is a, again, it's a widely known thing.
It's been implemented in Swift and Rust and many languages.
So it's not Haskell, um, which is where everybody learns, learns their tricks from, um, but the, uh, but we need to implement that and that'll enable a new level of expressivity.
Uh, so classes.
Yeah.
Classes are a big deal.
That's a big deal.
Still to be implemented.

Um, like you said, uh, Lambda syntax, and then there's like detail stuff,
like whole module import, um, support for top level code and file scope.
So, and then global variables also.
So being able to have variables outside of a top level.
Well, and so this comes back to the, where Mojo came from and the fact
that this is your point one, right?
And so we're building some modules, building an AI stack, right?
And an AI stack has a bunch of problems working with hardware and writing
high performance kernels and doing with the kernel fusion thing I was talking
about and getting the most out of the hardware.
And so we've really prioritized and built Mojo to solve modules problem.
Right now, our North Star is build out and support all the things.
And so we're making incredible progress.
By the way, Mojo is only like seven months old.
So that's another interesting thing.
I mean, part of the reason I wanted to mention some of these things is like,
there's a lot to do and it's pretty cool how you just kind of, sometimes you
take for granted how much there is in a programming language, how many
cool features you kind of rely on.
And this is kind of a nice reminder when you lay it as a to-do list.
Yeah.
And so, I mean, but also you look into, it's amazing how much is also there.
And you take it for granted that a value, if you define it, it will
get destroyed automatically.
Like that little feature itself is actually really complicated, given the way
the ownership system has to work.
And the way that works within Mojo is a huge step forward from what Russ and
Swift have done.
Can you say that again, when a value, when you define it gets destroyed automatically?
Yeah.
So like, say you have a string, right?
So you just find a string on the stack.
Okay.
Whatever that means.
Like in your local function.
Right.
And so you say, like whether it be in a def, and so you just say X equals hello world.
Right.
Well, if your string type requires you to allocate memory, then once destroyed,
you have to deallocate it.
So in Python and Mojo, you define that with the Dell method.
Right.
Where does that get run?
Well, it gets run sometime between the last use of the value and the end of the program.

Like in this, you now get into garbage collection, you get into like all these long debated, you talk about religions and, and trade-offs and things like this.
This is a hugely, hotly contested world.
If you look at C++, the way this works is that if you define a variable or a set of variables within a function, they get destroyed in a last in, first out order.
So it's like nesting.
Okay.
Um, this has a huge problem because if you define, you have a big scope and you define a whole bunch of values at the top, and then you use them and then you do a whole bunch of code that doesn't use them.
They don't get destroyed until the very end of that scope.
Right.
And so this also destroys tail calls.
So good functional programming.
Right.
This, this has a bunch of different impacts on, um, you know, you talk about reference accounting optimizations and things like this, a bunch of very low level things.
And so what Mojo does is it has a different approach on that from any language I'm familiar with where it destroys them as soon as possible.
And by doing that, you get better memory use, you get better predictability, you get tail calls that work, you get a bunch of other things, you get better ownership tracking.
There's a bunch of these very simple things that are very fundamental that are already built in there in Mojo today that are the things that nobody talks about generally, but when they don't work right, you find out and you have to complain about.
Is it trivial to know, uh, what's the soonest possible to delete a thing that's not going to be used again?
Yeah.
Well, I mean, it's generally trivial.
It's, it's after the last use of it.
So if you just find X as a string and then you have some use of X somewhere in your code within that scope, I mean, within a scope that is accessible.
It's, yeah, exactly.
So you can only use something within its scope.
And so then it doesn't wait until the end of the scope to delete it.
It, it destroys it after the last use.
So there's kind of some very eager machine that's just sitting there and deleting and it's all in the compiler.
So it's not at runtime, which is also cool.
And so the, yeah.
And so what, and this is actually non-trivial because you have control flow.
And so it gets complicated pretty quickly.
And so like getting this right was not,
Oh, so you have to insert delete like in a lot of places.

Potentially.
Yeah.
Exactly.
And so the compiler has to reason about this.
And this is where, again, it's experienced building languages and not getting this right.
So again, you get another chance to do it.
And you've got basic things like this, right?
But it's, it's extremely powerful when you do that.
Right.
And so there's a bunch of things like that that kind of combine together.
And this comes back to the, you get a chance to do it the right way, do it the right way and make sure that every brick you put down is really good.
So that when you put more bricks on top of it, they stack up to something that's beautiful.
Well, there's also like, how many design discussions do there have to be about particular details, like implementation of particular small features?
Because the features that seem small, I bet some of them might be like really require really big design decisions.
Yeah.
Well, so I mean, let me give you another example of this.
Python has a feature called async await.
So it's, it's a new feature.
I mean, in the long arc of history, it's a relatively new feature, right?
That allows way more expressive asynchronous programming.
Okay.
Again, this is, this is a, Python's a beautiful thing.
And they did things that are great for mojo for completely different reasons.
The reason the async await got added to Python, as far as I know, is because Python doesn't support threads.
Okay.
And so Python doesn't support threads, but you want to work with networking and other things like that that can block.
I mean, Python does support threads.
It's just not its strength.
And so, and so they added this feature called async await.
It's also seen in other languages like Swift and JavaScript and many other places as well.
Async await in mojo is amazing because we have a high performance heterogeneous compute runtime underneath the covers that then allows non-blocking IO.
So you get full use of your accelerator.
That's huge, turns out.
It's actually really an important part of fully utilizing the machine.
You talk about design discussions.
That took a lot of discussions, right?
And it probably will require more iteration.
And so my philosophy with mojo is that, you know, we have a small team of really good

people that are pushing forward and they're very good at the extremely deep knowing how the compiler and runtime and like all the low level stuff works together.
But they're not perfect.
Same thing as the Swift team, right?
And this is where one of the reasons we released mojo much earlier is so we can get feedback.
And we've already like renamed a keyword to the community feedback and we use an ampersand and now it's named in and out.
We're not renaming existing Python keywords because that breaks compatibility, right?
We're naming things we're adding and making sure that they are designed well.
We get usage experience.
We iterate and work with the community because, again, if you scale something really fast and everybody writes all their code and they start using it in production, then it's impossible to change.
And so you want to learn from people.
You want to iterate and work on that early on.
And this is where design discussions, it's actually quite important.
Could you incorporate an emoji into the language, into the main language?
Like a, do you have a favorite one?
Well, I really like, there's a humor like Rawful, whatever, rolling on the floor laughing.
So that could be like, what would that be, the use case for that?
Like an exception, throw an exception of some sort.
You should totally file a feature request.
Or maybe a hard one.
It has to be a hard one.
People have told me that I'm insane.
So this is, I'm liking this.
I'm going to use the viral nature of the internet to actually get this past.
I mean, it's funny you come back to the flame emoji, file extension, right?
The, you know, we have the option to use the flame emoji, which just even that concept cause, for example, the people that get up to say, now I've seen everything.
Yeah, there's something, it kind of is reinvigorating.
It's like, uh, it's like, oh, that's possible.
That's really cool.
That, for some reason, that makes everything else seem really exciting.
The world is ready for this stuff, right?
And so, you know, when we have a package manager, we'll clearly have to innovate by having the compiled package saying be the little box with the bow on it, right?
I mean, it has to be done.
It has to be done.
Is there some stuff on the roadmap that you're particularly stressed about or excited about that you're thinking about a lot?
I mean, as a today snapshot, which will be obsolete tomorrow, the lifetime stuff is really exciting.
And so lifetimes give you safe references to memory without dangling pointers.

And so this has been done in languages like Rust before.
And so we have a new approach, which is really cool.
I'm very excited about that.
That'll be out to the community very soon.
The traits feature is really a big deal.
And so that's blocking a lot of API design.
And so there's that.
I think that's really exciting.
A lot of it is these kind of table stakes features.
One of the things that is, again, also lessons learned with Swift is that programmers in general like to add syntactic sugar.
And so it's like, oh, well, this annoying thing, like in Python, you have to spell unbar unbar add, why can't I just use plus def plus?
Come on, why can't I just do that, right?
And so trivial bit of syntactic sugar, it makes sense.
It's beautiful.
It's obvious.
We're trying not to do that.
And so for two different reasons, one of which is that, again, lesson learned with Swift, Swift has a lot of syntactic sugar, which may be a good thing.
Maybe not.
I don't know.
But because it's such an easy and addictive thing to do, sugar, like make sure blood gets crazy, right?
Like the community will really dig into that and want to do a lot of that.
And I think it's very distracting from building the core abstractions.
The second is we want to be a good member of the Python community, right?
And so we want to work with the broader Python community.
And yeah, we're pushing forward a bunch of systems programming features, and we need to build them out to understand them.
But once we get a long ways forward, I want to make sure that we go back to the Python community and say, okay, let's do some design reviews.
Let's actually talk about this stuff.
Let's figure out how we want this stuff all to work together.
And syntactic sugar just makes all that more complicated.
So.
And yeah, list comprehension is like yet to be implemented.
And my favorite, I mean, dictionaries.
Yeah, there's some, there's some basic zero point one, zero point one.
Yeah, but nonetheless, it's actually so quite interesting and useful.
As you mentioned, modular is very new.
Mojo is very new.
It's a relatively small team.
Yeah.

This building up this gigantic stack.
It's incredible stack that's going to perhaps define the future of development of our AI overlords.
Uh, we just hope it will be useful as do all of us.
Uh, so what, uh, what have you learned from this process of building up a team?
Maybe one question is, how do you hire?
Yeah, great programmers, great people that operate in this compiler, hardware, machine learning, software, interface design space.
Yeah.
And maybe you're a little bit fluid in what they can do.
So, okay, so.
Language design too.
So building a company is just as interesting in different ways as building a language.
Like different skill sets, different things, but super interesting.
And I've built a lot of teams in a lot of different places.
Um, if you zoom in from the big problem into recruiting, well, so here's our problem.
Okay, I'll just, I'll be very straightforward about this.
We started modular with a lot of conviction about, we understand the problems.
We understand the customer pain points.
We need to work backwards from the suffering in the industry.
And if we solve those problems, we think it'll be useful for people.
But the problem is, is that the people we need to hire, as you say, are all these super specialized people that have jobs at big tech, big tech worlds, right?
And, you know, we, I don't think we have, um, product market fit in the way that a normal startup does, that we don't have product market fit challenges.
Because right now everybody's using AI and so many of them are suffering and they won't help.
And so again, we started with strong conviction.
Now, again, you have to hire and recruit the best and the best all have jobs.
And so what we've done is we said, okay, well, let's build an amazing culture.
Start with that.
That's usually not something a company starts with.
Usually you hire a bunch of people and then people start fighting and it turns into a gigantic mess.
And then you try to figure out how to improve your culture later.
Um, my co-founder Tim, in particular, is super passionate about making sure that that's right.
And we've spent a lot of time early on to make sure that we can scale.
Can you comment, sorry, before we get to the second, what makes for a good culture?
Um, so, I mean, there's many different cultures and I have learned many things.
Hey, you worked with several very unique, almost famously unique cultures.
And some of them I learned what to do and some of them I learned what not to do.
Okay.
And so, um, we want an inclusive culture.

Uh, I believe in like amazing people working together.
And so I've seen cultures where people, you have amazing people and they're fighting each other.
I see amazing people and they're told what to do, like doubt, shout, line up and do what I say.
It doesn't matter if it's the right thing, do it.
Right.
And neither of these is the, and I've seen people that have no direction.
They're just kind of floating in different places and they want to be amazing.
They just don't know how.
And so a lot of it starts with have a clear vision.
Right.
And so we have a clear vision of what we're doing.
And, um, so I kind of grew up at Apple in my engineering life.
Right.
And so a lot of the Apple DNA rubbed off on me.
Um, my co-founder Tim also is like a strong product guy.
And so what we learned is, you know, I decided at Apple that you don't work from building cool technology.
You don't work from like, come up with a cool product and think about the features you'll have in the big checkboxes and stuff like this.
Cause if you go talk to customers, they don't actually care about your product.
They don't care about your technology.
What they care about is their problems.
Right.
And if your product can help solve their problems, well, hey, they might be interested in that.
Right.
And so if you speak to them about their problems, if you understand and you have compassion, you understand what people are working with, then you can work backwards to building an amazing product.
So the vision starts by defining the problem.
And then you can work backwards in solving technology.
And at Apple, like it's, I think pretty famously said that, you know, for every, you know, there's a hundred no's for every yes.
I would refine that to say that there's a hundred not yets for every yes.
But, um, famously, if you go back to the iPhone, for example, right, the iPhone one, I read, I mean, many people laughed at it because it didn't have 3G.
It didn't have copy and paste.
Right.
And then a year later, okay, finally it has 3G, but it still doesn't have copy and paste. It's a joke.
Nobody will ever use this product, blah, blah, blah, blah, blah, blah, blah.
Right.

Well, your three had copy and paste and people stopped talking about it.
Right.
And so, and so being laser focused and having conviction, understanding what the core problems are and giving the team the space to be able to build the right tech is really important.
Um, also, I mean, you come back to recruiting, you have to pay well.
Right.
So we have to pay industry-leading salaries and have good benefits and things like this.
That's a big piece.
We're a remote first company.
And so we have to, uh, so remote first has a very strong set of pros and cons.
On the one hand, you can hire people from wherever they are, and you can attract amazing talent, even if they live in strange places or unusual places.
Um, on the other hand, you have time zones.
On the other hand, you have like everybody on the internet will fight if they don't understand each other.
And so we've had to learn how to, like have a system where we actually fly people in and we get the whole company together periodically.
And then we get work groups together and we plan and execute together.
And there's like an intimacy to the in-person brainstorming.
I guess you lose, but maybe you don't, maybe if you get to know each other well and you trust each other, maybe you can do that.
Well, so when the pandemic first hit, I mean, I'm curious about your experience too.
The first thing I missed was having whiteboards.
Yeah.
Right.
Those design discussions were like, I can high intensity work through things, get things done, work through the problem of the day, understand where you're on, figure out and solve the problem and move forward.
But we figured out ways to work around that now with, you know, all these screen sharing and other things like that that we do.
The thing I miss now is sitting down at a lunch table with the team.
Yeah.
The spontaneous things like those, the coffee, the coffee bar things and the, and the bumping into each other and getting to know people outside of the transactional solve a problem over zoom thing.
And I think there's, there's just a lot of stuff that I'm not an expert at this.
I don't know who is, hopefully there's some people, but there's stuff that somehow is missing on zoom, even with the whiteboard.
If you look at that, if you have a room with one person at the whiteboard and then there's like three other people at a table, there's a, first of all, there's a social aspect of that where you're just shooting the shit a little bit, almost like, yeah, as people are just kind of coming in and yeah, that,

but also while like it's a breakout discussion that happens for like
seconds at a time, maybe an inside joke or it's like this interesting dynamic
that happens that zoom, you're bonding, you're bonding, you're bonding, but
through that bonding, you get the excitement, there's certain ideas are
like complete bullshit and you'll see that in the faces of others.
That you won't see necessarily on zoom and like something, it feels like that
should be possible to do without being in person.
Well, I mean, being in person is a very different thing.
Yeah, I don't, it's worth it, but you can't always do it.
And so again, we're still learning and we're also learning as like humanity
with this new reality, right?
But, but what we found is that getting people together, whether it be a team
or the whole company or whatever is worth the expense because people work
together and are happier after that.
Like it just, it just, like there's a massive period of time where you like
go out and things start getting frayed, pull people together and then you
realize that we're all working together.
We see things the same way.
We work through the disagreement or the misunderstanding.
We're talking across each other and then you work much better together.
And so things like that, I think are really quite important.
What about people that are kind of specialized in very different aspects
of the stack working together?
What are some interesting challenges there?
Yeah.
Well, so I mean, I mean, there's lots of interesting people, as you can tell,
I'm, you know, hard to deal with too.
But you're one of the most lovable people.
So one of the, so there's different philosophies in building teams for me.
And so some people say hire 10x programmers and that's the only thing
whatever that means, right?
What I believe in is building well balanced teams.
Teams that have people that are different in them.
Like if you have all generals and no troops or all troops and no generals,
or you have all people that think in one way and not the other way, what you
get is you get a very biased and skewed and weird situation where people end
up being unhappy.
And so what I like to do is I like to build teams of people where they're
not all the same, you know, we do have teams and they're focused on like runtime
or compiler GPU or whatever the specialty is, but people bring a
different take and have a different perspective.
And I look for people that complement each other.
And particularly if you look at leadership teams and things like this, you don't
want everybody thinking the same way.

You want people bringing different perspectives and experiences.
And so I think that's really important.
That's team, but what about building a company as ambitious as modular?
So what are some interesting questions there?
Oh, I mean, so many, like, so, um, one of the things I love about, okay.
So modular is the first company I built from scratch.
Um, uh, one of the first things that was profound was I'm not cleaning up somebody else's mess, right?
And so if you look at that's liberating to some degree, it's super liberating.
And, um, and also many of the projects I've built in the past have not been core to the product of the company.
Swift is not Apple's product, right?
Uh, MLIR is not Google's revenue machine or whatever.
Right.
It's not, it's, it's important, but it's like working on the accounting software for, you know, the, the retail giant or something, right?
It's, it's, it's like enabling infrastructure and technology.
And so at modular, the tech we're building is here to solve people's problems.
Like it is directly the thing that we're giving to people.
And so this is a really big difference.
And what it means for me as a leader, but also for many of our engineers is they're working on the thing that matters.
And that's actually pretty, I mean, again, for, for a compiler people and things like that, that's, that's usually not the case, right?
And so that's, that's also pretty exciting and, and quite nice.
But the, um, one of the ways that this manifests is it makes it easier to make decisions.
And so one of the challenges I've had in other worlds is it's like, okay, well, community matters somehow for the goodness of the world, like, or open source matters theoretically, but I don't want to pay for a t-shirt, right?
Or some swag, like, well, t-shirts cost 10 bucks each.
You can have a hundred t-shirts for a thousand dollars to a mega corporate thousand dollars is uncountably, can't count that low, right?
But justifying it and getting a t-shirt, by the way, if you'd like a t-shirt.
Well, I would 100% like a t-shirt.
Are you joking?
You can have a fire emoji t-shirt.
I will, I will treasure this.
I will pass it down to my grandchildren.
And so, you know, it's, it's very liberating to be able to decide, I think that Lex should have a t-shirt, right?
And it becomes very simple because I like Lex.
This, this, uh, this is awesome.
Um, so I have to ask you about the one of the interesting

developments with large language models.
Is that they're able to generate code, uh, recently, really well.
Yes.
To a degree that, uh, maybe, uh, I don't know if you understand, but I have, I struggle to understand because it, it forces me to ask questions about the nature of programming, of the nature of thought, because the, uh, language models are able to predict the kind of code I was about to write so well, that it makes me wonder like how unique my brain is and where the valuable ideas actually come from.
Like how much do I contribute in terms of, uh, ingenuity, innovation to code.
I write or design and that kind of stuff.
Um, when you stand on the shoulders of giants, are you really doing anything?
And what LLMs are helping you do is they help you stand on the shoulders of giants in your program.
There's mistakes.
They're interesting that you learn from, but I just, it would love to get your opinion first high level of what you think about, uh, this impact of large right language models when they do programs and this is when they generate code.
Yeah.
Well, so, um, I don't know where it all goes.
Yeah.
Um, I'm an optimist and I'm a human optimist.
Right.
I think that, um, things I've seen are that a lot of the LLMs are really good at crushing lead code projects and they can reverse the link list like crazy.
Um, well, it turns out there's a lot of instances of that on the internet and it's a pretty stock thing.
And so if you want to see standard questions answered, LLMs can memorize all the answers and that can be amazing.
And also they do generalize out from that.
And so there's good work on that.
But, um, but I think that if you, in my experience, building things, building something like you talk about mojo or you talk about these things, or you talk about building an applied solution to a problem, it's also about working with people.
It's about understanding the problem.
What is the product that you want to build?
What are the use case?
What are the customers?
Can't just go survey all the customers because they'll tell you that they want to faster horse.
Maybe they need a car, right?
And so a lot of it comes into, um, you know, I don't feel like we have to compete with LLMs.
I think they'll help automate a ton of the mechanical stuff out of the way.
And just like, you know, I think we all try to scale through delegation and things

like this, delegating, wrote things to an LLM.
I think it's an extremely valuable and approach that will help us all scale and be more productive.
But I think it's a, it's a fascinating companion.
But I'd say I don't think that means that we're going to be done with coding.
But there's power in it as a companion.
And, uh, from there, I could, I would love to zoom in onto mojo a little bit.
Do you think, uh, do you think about that?
Do you think about LLMs generating mojo code?
Uh, and helping sort of, like when you design new programming language, it almost seems like, man, it would be nice to sort of, um, almost as a way to learn how I'm supposed to use this thing for them to be trained on some of the mojo code.
Well, so I do lead an AI company.
So maybe there'll be a mojo LLM at some point.
Uh, but if your question is like, how do we make a language to be suitable for LLMs?
Yeah.
I think that the, um, I think the cool thing about LLMs is you don't have to.
Right.
And so if you look at what is English or any of these other terrible languages that we as humans deal with on a continuous basis, they're never designed for machines and yet they're the intermediate representation.
They're the exchange format that we humans use to get stuff done.
Right.
And so these programming languages, they're an intermediate representation between the human and the computer or the human and the compiler roughly.
Right.
And so I think the LLMs will have no problem learning whatever keyword we pick.
Maybe the phy emoji is going to break it doesn't tokenize the reverse of that.
It will actually enable it because one of the issues I could see with being a superset of Python is there would be confusion by the gray area.
So it would be mixing stuff.
Uh, but I'm a human optimist.
I'm also an LLM optimist.
I think that we'll solve that problem.
But the, um, but, but you look at that and you say, okay, well, reducing the rote thing, right?
It turns out compilers are very particular and they really want things.
They really want the indentation to be right.
They really want the colon to be there on your else or else it'll complain.
Right.
I mean, compilers can do better at this, but, um, LLMs can totally help solve that problem.
And so I'm very happy about the new predictive coding and co-pilot type

features and things like this, because I think it'll all just make us more productive.
It's still messy and fuzzy and uncertain, unpredictable.
So, but is there a future you see given how big of a leap GPT-4 was where you start to see something like LLMs inside a compiler or no?
I mean, you could do that.
Yeah, absolutely.
I mean, I think that would be interesting.
Is that wise?
Well, I mean, it would be very expensive.
So compilers run fast and they're very efficient and LLMs are currently very expensive.
There's on-device LLMs and there's other things going on.
And so maybe there's an answer there.
Um, I think that one of the things that I haven't seen enough of is that, so LLMs to me are amazing when you tap into the creative potential of the hallucinations.
Right.
And so if you're doing creative brainstorming or creative writing or things like that, the hallucinations work in your favor.
Um, if you're writing code that has to be correct because you're going to ship it in production, then maybe that's not actually a feature.
And so I think that there, there has been research and there has been work on building algebraic reasoning systems and kind of like figuring out more things that feel like proofs.
And so I think that there could be interesting work in terms of building more reliable at scale systems and that could be interesting.
But if you chase that rabbit hole down, the question then becomes, how do you express your intent to the machine?
And so maybe you want LLMs to provide the spec, but you have a different kind of net that then actually implements the code.
Right.
So it's used it as documentation and, and inspiration versus the actual implementation.
Yeah, potentially.
Since a successful modular will be the thing that runs, I say so jokingly, our AI overlords, but AI systems that I used across, uh, I know it's a cliche term, but, uh, in and of things.
So cost, so I'll joke and say like AGI should be written in Mojo.
Yeah.
AGI should be written in Mojo.
You're joking, but it's also possible that it's not a joke.
Uh, that a lot of the ideas behind Mojo is, uh, seems like the, the natural set of ideas that would enable at scale training and inference of AI systems.
Um, so it's just, I have to ask you about the big philosophical question about human civilization.
So folks like, uh, uh, L.A.

Zaryat Kowski are really concerned about the threat of AI.
Uh, do you think about the, the good and the bad that can happen at scale deployment of AI systems?
Well, so I've, I've thought a lot about it and there's a lot of different parts to this problem, everything from job displacement to sky nut, things like this.
And so you can zoom in to sub parts of this problem.
Um, I'm not super optimistic about AGI being solved next year.
I don't think that's going to happen personally.
So you have all kind of Zen like calm about, is there's a nervousness because the leap of GPT for seemed so big.
Sure.
It's like we're almost, we're, there's some kind of transition era period.
You're thinking.
Well, so, so, so, I mean, there's a couple of things going on there.
One is, um, I'm sure GPT five and seven and 19 will be also huge leaps.
Um, they're also getting much more expensive to run.
And so there may be a limiting function in terms of just expense on one hand and train, like that, that could be a limiter that slows things down.
But I think the bigger limiter outside of like sky net takes over and I don't spend any time thinking about that because if sky net takes over and kills us all, then I'll be dead.
So I don't worry about that.
So, you know, I mean, that's just, okay, I have other things to worry about.
I'll just focus on, I'll focus and not worry about that one.
Um, but I think that the, the other thing I'd say is that AI moves quickly, but humans move slowly and we adapt slowly.
And so what I expect to happen is just like any technology diffusion, like the promise and then the application takes time to roll out.
And so I think that, um, I'm not even too worried about autonomous cars defining away all the taxi drivers.
Remember autonomy is supposed to be solved by 2020.
Yeah, I boy, do I.
So, and, um, and so like, I think that on the one hand, we can see amazing progress, but on the other hand, we can see that, uh, you know, the reality is a little bit more complicated and it may take longer to roll out than, than you might expect.
Well, that's in the physical space.
I, I do think in the digital space is, uh, the stuff that's built on top of LLMs that runs, you know, the millions of apps that could be built on top of them and that could be run on millions of devices, millions of types of devices.
I, I just think that the rapid effect it has on human civilization could be truly transformative to it.
Yeah, well, we don't even know.
Well, and so that, well, and there I think it depends on, are you an optimist or a pessimist or a masochist?

Um, uh, just to clarify, uh, optimist about human civilization.
Me too.
And so I look at that as saying, okay, cool, what will I do?
Right.
And so some people say, oh my God, is it going to destroy us all?
How do we prevent that?
I, I kind of look at it from a, is it going to unlock us all?
Right.
You talk about coding.
It's going to make, so I don't have to do all the repetitive stuff.
Well, suddenly that's a very optimistic way to look at it.
And you look at what a lot of, a lot of these technologies have done to improve our lives.
And I want that to go faster.
What do you think the future of programming looks like in the next 10, 20, 30, 50 years?
The LMS and, uh, with, with mojo, with modular, like the vision for devices, the hardware to the compilers, to this, to the different stacks of software.
Well, so what I want, I mean, coming, coming back to my arch nemesis, right?
It's complexity, right?
So again, me being the optimist, if we drive down complexity, we can make these tools, these technologies, these cool hardware widgets accessible to way more people.
Right.
And so what I'd love to see is more personalized experiences, more, uh, things, the research getting into production instead of being lost at NeurIPS, right?
And so, and like the, the, the, these things that impact people's lives by entering products.
And so one of the things that I'm a little bit concerned about is right now, um, the big companies are investing huge amounts of money and are driving the top line of AI capability for really quickly.
But if it means that you have to have a hundred million dollars to train a model or more, a hundred billion dollars, right?
Well, that's going to make it very concentrated with very few people in the world that can actually do this stuff.
I would much rather see lots of people across the industry be able to participate and use this, right?
And you look at this, you know, I mean, a lot of great research has been done in the health world and looking at, like detecting pathologies and doing radiology with AI and like doing all these things.
Well, the problem today is that to deploy and build these systems, you have to be an expert in radiology and an expert in AI.
And if we can break down the barriers so that more people can use AI techniques, and it's more like programming Python, which roughly everybody can do if they

want to, right?
Then I think that we'll get a lot more practical application of these techniques and a lot more niche year, cool, but narrower demands.
And I think that's, that's going to be really cool.
Do you think we'll have more or less programmers in the world than now?
Well, so I think we'll have more, more programmers, but they may not consider themselves to be programmers.
That'd be a different name for you, right?
I mean, do you consider somebody that uses, you know, I think that arguably the most popular programming language is Excel.
Yeah.
Right.
Yep.
And so do they consider themselves to be programmers?
Maybe not.
I mean, some of them make crazy macros and stuff like that.
But what you mentioned, Steve Jobs, it's the bicycle for the mind that allows you to go faster, right?
And so I think that as we look forward, right, what is AI?
I look at it as hopefully a new programming paradigm.
It's like object-oriented programming, right?
If you want to write a cat detector, you don't use for loops.
Turns out that's not the right tool for the job, right?
And so right now, unfortunately, because, I mean, it's not unfortunate, but it's just kind of where things are.
AI is this weird, different thing that's not integrated into programming languages and normal tool chains and all of the technology is really weird and doesn't work right.
And you have to babysit it.
And every time you switch to hardware, it's different.
It shouldn't be that way.
When you change that, when you fix that, suddenly, again, the tools, technologies can be way easier to use.
You can start using them for many more things.
And so that's, that's why I would be excited about it.
What kind of advice could you give to somebody in high school right now or maybe early college who's curious about programming and feeling like the world is changing really quickly here?
Yeah.
Well, what kind of stuff to learn?
What kind of stuff to work on?
Should they finish college?
Should they go work at a company?
Should they build a thing?

What do you think?
Well, so, I mean, one of the things I'd say is that you'll be most successful
if you work on something you're excited by.
And so don't get the book and read the book cover to cover and study
and memorize and recite and flashcard and go build something.
Like go solve a problem.
Go build the thing that you want to exist.
Go build an app.
Go build, train a model.
Like go build something and actually use it and set a goal for yourself.
And if you do that, then you'll, you know, there's a success.
There's the adrenaline rush.
There's the achievement.
There's the unlock that I think is where, you know, if you keep setting goals
and you keep doing things and building things, learning by building is really powerful.
In terms of career advice, I mean, everybody's difference
is very hard to give generalized advice.
I'll speak as, you know, a compiler nerd.
If everybody's going left, sometimes it's pretty cool to go right.
Yeah.
And so just because everybody's doing a thing, it doesn't mean
you have to do the same thing and follow the herd.
In fact, I think that sometimes the most exciting paths
for life lead to being curious about things that nobody else actually focuses on, right?
And it turns out that understanding deeply parts of the problem
that people want to take for granted makes you extremely valuable
and specialized in ways that the herd is not.
And so again, I mean, there's lots of rooms for specialization,
lots of rooms for generalists.
There's lots of room for different kinds and parts of the problem.
But, but I think that it's, you know, just because everything,
everybody's doing one thing doesn't mean you should necessarily do it.
And now the herd is using Python.
So if you want to be a rebel, go check out Mojo and help Chris
and the rest of the world fight the arch nemesis of complexity.
Cause simple is beautiful.
There you go.
Because you're an incredible person.
You've, you've been so kind to me ever since we met.
You've been extremely supportive.
I'm forever grateful for that.
Thank you for being who you are, for being legit, for being kind,
for fighting this really interesting problem of how to make
AI accessible to a huge number of people, a huge number of devices.

Yeah.
Well, so Lex, you're a pretty special person too, right?
And so I think that, you know, one of the funny things about you is that
besides being curious and pretty damn smart, you're actually willing to push on
things and you're, I think that you've got an agenda to like make the world think,
which I think is a pretty good agenda.
It's a pretty good one.
Thank you so much for talking to Chris.
Yeah, thanks Lex.
Thanks for listening to this conversation with Chris Ladner to support
this podcast.
Please check out our sponsors in the description.
And now let me leave you some words from Isaac Asimov.
I do not fear computers.
I fear the lack of them.
Thank you for listening and hope to see you next time.