MOJO — The New Age of Programming for AI

usability of Python with the performance of C. This allows to write portable code that’s faster than C and seamlessly inter-op with the Python ecosystem. These are some of the interesting features that set MOJO apart from it's competition .

PROGRESSIVE TYPES *— Leverage types for better performance and error checking.*
PORTABLE PARAMETRIC ALGORITHMS *— Leverage compile-time meta-programming to write hardware-agnostic algorithms and reduce boilerplate.*
LANGUAGE INTEGRATED AUTO-TUNING *— Automatically find the best values for your parameters to take advantage of target hardware.*

Along with these MOJO can parallel process across multiple cores unlike python that's mostly single thread distribution and it also doesn’t have a global interpreter lock like python . This makes it 35000 times faster than python .

LANGUAGES	TIME (S) *	SPEEDUP VS PYTHON
PYTHON 3.10.9	1027 s	1x
PYPY	46.1 s	22x
SCALAR C++	0.20 s	5000x
MOJO 🔥	0.03 s	35000x

Productive in a high-level language with Python-like usability and safety and predictability guarantees of systems programming languages like C and Rust. As such, Mojo was born to satisfy the needs of Modular's internal development efforts on the AI Engine.

What's an AI Engine ?

The AI Engine is a high-performance inference engine for leading AI frameworks like PyTorch and TensorFlow. The AI engine is run by these four steps as in any other :

1. **The Importer** takes a serialized model (e.g., a TensorFlow SavedModel) and converts it to a graph representation that the compiler can optimize.
2. **The Graph Optimizer** takes this graph representation and performs graph-level optimizations (e.g., constant folding, shape inference).
3. **The Kernel Generator** produces high-performance machine learning kernels tailored to individual architectures

and microarchitectures, including advanced techniques like operator fusion.

4. **The Runtime** executes the optimized kernel graph with low overhead to deliver state-of-the-art performance.

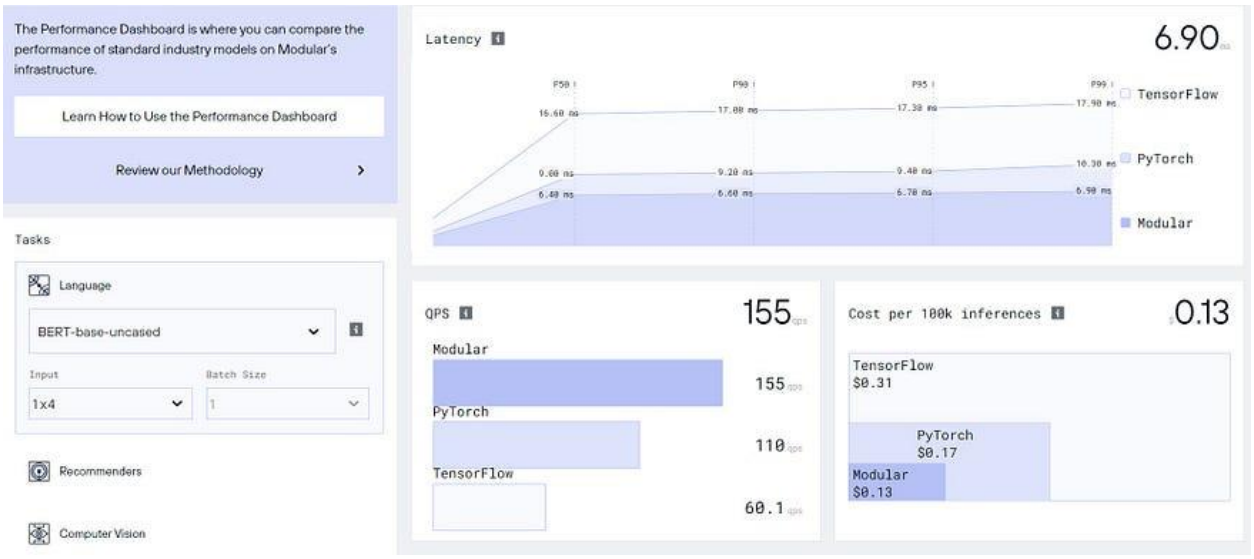
For the user, the AI Engine offers both Python and C/C++ APIs.

Modular Inference Engine

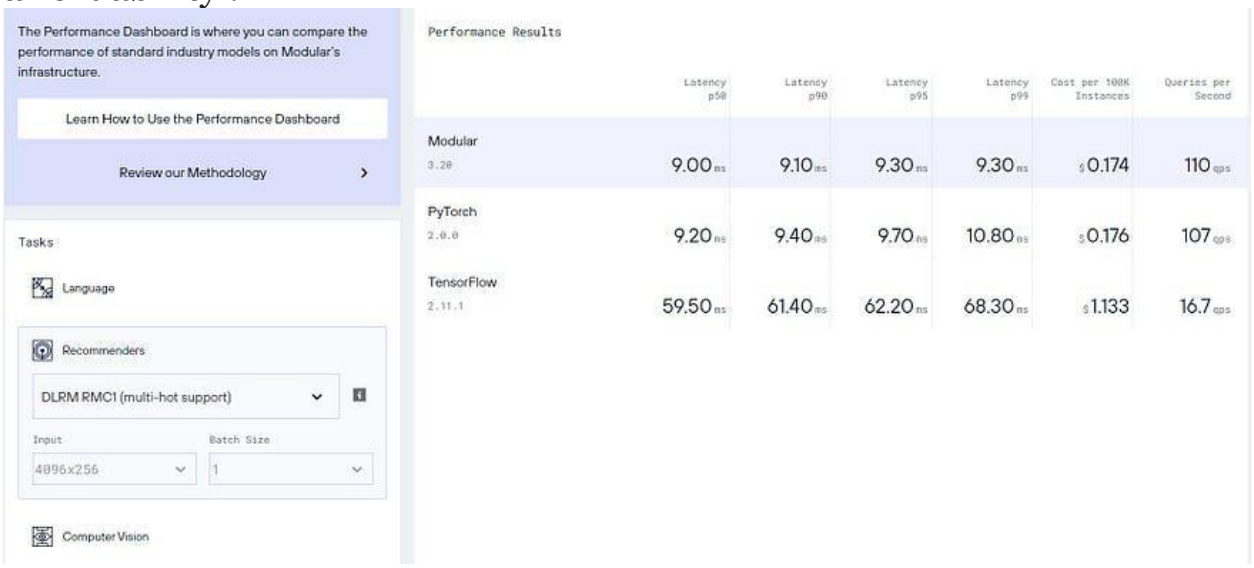
The Modular Inference Engine can help simplify workflow and reduce inference latency so scaling the AI products is efficient . It supercharges all models from TensorFlow and PyTorch, and runs on a wide variety of hardware backends.

- Requires no changes to your model.
- Delivers high performance on a wide range of hardware.
- Just load your model, execute, and watch latency drop.

They also tend to deliver superior results when used on language and recommendation models.



It's this engine that allows it to have superior performance with great affordability .



Further it can be trained in any framework and deployed through any cloud service . Reduces latency, increase throughput, and improves resource efficiency across CPUs, GPUs and accelerators. Productionize larger models and significantly reduces computing costs. Tackles

model conversion challenges easily. Seamless integration with popular libraries.

So MOJO was primarily developed as a language to power these engines, as it's faster than C++ and incorporates the best security features of RUST. MOJO is specifically built for MLIR (**Multi-Level Intermediate Representation)** compiler that is used to build reusage compiler infrastructure and reduce duplicate codes. Also letting to have access to all hardware compliers like CPU and GPU. This could be a game changer and where MOJO can create a little vital niche for itself.

Multi-Level Intermediate Representation Overview

The MLIR project is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

Weekly Public Meeting

We host a **weekly public meeting** about MLIR and the ecosystem. To be notified of the next meeting, please subscribe to the MLIR Announcements category on Discourse.

You can register to this public calendar to keep up-to-date with the schedule.

If you'd like to discuss a particular topic or have questions, please add it to the agenda doc.

The meetings are recorded and published in the talks section.

More resources

For more information on MLIR, please see:

- The MLIR section of the LLVM forums for any questions.
- Real-time discussion on the MLIR channel of the LLVM discord server.
- Previous talks.

What is MLIR for?

MLIR is intended to be a hybrid IR which can support multiple different requirements in a unified infrastructure. For example, this includes:

- The ability to represent dataflow graphs (such as in TensorFlow), including dynamic shapes, the user-extensible op ecosystem, TensorFlow variables, etc.
- Optimizations and transformations typically done on such graphs (e.g. in Grappler).
- Ability to host high-performance-computing-style loop optimizations across kernels (fusion, loop interchange, tiling, etc.), and to transform memory layouts of data.
- Code generation “lowering” transformations such as DMA insertion, explicit cache management, memory tiling, and vectorization for 1D and 2D register architectures.
- Ability to represent target-specific operations, e.g. accelerator-specific high-level operations.
- Quantization and other graph transformations done on a Deep-Learning graph.
- Polyhedral primitives.
- Hardware Synthesis Tools / HLS.

MLIR is a common IR that also supports hardware specific operations. Thus, any investment into the infrastructure surrounding MLIR (e.g. the compiler passes that work on it) should yield good returns; many targets can use that infrastructure and will benefit from it.

MLIR is a powerful representation, but it also has non-goals. We do not try to support low level machine code generation algorithms (like

register allocation and instruction scheduling). They are a better fit for lower level optimizers (such as LLVM). Also, we do not intend MLIR to be a source language that end-users would themselves write kernels in (analogous to CUDA C++). On the other hand, MLIR provides the backbone for representing any such DSL and integrating it in the ecosystem.

Compiler infrastructure

We benefited from experience gained from building other IRs (LLVM IR, XLA HLO, and Swift SIL) when building MLIR. The MLIR framework encourages existing best practices, e.g. writing and maintaining an IR spec, building an IR verifier, providing the ability to dump and parse MLIR files to text, writing extensive unit tests with the FileCheck tool, and building the infrastructure as a set of modular libraries that can be combined in new ways.

Other lessons have been incorporated and integrated into the design in subtle ways. For example, LLVM has non-obvious design mistakes that prevent a multithreaded compiler from working on multiple functions in an LLVM module at the same time. MLIR solves these problems by having limited SSA scope to reduce the use-def chains and by replacing cross-function references with explicit `symbol reference`.

Features of Mojo

Mojo has a number of features that make it a powerful and versatile programming language. Some of these features include:

Speed

Mojo is designed to be fast. It uses a number of techniques to achieve this, including:

- Mojo's gradual typing system enables superior performance and improved error checking capabilities
- A built-in autotuning system that can automatically optimize your code for the specific hardware that you are using.
- A compiler that is optimized for faster speed.

Note: Gradual typing is a type system that allows for a mix of statically typed and dynamically typed code

As a result of these techniques, Mojo can achieve speeds that are up to 35,000 times faster than Python. This makes it a great choice for developing AI models and applications that require high performance.

Efficiency

Mojo is also designed to be efficient. It uses a Gradual typing system that allows for better performance and error checking. This type of system is based on the Hindley-Milner type system, which is one of the most powerful and efficient type systems in existence.

This means that you can get the best possible performance from Mojo without having to manually tune your code.

Ease of use

Mojo is easy to use. It has a syntax that is similar to Python, so it is easy to learn for beginners. Mojo also has a number of features that make it easy to use, such as:

- A built-in debugger
- A built-in REPL (Read-Eval-Print Loop)
- A large library of documentation and tutorials

Open source

Mojo is open source. This means that it is free to use and modify. The Mojo community is very active, and there are a number of resources available to help you learn about and use Mojo.

Despite Mojo being a new language, it has the potential to be a major player in the AI space. It is fast, efficient, and easy to use, and it is backed by a strong community. If you are looking for a language that can help you build AI models and applications, then Mojo is worth checking out.

Comparison of Mojo with other programming languages

The following table compares Mojo with Python, PyPy, Scalar C++, and Mojo. The table shows the time it takes to run a simple program in each language, as well as the speedup that Mojo provides over Python.

LANGUAGES	TIME (S)	SPEEDUP VS PYTHON
PYTHON 3.10.9	1027 s	1x
PYPY	46.1 s	22x
SCALAR C++	0.20 s	5000x
MOJO 🔥	0.03 s	35000x

As you can see, Mojo is significantly faster than Python, PyPy, and Scalar C++. It is up to 35000 times faster than Python, which makes it a great choice for developing AI models and applications that require high performance.

However, it is important to note that Mojo is still under development. There may be some bugs or limitations that have not yet been discovered. If you are planning on using Mojo for a critical project, it is important to test it thoroughly before using it in production.

Here are some additional details about the languages in the table:

- Python is a popular general-purpose programming language that is known for its ease of use. However, it is not as fast as some other languages, such as C++ and Rust.
- PyPy is a Python implementation that uses a just-in-time compiler to improve performance. It is significantly faster than Python, however, it is not as fast as Mojo.
- Scalar C++ is a high-performance programming language that is known for its speed. However, it can be difficult to learn and use.
- Mojo is a new programming language that is designed to be fast, efficient, and easy to use, still under development.

Mojo🔥 Vs. Python:

Mojo and Python are two popular programming languages that are used for a variety of tasks, including data science, machine learning, and artificial intelligence. Both languages have their own strengths and weaknesses, so it is important to understand the differences between them before choosing one to use for a project.

Mojo🔥

- Mojo is a recently introduced programming language that prioritizes speed, efficiency, and user-friendly features.
- Mojo supports parallel processing across multiple cores, which makes it an ideal language for building AI and machine learning applications.
- Mojo is a superset of Python, which means that it supports all of the features of Python and more.

- Mojo has the potential to disrupt the industry. It has already gained some popularity amongst developers and is being used in a growing number. If Mojo continues to develop and improve, it could become a major player in the programming language landscape.

Python

- Python is a general-purpose programming language that is widely used for a variety of tasks, including data science, machine learning, and artificial intelligence.
- Python is known for its simplicity and readability, which makes it a popular choice for beginners and experienced developers alike.
- Python struggles with parallel processing, while Mojo excels at it, which can lead to performance issues for tasks that require a lot of processing power.
- Python is a mature language with a large ecosystem of libraries and tools.

Here is a table that summarizes the key differences between Mojo and Python:

Feature	Mojo🔥	Python
Parallel processing	Yes	No
Simplicity	Easy to learn and use	Easy to learn and use
Readability	Easy to read and understand	Easy to read and understand
Speed	Faster than Python	Slower than Mojo
Flexibility	More flexible than Python	Less flexible than Mojo
Ecosystem	Rich ecosystem of libraries	Rich ecosystem of libraries

Ultimately, the best language for you will depend on your specific needs and requirements. If you are looking for a language that is well-suited for parallel processing, then Mojo is a better choice. However, if you are looking for an easy-to-learn language, both Python and Mojo are great options.

Use Cases of Mojo🔥

Mojo for AI Development

Mojo is a new programming language that is designed to be a powerful tool for AI development. It is fast, efficient, and easy to use, and it supports parallel processing across multiple cores. This makes it possible to build AI applications that are significantly faster than those built in other languages.

Where to Use Mojo

Mojo can be used to build a wide variety of AI applications, including:

• Image recognition	• Desktop applications
• Natural language processing	• Mobile applications
• Speech recognition	• Data Science
• Reinforcement learning	• Machine learning
• Web applications	• Artificial intelligence

Mojo Examples: Real-world Applications of Mojo

Explore a range of real-world examples showcasing the versatile applications of Mojo in the field of AI development:

Image Recognition: Mojo facilitates the development of advanced image recognition applications capable of identifying objects in images. Applications built with Mojo can successfully recognize faces, cars, and various other objects.

Natural Language Processing: Mojo empowers the creation of natural language processing applications that understand and process human

language. With Mojo, developers can build applications for language translation, text summarization, and question-answering.

Speech Recognition: Mojo enables the development of speech recognition applications that convert spoken language into text. Applications leveraging Mojo can accurately transcribe audio recordings and generate text transcripts.

Reinforcement Learning: Mojo supports the creation of reinforcement learning applications that train AI agents to perform diverse tasks. Developers can utilize Mojo to build applications that teach AI agents to play games, drive vehicles, and make decisions.

Web Applications: Mojo facilitates the development of web applications accessible to users on the internet. Developers can use Mojo to build e-commerce websites, social media platforms, and other web-based applications.

Desktop Applications: Mojo empowers the creation of desktop applications that run on personal computers. Developers can leverage Mojo to build word processors, spreadsheets, and other desktop-based applications.

Mobile Applications: Mojo enables the development of mobile applications for smartphones and tablets. Developers can utilize Mojo to build games, productivity apps, and other mobile-based applications.

Data Science: Mojo supports the creation of data science applications for analyzing and interpreting large volumes of data. Developers can use Mojo to build applications that predict customer behavior, detect fraud, and make data-driven decisions.

AI and Machine Learning: Mojo is a programming language that supports the development of machine learning and artificial intelligence applications. It can be used to build applications that learn and improve over time, as well as applications that perform tasks associated with human intelligence.

These examples highlight the diverse range of applications where Mojo can be effectively utilized. As Mojo continues to evolve, we can expect the emergence of even more innovative and exciting use cases, harnessing the power of this advanced language model.

If you're looking for AI and machine learning services, we can help. Our team specializes in developing intelligent algorithms, predictive models, and data-driven decision-making systems. Visit our AI & Machine Learning Services page to learn more.

End Note

Mojo is a powerful new programming language that is ideal for AI development. It is fast, efficient, easy to use, and flexible, making it a great choice for building a wide variety of AI applications. As Mojo continues to evolve, we can anticipate the emergence of even more innovative and exciting use cases leveraging this powerful language.

We, at Seaflux, are AI & Machine Learning enthusiasts, who are helping enterprises worldwide. Have a query or want to discuss AI projects where Mojo and Python can be leveraged? Schedule a meeting with us here, we'll be happy to talk to you.

Why Mojo🔥

When we started Modular, we had no intention of building a new programming language. But as we were building our platform to unify the world's ML/AI infrastructure, we realized that programming across the entire stack was too complicated. Plus, we were writing a lot of MLIR by hand and not having a good time.

What we wanted was an innovative and scalable programming model that could target accelerators and other heterogeneous systems that are pervasive in the AI field. This meant a programming language with powerful compile-time metaprogramming, integration of adaptive compilation techniques, caching

throughout the compilation flow, and other features that are not supported by existing languages.

And although accelerators are important, one of the most prevalent and sometimes overlooked "accelerators" is the host CPU. Nowadays, CPUs have lots of tensor-core-like accelerator blocks and other AI acceleration units, but they also serve as the "fallback" for operations that specialized accelerators don't handle, such as data loading, pre- and post-processing, and integrations with foreign systems. So it was clear that we couldn't lift AI with just an "accelerator language" that worked with only specific processors.

Applied AI systems need to address all these issues, and we decided there was no reason it couldn't be done with just one language. Thus, Mojo was born.

A language for next-generation compiler technology

When we realized that no existing language could solve the challenges in AI compute, we embarked on a first-principles rethinking of how a programming language should be designed and implemented to solve our problems. Because we require high-performance support for a wide variety of accelerators, traditional compiler technologies like LLVM and GCC were not suitable (and any languages and tools based on them would not suffice). Although they support a wide range of CPUs and some commonly used GPUs, these compiler technologies were designed decades ago and are unable to fully support modern chip architectures. Nowadays, the standard technology for specialized machine learning accelerators is MLIR.

MLIR is a relatively new open-source compiler infrastructure started at Google (whose leads moved to Modular) that has been widely adopted across the machine learning accelerator community. MLIR's strength is its ability to build *domain specific* compilers, particularly for weird domains that aren't traditional CPUs and GPUs, such as AI ASICS, quantum computing systems, FPGAs, and custom silicon.

Given our goals at Modular to build a next-generation AI platform, we were already using MLIR for some of our infrastructure, but we didn't have a programming language that could unlock MLIR's full potential across our stack. While many other projects now use MLIR, Mojo is the first major language designed expressly *for MLIR*, which makes Mojo uniquely powerful when writing systems-level code for AI workloads.

A member of the Python family

Our core mission for Mojo includes innovations in compiler internals and support for current and emerging accelerators, but don't see any need to innovate in language *syntax* or *community*. So we chose to embrace the Python ecosystem because it is so widely used, it is loved by the AI ecosystem, and because we believe it is a really nice language.

The Mojo language has lofty goals: we want full compatibility with the Python ecosystem, we want predictable low-level performance and low-level control, and we need the ability to deploy subsets of code to accelerators. Additionally, we don't want to create a fragmented software ecosystem—we don't want Python users who adopt Mojo to draw comparisons to the painful migration from Python 2 to 3. These are no small goals!

Fortunately, while Mojo is a brand-new code base, we aren't really starting from scratch conceptually. Embracing Python massively simplifies our design efforts, because most of the syntax is already specified. We can instead focus our efforts on building Mojo's compilation model and systems programming features. We also benefit from tremendous lessons learned from other languages (such as Rust, Swift, Julia, Zig, Nim, etc.), from our prior experience migrating developers to new compilers and languages, and we leverage the existing MLIR compiler ecosystem.

Further, we decided that the right *long-term goal* for Mojo is to provide a **superset of Python** (that is, to make Mojo compatible with existing Python programs) and to embrace the CPython implementation for long-tail ecosystem support. If you're a Python programmer, we hope that Mojo is immediately familiar, while also providing new tools to develop safe and performant systems-level code that would otherwise require C and C++ below Python.

We aren't trying to convince the world that "static is best" or "dynamic is best." Rather, we believe that both are good when used for the right applications, so we designed Mojo to allow you, the programmer, to decide when to use static or dynamic.

Why we chose Python

Python is the dominant force in ML and countless other fields. It's easy to learn, known by important cohorts of programmers, has an amazing community, has tons of valuable packages, and has a wide variety of good tooling. Python supports the development of beautiful and expressive APIs through its dynamic programming features, which led machine learning frameworks like TensorFlow and PyTorch to embrace Python as a frontend to their high-performance runtimes implemented in C++.

For Modular today, Python is a non-negotiable part of our API surface stack—this is dictated by our customers. Given that everything else in our stack is negotiable, it stands to reason that we should start from a "Python-first" approach.

More subjectively, we believe that Python is a beautiful language. It's designed with simple and composable abstractions, it eschews needless punctuation that is redundant-in-practice with indentation, and it's built with powerful (dynamic) metaprogramming features. All of which provide a runway for us to extend the language to what we need at Modular. We hope that people in the Python ecosystem see our direction for Mojo as taking Python ahead to the next level—completing it—instead of competing with it.

Compatibility with Python

We plan for full compatibility with the Python ecosystem, but there are actually two types of compatibility, so here's where we currently stand on them both:

- In terms of your ability to import existing Python modules and use them in a Mojo program, Mojo is 100% compatible because we use CPython for interoperability.
- In terms of your ability to migrate any Python code to Mojo, it's not fully compatible yet. Mojo already supports many core features from Python, including async/await, error handling, variadics, and so on. However, Mojo is still young and missing many other features from Python. Mojo doesn't even support classes yet!

There is a lot of work to be done, but we're confident we'll get there, and we're guided by our team's experience building other major technologies with their own compatibility journeys:

- The journey to the Clang compiler (a compiler for C, C++, Objective-C, CUDA, OpenCL, and others), which is a "compatible replacement" for GCC, MSVC and other existing compilers. It is hard to make a direct comparison, but the complexity of the Clang problem appears to be an order of magnitude bigger than implementing a compatible replacement for Python.
- The journey to the Swift programming language, which embraced the Objective-C runtime and language ecosystem, and progressively migrated millions of programmers (and huge amounts of code). With Swift, we learned lessons about how to be "run-time compatible" and cooperate with a legacy runtime.

In situations where you want to mix Python and Mojo code, we expect Mojo to cooperate directly with the CPython runtime and have similar support for integrating with CPython classes and objects without having to compile the code itself. This provides plug-in compatibility with a massive ecosystem of existing code, and it enables a progressive migration approach in which incremental migration to Mojo yields incremental benefits.

Overall, we believe that by focusing on language design and incremental progress towards full compatibility with Python, we will get where we need to be in time.

However, it's important to understand that when you write pure Mojo code, there is nothing in the implementation, compilation, or runtime that uses any existing Python technologies. On its own, it is an entirely new language with an entirely new compilation and runtime system.

Intentional differences from Python

While Python compatibility and migratability are key to Mojo's success, we also want Mojo to be a first-class language (meaning that it's a standalone language rather than dependent upon another language). It should not be limited in its ability to introduce new keywords or grammar productions merely to maintain compatibility. As such, our approach to compatibility is two-fold:

1. We utilize CPython to run all existing Python 3 code without modification and use its runtime, unmodified, for full compatibility with the entire ecosystem. Running code this way provides no benefit from Mojo, but the sheer existence and availability of this ecosystem will rapidly accelerate the bring-up of Mojo, and leverage the fact that Python is really great for high-level programming already.

2. We will provide a mechanical migration tool that provides very good compatibility for people who want to migrate code from Python to Mojo. For example, to avoid migration errors with Python code that uses identifier names that match Mojo keywords, Mojo provides a backtick feature that allows any keyword to behave as an identifier.

Together, this allows Mojo to integrate well in a mostly-CPython world, but allows Mojo programmers to progressively move code (a module or file at a time) to Mojo. This is a proven approach from the Objective-C to Swift migration that Apple performed.

It will take some time to build the rest of Mojo and the migration support, but we are confident that this strategy allows us to focus our energies and avoid distractions. We also think the relationship with CPython can build in both directions—wouldn't it be cool if the CPython team eventually reimplemented the interpreter in Mojo instead of C? 🔥

Python's problems

By aiming to make Mojo a superset of Python, we believe we can solve many of Python's existing problems.

Python has some well-known problems—most obviously, poor low-level performance and CPython implementation details like the global interpreter lock (GIL), which makes Python single-threaded. While there are many active projects underway to improve these challenges, the issues brought by Python go deeper and are particularly impactful in the AI field. Instead of talking about those technical limitations in detail, we'll talk about their implications here in 2023.

Note that everywhere we refer to Python in this section is referring to the CPython implementation. We'll talk about other implementations later.

The two-world problem

For a variety of reasons, Python isn't suitable for systems programming. Fortunately, Python has amazing strengths as a glue layer, and low-level bindings to C and C++ allow building libraries in C, C++ and many other languages with better performance characteristics. This is what has enabled things like NumPy, TensorFlow, PyTorch, and a vast number of other libraries in the ecosystem.

Unfortunately, while this approach is an effective way to build high-performance Python libraries, it comes with a cost: building these hybrid libraries is very complicated. It requires low-level understanding of the internals of CPython, requires knowledge of C/C++ (or other) programming (undermining one of the original goals of using Python in the first place), makes it difficult to evolve large frameworks, and (in the case of ML) pushes the world towards "graph based" programming models, which have worse fundamental usability than "eager mode" systems. Both TensorFlow and PyTorch have faced significant challenges in this regard.

Beyond the fundamental nature of how the two-world problem creates system complexity, it makes everything else in the ecosystem more complicated. Debuggers generally can't step across Python and C code, and those that can aren't widely accepted. It's painful that the Python

package ecosystems has to deal with C/C++ code in addition to Python. Projects like PyTorch, with significant C++ investments, are intentionally trying to move more of their codebase to Python because they know it gains usability.

The three-world and N-world problem

The two-world problem is commonly felt across the Python ecosystem, but things are even worse for developers of machine learning frameworks. AI is pervasively accelerated, and those accelerators use bespoke programming languages like CUDA. While CUDA is a relative of C++, it has its own special problems and limitations, and it does not have consistent tools like debuggers or profilers. It is also effectively locked into a single hardware maker.

The AI world has an incredible amount of innovation on the hardware front, and as a consequence, complexity is spiraling out of control. There are now several attempts to build limited programming systems for accelerators (OpenCL, Sycl, OneAPI, and others). This complexity explosion is continuing to increase and none of these systems solve the fundamental fragmentation in the tools and ecosystem that is hurting the industry so badly—they're *adding to the fragmentation*.

Mobile and server deployment

Another challenge for the Python ecosystem is deployment. There are many facets to this, including how to control dependencies, how to deploy hermetically compiled "a.out" files, and how to improve multi-threading and performance. These are areas where we would like to see the Python ecosystem take significant steps forward.

Related work

We are aware of many other efforts to improve Python, but they do not solve the fundamental problem we aim to solve with Mojo.

Some ongoing efforts to improve Python include work to speed up Python and replace the GIL, to build languages that look like Python but are subsets of it, and to build embedded domain-specific languages (DSLs) that integrate with Python but which are not first-class languages. While we cannot provide an exhaustive list of all the efforts, we can talk about some challenges faced in these projects, and why they don't solve the problems that Mojo does.

Improving CPython and JIT compiling Python

Recently, the community has spent significant energy on improving CPython performance and other implementation issues, and this is showing huge results. This work is fantastic because it incrementally improves the current CPython implementation. For example, Python 3.11 has increased performance 10-60% over Python 3.10 through internal improvements, and Python 3.12 aims to go further with a trace optimizer. Many other projects are attempting to tame the GIL, and projects like PyPy (among many others) have used JIT compilation and tracing approaches to speed up Python.

While we are fans of these great efforts, and feel they are valuable and exciting to the community, they unfortunately do not satisfy our needs at Modular, because they do not help

provide a unified language onto an accelerator. Many accelerators these days support very limited dynamic features, or do so with terrible performance. Furthermore, systems programmers don't seek only "performance," but they also typically want a lot of **predictability and control** over how a computation happens.

We are looking to eliminate the need to use C or C++ within Python libraries, we seek the highest performance possible, and we cannot accept dynamic features at all in some cases. Therefore, these approaches don't help.

Python subsets and other Python-like languages

There are many attempts to build a "deployable" Python, such as TorchScript from the PyTorch project. These are useful because they often provide low-dependency deployment solutions and sometimes have high performance. Because they use Python-like syntax, they can be easier to learn than a novel language.

On the other hand, these languages have not seen wide adoption—because they are a subset of Python, they generally don't interoperate with the Python ecosystem, don't have fantastic tooling (such as debuggers), and often change-out inconvenient behavior in Python unilaterally, which breaks compatibility and fragments the ecosystem further. For example, many of these change the behavior of simple integers to wrap instead of producing Python-compatible math.

The challenge with these approaches is that they attempt to solve a weak point of Python, but they aren't as good at Python's strong points. At best, these can provide a new alternative to C and C++, but without solving the dynamic use-cases of Python, they cannot solve the "two world problem." This approach drives fragmentation, and incompatibility makes *migration* difficult to impossible—recall how challenging it was to migrate from Python 2 to Python 3.

Python supersets with C compatibility

Because Mojo is designed to be a superset of Python with improved systems programming capabilities, it shares some high-level ideas with other Python supersets like Pyrex and Cython. Like Mojo, these projects define their own language that also support the Python language. They allow you to write more performant extensions for Python that interoperate with both Python and C libraries.

These Python supersets are great for some kinds of applications, and they've been applied to great effect by some popular Python libraries. However, they don't solve Python's two-world problem and because they rely on CPython for their core semantics, they can't work without it, whereas Mojo uses CPython only when necessary to provide compatibility with existing Python code. Pure Mojo code does not use any pre-existing runtime or compiler technologies, it instead uses an MLIR-based infrastructure to enable high-performance execution on a wide range of hardware.

Embedded DSLs in Python

Another common approach is to build an embedded domain-specific languages (DSLs) in Python, typically installed with a Python decorator. There are many examples of this (the `@tf.function` decorator in TensorFlow, the `@triton.jit` in OpenAI's Triton

programming model, etc.). A major benefit of these systems is that they maintain compatibility with the Python ecosystem of tools, and integrate natively into Python logic, allowing an embedded mini language to co-exist with the strengths of Python for dynamic use cases.

Unfortunately, the embedded mini-languages provided by these systems often have surprising limitations, don't integrate well with debuggers and other workflow tooling, and do not support the level of native language integration that we seek for a language that unifies heterogeneous compute and is the primary way to write large-scale kernels and systems.

With Mojo, we hope to move the usability of the overall system forward by simplifying things and making it more consistent. Embedded DSLs are an expedient way to get demos up and running, but we are willing to put in the additional effort and work to provide better usability and predictability for our use-case.

Mojo🔥 FAQ

We tried to anticipate your questions about Mojo on this page. If this page doesn't answer all your questions, also check out our Mojo community channels.

Motivation

Why did you build Mojo?

We built Mojo to solve an internal challenge at Modular, and we are using it extensively in our systems such as our AI Engine. As a result, we are extremely committed to its long term success and are investing heavily in it. Our overall mission is to unify AI software and we can't do that without a unified language that can scale across the AI infrastructure stack. That said, we don't plan to stop at AI—the north star is for Mojo to support the whole gamut of general-purpose programming over time. For a longer answer, read Why Mojo.

Why is it called Mojo?

Mojo means "a magical charm" or "magical powers." We thought this was a fitting name for a language that brings magical powers to Python, including unlocking an innovative programming model for accelerators and other heterogeneous systems pervasive in AI today.

Why does mojo have the 🔥 file extension?

We paired Mojo with fire emoji 🔥 as a fun visual way to impart onto users that Mojo empowers them to get their Mojo on—to develop faster and more efficiently than ever before. We also believe that the world can handle a unicode extension at this point, but you can also just use the `.mojo` extension. :)

What problems does Mojo solve that no other language can?

Mojo combines the usability of Python with the systems programming features it's missing. We are guided more by pragmatism than novelty, but Mojo's use of MLIR allows it to scale to new exotic hardware types and domains in a way that other languages haven't demonstrated (for an example of Mojo talking directly to MLIR, see our low-level IR in Mojo notebook). It also includes autotuning, and has caching and distributed compilation built into its core. We also believe Mojo has a good chance of unifying hybrid packages in the broader Python community.

What kind of developers will benefit the most from Mojo?

Mojo's initial focus is to bring programmability back to AI, enabling AI developers to customize and get the most out of their hardware. As such, Mojo will primarily benefit researchers and other engineers looking to write high-performance AI operations. Over time, Mojo will become much more interesting to the general Python community as it grows to be a superset of Python. We hope this will help lift the vast Python library ecosystem and empower more traditional systems developers that use C, C++, Rust, etc.

Why build upon Python?

Effectively, all AI research and model development happens in Python today, and there's a good reason for this! Python is a powerful high-level language with clean, simple syntax and a massive ecosystem of libraries. It's also one of the world's most popular programming languages, and we want to help it become even better. At Modular, one of our core principles is meeting customers where they are—our goal is not to further fragment the AI landscape but to unify and simplify AI development workflows.

Why not enhance CPython (the major Python implementation) instead?

We're thrilled to see a big push to improve CPython by the existing community, but our goals for Mojo (such as to deploy onto GPUs and other accelerators) need a fundamentally different architecture and compiler approach underlying it. CPython is a significant part of our compatibility approach and powers our Python interoperability.

Why not enhance another Python implementation (like Codon, PyPy, etc)?

Codon and PyPy aim to improve performance compared to CPython, but Mojo's goals are much deeper than this. Our objective isn't just to create "a faster Python," but to enable a whole new layer of systems programming that includes direct access to accelerated hardware, as outlined in Why Mojo. Our technical implementation approach is also very different, for example, we are not relying on heroic compiler and JIT technologies to "devirtualize" Python.

Furthermore, solving big challenges for the computing industry is hard and requires a fundamental rethinking of the compiler and runtime infrastructure. This drove us to build an entirely new approach and we're willing to put in the time required to do it properly (see our blog post about building a next-generation AI platform), rather than tweaking an existing system that would only solve a small part of the problem.

Why not make Julia better?

We think Julia is a great language and it has a wonderful community, but Mojo is completely different. While Julia and Mojo might share some goals and look similar as an easy-to-use and

high-performance alternative to Python, we're taking a completely different approach to building Mojo. Notably, Mojo is Python-first and doesn't require existing Python developers to learn a new syntax.

Mojo also has a bunch of technical advancements compared to Julia, simply because Mojo is newer and we've been able to learn from Julia (and from Swift, Rust, C++ and many others that came before us). For example, Mojo takes a different approach to memory ownership and memory management, it scales down to smaller envelopes, and is designed with AI and MLIR-first principles (though Mojo is not only for AI).

That said, we also believe there's plenty of room for many languages and this isn't an OR proposition. If you use and love Julia, that's great! We'd love for you to try Mojo and if you find it useful, then that's great too.

Functionality

Where can I learn more about Mojo's features?

The best place to start is the Mojo programming manual, which is very long, but it covers all the features we support today. And if you want to see what features are coming in the future, take a look at the roadmap.

What does it mean that Mojo is designed for MLIR?

MLIR provides a flexible infrastructure for building compilers. It's based upon layers of intermediate representations (IRs) that allow for progressive lowering of any code for any hardware, and it has been widely adopted by the hardware accelerator industry since its first release. Although you can use MLIR to create a flexible and powerful compiler for any programming language, Mojo is the world's first language to be built from the ground up with MLIR design principles. This means that Mojo not only offers high-performance compilation for heterogeneous hardware, but it also provides direct programming support for the MLIR intermediate representations. For a simple example of Mojo talking directly to MLIR, see our low-level IR in Mojo notebook.

Is Mojo only for AI or can it be used for other stuff?

Mojo is a general purpose programming language. We use Mojo at Modular to develop AI algorithms, but as we grow Mojo into a superset of Python, you can use it for other things like HPC, data transformations, writing pre/post processing operations, and much more. For examples of how Mojo can be used for other general programming tasks, see our Mojo examples.

Is Mojo interpreted or compiled?

Mojo supports both just-in-time (JIT) and ahead-of-time (AOT) compilation. In either a REPL environment or Jupyter notebook, Mojo is JIT'd. However, for AI deployment, it's important that Mojo also supports AOT compilation instead of having to JIT compile everything. You can compile your Mojo programs using the `mojo` CLI.

How does Mojo compare to Triton Lang?

Triton Lang is a specialized programming model for one type of accelerator, whereas Mojo is a more general language that will support more architectures over time and includes a debugger, a full tool suite, etc. For more about embedded domain-specific languages (EDSLs) like Triton, read the "Embedded DSLs in Python" section of Why Mojo.

How does Mojo help with PyTorch and TensorFlow acceleration?

Mojo is a general purpose programming language, so it has no specific implementations for ML training or serving, although we use Mojo as part of the overall Modular AI stack. The Modular AI Engine, for example, supports deployment of PyTorch and TensorFlow models, while Mojo is the language we use to write the engine's in-house kernels.

Does Mojo support distributed execution?

Not alone. You will need to leverage the Modular AI Engine for that. Mojo is one component of the Modular stack that makes it easier for you to author highly performant, portable kernels, but you'll also need a runtime (or "OS") that supports graph level transformations and heterogeneous compute.

Will Mojo support web deployment (such as Wasm or WebGPU)?

We haven't prioritized this functionality yet, but there's no reason Mojo can't support it.

How do I convert Python programs or libraries to Mojo?

Mojo is still early and not yet a Python superset, so only simple programs can be brought over as-is with no code changes. We will continue investing in this and build migration tools as the language matures.

What about interoperability with other languages like C/C++?

Yes, we want to enable developers to port code from languages other than Python to Mojo as well. We expect that due to Mojo's similarity to the C/C++ type systems, migrating code from C/C++ should work well and it's in our roadmap.

How does Mojo support hardware lowering?

Mojo leverages LLVM-level dialects for the hardware targets it supports, and it uses other MLIR-based code-generation backends where applicable. This also means that Mojo is easily extensible to any hardware backend. For more information, read about our vision for pluggable hardware.

How does Mojo autotuning work?

For details about what autotuning capabilities we support so far, check out the programming manual. But stay tuned for more details!

Who writes the software to add more hardware support for Mojo?

Mojo provides all the language functionality necessary for anyone to extend hardware support. As such, we expect hardware vendors and community members will contribute additional

hardware support in the future. We'll share more details about opening access to Mojo in the future, but in the meantime, you can read more about our hardware extensibility vision.

How does Mojo provide a 35,000x speed-up over Python?

Modern CPUs are surprisingly complex and diverse, but Mojo enables systems-level optimizations and flexibility that unlock the features of any device in a way that Python cannot. So the hardware matters for this sort of benchmark, and for the Mandelbrot benchmarks we show in our launch keynote, we ran them on an AWS r7iz.metal-16xl machine.

For lots more information, check out our 3-part blog post series about how Mojo gets a 35,000x speedup over Python.

By the way, all the kernels that power the Modular AI Engine are written in Mojo. We also compared our matrix multiplication implementation to other state-of-the-art implementations that are usually written in assembly. To see the results, see our blog post about unified matrix multiplication.

Performance

Mojo's matmul performance in the notebook doesn't seem that great. What's going on?

The Mojo Matmul notebook uses matrix multiplication to show off some Mojo features in a scenario that you would never attempt in pure Python. So that implementation is like a "toy" matmul implementation and it doesn't measure up to the state of the art.

Plus, if you're using the Mojo Playground, that VM environment is not set up for stable performance evaluation. Modular has a separate matmul implementation written in Mojo (and used by the Modular AI Engine) that is not available with this release, but you can read about it in this blog post.

Are there any AI related performance benchmarks for Mojo?

It's important to remember that Mojo is a general-purpose programming language, and any AI-related benchmarks will rely heavily upon other framework components. For example, our in-house kernels for the Modular AI Engine are all written in Mojo and you can learn more about our kernel performance in our matrix multiplication blog post. For details about our end-to-end model performance relative to the latest releases of TensorFlow and PyTorch, check out our performance dashboard.

Mojo SDK

How can I get access to the SDK?

You can get the Mojo SDK here!

Is the Mojo Playground still available?

Yes, you can get access today to the Mojo Playground, a hosted set of Mojo-supported Jupyter notebooks.

What does the Mojo SDK ship with?

The Mojo SDK includes the Mojo standard library and `mojo` command-line tool, which provides a REPL similar to the `python` command, along with `build`, `run`, `package`, `doc` and `format` commands. We've also published a Mojo language extension for VS Code.

What operating systems are supported?

Currently, x86-64 Ubuntu 20.04/22.04 is supported, and support for Windows and Mac will follow. Until then, you can use WSL on Windows, and you can use Docker on Intel Macs. For all other systems you can install the SDK on a remote Linux machine—our setup guide includes instructions on how to set this up.

Is there IDE Integration?

Yes, we've published an official Mojo language extension for VS Code.

The extension supports various features including syntax highlighting, code completion, formatting, hover, etc. It works seamlessly with remote-ssh and dev containers to enable remote development in Mojo.

Does the Mojo SDK collect telemetry?

Yes, in combination with the Modular CLI tool, the Mojo SDK collects some basic system information and crash reports that enable us to identify, analyze, and prioritize Mojo issues.

Mojo is still in its early days, and this telemetry is crucial to help us quickly identify problems and improve Mojo. Without this telemetry, we would have to rely on user-submitted bug reports, and in our decades of building developer products, we know that most people don't bother. Plus, a lot of product issues are not easily identified by users or quantifiable with individual bug reports. The telemetry provides us the insights we need to build Mojo into a premier developer product.

Of course, if you don't want to share this information with us, you can easily opt-out of all telemetry, using the `modular` CLI. To stop sharing system information, run this:

```
modular config-set telemetry.enabled=false
```

To stop sharing crash reports, run this:

```
modular config-set crash_reporting.enabled=false
```

Versioning & compatibility

What's the Mojo versioning strategy?

Mojo is still in early development and not at a 1.0 version yet. It's still missing many foundational features, but please take a look at our roadmap to understand where things are headed. As such, the language is evolving rapidly and source stability is not guaranteed.

How often will you be releasing new versions of Mojo?

Mojo development is moving fast and we are regularly releasing updates. Please join the Mojo Discord channel for notifications and sign up for our newsletter for more coarse-grain updates.

Mojo Playground

What sort of computer is backing each instance in the Mojo Playground?

The Mojo Playground runs on a fleet of AWS EC2 C6i (c6i.8xlarge) instances that is divided among active users. Due to the shared nature of the system, the number of vCPU cores provided to your session may vary. We guarantee 1 vCPU core per session, but that may increase when the total number of active users is low.

Each user also has a dedicated volume in which you can save your own files that persist across sessions.

Open Source

Will Mojo be open-sourced?

Over time we expect to open-source core parts of Mojo, such as the standard library. However, Mojo is still young, so we will continue to incubate it within Modular until more of its internal architecture is fleshed out. We don't have an established plan for open-sourcing yet.

Why not develop Mojo in the open from the beginning?

Mojo is a big project and has several architectural differences from previous languages. We believe a tight-knit group of engineers with a common vision can move faster than a community effort. This development approach is also well-established from other projects that are now open source (such as LLVM, Clang, Swift, MLIR, etc.).

Community

Where can I ask more questions or share feedback?

If you have questions about upcoming features or have suggestions for the language, be sure you first read the Mojo roadmap, which provides important information about our current priorities and links to our GitHub channels where you can report issues and discuss new features.

To get in touch with the Mojo team and developer community, use the resources on our Mojo community page.

Can I share Mojo code from the Mojo Playground?

Yes! You're welcome and encouraged to share your Mojo code any way you like. We've added a feature in the Mojo Playground to make this easier, and you can learn more in the Mojo Playground by opening the `help` directory in the file browser.

However, the Mojo SDK is also now available, so you can also share `.mojo` source files and `.ipynb` notebooks to run locally!

Has anyone heard if Django (or FastAPI, etc) will be ported to Mojo programming language?

According to the docs, Mojo is supposed to be a superset of Python, so technically nothing will need to be ported.

Enter Mojo

Chris Lattner is responsible for creating many of the projects that we all rely on today – even although we might not even have heard of all the stuff he built! As part of his PhD thesis he started the development of LLVM, which fundamentally changed how compilers are created, and today forms the foundation of many of the most widely used language ecosystems in the world. He then went on to launch Clang, a C and C++ compiler that sits on top of LLVM, and is used by most of the world's most significant software developers (including providing the backbone for Google's performance critical code). LLVM includes an "intermediate representation" (IR), a special language designed for machines to read and write (instead of for people), which has enabled a huge community of software to work together to provide better programming language functionality across a wider range of hardware.

Chris saw that C and C++ however didn't really fully leverage the power of LLVM, so while he was working at Apple he designed a new language, called "Swift", which he describes as "syntax sugar for LLVM". Swift has gone on to become one of the world's most widely used programming languages, in particular because it is today the main way to create iOS apps for iPhone, iPad, MacOS, and Apple TV.

Unfortunately, Apple's control of Swift has meant it hasn't really had its time to shine outside of the cloistered Apple world. Chris led a team for a while at Google to try to move Swift out of its Apple comfort zone, to become a replacement for Python in AI model development. I was very excited about this project, but sadly it did not receive the support it needed from either Apple or from Google, and it was not ultimately successful.

Having said that, whilst at Google Chris did develop another project which became hugely successful: MLIR. MLIR is a replacement for LLVM's IR for the modern age of many-core computing and AI workloads. It's critical for fully leveraging the power of hardware like GPUs, TPUs, and the vector units increasingly being added to server-class CPUs.

So, if Swift was “syntax sugar for LLVM”, what’s “syntax sugar for MLIR”? The answer is: Mojo! Mojo is a brand new language that’s designed to take full advantage of MLIR. And also Mojo is Python.

Wait what?

OK let me explain. Maybe it’s better to say Mojo is Python++. It will be (when complete) a strict superset of the Python language. But it also has additional functionality so we can write high performance code that takes advantage of modern accelerators.

Mojo seems to me like a more pragmatic approach than Swift. Whereas Swift was a brand new language packing all kinds of cool features based on latest research in programming language design, Mojo is, at its heart, just Python. This seems wise, not just because Python is already well understood by millions of coders, but also because after decades of use its capabilities and limitations are now well understood. Relying on the latest programming language research is pretty cool, but its potentially-dangerous speculation because you never really know how things will turn out. (I will admit that personally, for instance, I often got confused by Swift’s powerful but quirky type system, and sometimes even managed to confuse the Swift compiler and blew it up entirely!)

A key trick in Mojo is that you can opt in at any time to a faster “mode” as a developer, by using “fn” instead of “def” to create your function. In this mode, you have to declare exactly what the type of every variable is, and as a result Mojo can create optimised machine code to implement your function. Furthermore, if you use “struct” instead of “class”, your attributes will be tightly packed into memory, such that they can even be used in data structures without chasing pointers around. These are the kinds of features that allow languages like C to be so fast, and now they’re accessible to Python programmers too – just by learning a tiny bit of new syntax.

How is this possible?

There has, at this point, been hundreds of attempts over decades to create programming languages which are concise, flexible, fast, practical, and easy to use – without much success. But somehow, Modular seems to have done it. How could this be? There are a couple of hypotheses we might come up with:

1. Mojo hasn’t actually achieved these things, and the snazzy demo hides disappointing real-life performance, or
2. Modular is a huge company with hundreds of developers working for years, putting in more hours in order to achieve something that’s not been achieved before.

Neither of these things are true. The demo, in fact, was created in just a few days before I recorded the video. The two examples we gave (matmul and mandelbrot) were not carefully chosen as being the only things that happened to work after trying dozens of approaches; rather, they were the only things we tried for the demo and they worked first time! Whilst there’s plenty of missing features at this early stage (Mojo isn’t even released to the public yet, other than an

online “playground”), the demo you see really does work the way you see it. And indeed you *can* run it yourself now in the playground.

Modular is a fairly small startup that’s only a year old, and only one part of the company is working on the Mojo language. Mojo development was only started recently. It’s a small team, working for a short time, so how have they done so much?

The key is that Mojo builds on some really powerful foundations. Very few software projects I’ve seen spend enough time building the right foundations, and tend to accrue as a result mounds of technical debt. Over time, it becomes harder and harder to add features and fix bugs. In a well designed system, however, every feature is easier to add than the last one, is faster, and has fewer bugs, because the foundations each feature builds upon are getting better and better. Mojo is a well designed system.

At its core is MLIR, which has already been developed for many years, initially kicked off by Chris Lattner at Google. He had recognised what the core foundations for an “AI era programming language” would need, and focused on building them. MLIR was a key piece. Just as LLVM made it dramatically easier for powerful new programming languages to be developed over the last decade (such as Rust, Julia, and Swift, which are all based on LLVM), MLIR provides an even more powerful core to languages that are built on it.

Another key enabler of Mojo’s rapid development is the decision to use Python as the syntax. Developing and iterating on syntax is one of the most error-prone, complex, and controversial parts of the development of a language. By simply outsourcing that to an existing language (which also happens to be the most widely used language today) that whole piece disappears! The relatively small number of new bits of syntax needed on top of Python then largely fit quite naturally, since the base is already in place.

The next step was to create a minimal Pythonic way to call MLIR directly. That wasn’t a big job at all, but it was all that was needed to then create all of Mojo on top of that – and work directly *in Mojo* for everything else. That meant that the Mojo devs were able to “dog-food” Mojo when writing Mojo, nearly from the very start. Any time they found something didn’t quite work great as they developed Mojo, they could add a needed feature to Mojo itself to make it easier for them to develop the next bit of Mojo!

This is very similar to Julia, which was developed on a minimal LISP-like core that provides the Julia language elements, which are then bound to basic LLVM operations. Nearly everything in Julia is built on top of that, using Julia itself.

I can’t begin to describe all the little (and big!) ideas throughout Mojo’s design and implementation – it’s the result of Chris and his team’s decades of work on compiler and language design and includes all the tricks and hard-won experience from that time – but what I can describe is an amazing result that I saw with my own eyes.

The Modular team internally announced that they’d decided to launch Mojo with a video, including a demo – and they set a date just a few weeks in the future. But at that time Mojo was

just the most bare-bones language. There was no usable notebook kernel, hardly any of the Python syntax was implemented, and nothing was optimised. I couldn't understand how they hoped to implement all this in a matter of weeks – let alone to make it any good! What I saw over this time was astonishing. Every day or two whole new language features were implemented, and as soon as there was enough in place to try running algorithms, generally they'd be at or near state of the art performance right away! I realised that what was happening was that all the foundations were already in place, and that they'd been explicitly designed to build the things that were now under development. So it shouldn't have been a surprise that everything worked, and worked well – after all, that was the plan all along!

This is a reason to be optimistic about the future of Mojo. Although it's still early days for this project, my guess, based on what I've observed in the last few weeks, is that it's going to develop faster and further than most of us expect…

Deployment

I've left one of the bits I'm most excited about to last: deployment. Currently, if you want to give your cool Python program to a friend, then you're going to have to tell them to first install Python! Or, you could give them an enormous file that includes the entirety of Python and the libraries you use all packaged up together, which will be extracted and loaded when they run your program.

Because Python is an interpreted language, how your program will behave will depend on the exact version of python that's installed, what versions of what libraries are present, and how it's all been configured. In order to avoid this maintenance nightmare, instead the Python community has settled on a couple of options for installing Python applications: *environments*, which have a separate Python installation for each program; or *containers*, which have much of an entire operating system set up for each application. Both approaches lead to a lot of confusion and overhead in developing and deploying Python applications.

Compare this to deploying a statically compiled C application: you can literally just make the compiled program available for direct download. It can be just 100k or so in size, and will launch and run quickly.

There is also the approach taken by Go, which isn't able to generate small applications like C, but instead incorporates a "runtime" into each packaged application. This approach is a compromise between Python and C, still requiring tens of megabytes for a binary, but providing for easier deployment than Python.

As a compiled language, Mojo's deployment story is basically the same as C. For instance, a program that includes a version of matmul written from scratch is around 100k.

This means that Mojo is *far more* than a language for AI/ML applications. It's actually a version of Python that allows us to write fast, small, easily-deployed applications that take advantage of all available cores and accelerators!

Alternatives to Mojo

Mojo is not the only attempt at solving the Python performance and deployment problem. In terms of languages, Julia is perhaps the strongest current alternative. It has many of the benefits of Mojo, and a lot of great projects are already built with it. The Julia folks were kind enough to invite me to give a keynote at their recent conference, and I used that opportunity to describe what I felt were the current shortcomings (and opportunities) for Julia:

As discussed in this video, Julia's biggest challenge stems from its large runtime, which in turn stems from the decision to use garbage collection in the language. Also, the multi-dispatch approach used in Julia is a fairly unusual choice, which both opens a lot of doors to do cool stuff in the language, but also can make things pretty complicated for devs. (I'm so enthused by this approach that I built a python version of it – but I'm also as a result particularly aware of its limitations!)

In Python, the most prominent current solution is probably Jax, which effectively creates a *domain specific language* (DSL) using Python. The output of this language is XLA, which is a machine learning compiler that predates MLIR (and is gradually being ported over to MLIR, I believe). Jax inherits the limitations of both Python (e.g the language has no way of representing structs, or allocating memory directly, or creating fast loops) and XLA (which is largely limited to machine learning specific concepts and is primarily targeted to TPUs), but has the huge upside that it doesn't require a new language or new compiler.

As previously discussed, there's also the new PyTorch compiler, and also Tensorflow is able to generate XLA code. Personally, I find using Python in this way ultimately unsatisfying. I don't actually get to use all the power of Python, but have to use a subset that's compatible with the backend I'm targeting. I can't easily debug and profile the compiled code, and there's so much "magic" going on that it's hard to even know what actually ends up getting executed. I don't even end up with a standalone binary, but instead have to use special runtimes and deal with complex APIs. (I'm not alone here – everyone I know that has used PyTorch or Tensorflow for targeting edge devices or optimised serving infrastructure has described it as being one of the most complex and frustrating tasks they've attempted! And I'm not sure I even know anyone that's actually completed either of these things using Jax.)

Another interesting direction for Python is Numba and Cython. I'm a big fan of these projects and have used both in my teaching and software development. Numba uses a special decorator to cause a python function to be compiled into optimised machine code using LLVM. Cython is similar, but also provides a Python-like language which has some of the features of Mojo, and converts this Python dialect into C, which is then compiled. Neither language solves the deployment challenge, but they can help a lot with the performance problem.

Neither is able to target a range of accelerators with generic cross-platform code, although Numba does provide a very useful way to write CUDA code (and so allows NVIDIA GPUs to be targeted).

I’m really grateful Numba and Cython exist, and have personally gotten a lot out of them. However they’re not at all the same as using a complete language and compiler that generates standalone binaries. They’re bandaid solutions for Python’s performance problems, and are fine for situations where that’s all you need.

But I’d much prefer to use a language that’s as elegant as Python and as fast as expert-written C, allows me to use one language to write everything from the application server, to the model architecture and the installer too, and lets me debug and profile my code directly in the language in which I wrote it.