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 we 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 the present.

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 ecosystem 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. Python 3.13 adds a JIT compiler to CPython, enables the use of multiple subinterpreters in a single Python process (thus sidestepping the GIL) and speeds up memory management. 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 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.

To learn about what we've built with Mojo so far, see the Mojo Manual.