

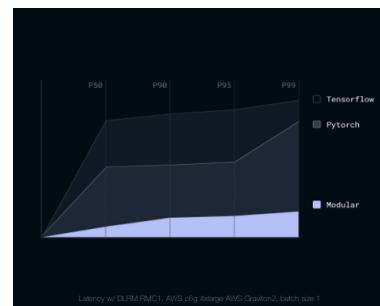
Welcome to the Modular docs

We're excited to share an early look at the Modular compute and AI infrastructure stack with the following documentation. Our mission to revolutionize the AI developer experience still has a ways to go, but [we're making rapid progress](#).

The AI Engine isn't generally available yet, but you can [sign up here for early access](#).

And although the Mojo language is still young, it's now available with the [Mojo SDK!](#)

[Talk to us on Discord](#)



[Modular AI Engine](#)

[The world's fastest unified inference engine, supercharging any model from TensorFlow or PyTorch on a wide range of hardware.](#)



[Mojo](#)

[A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.](#)

No matching items

[Go to Modular.com](#)

Modular AI Engine

The world's fastest unified inference engine, supercharging any model from TensorFlow or PyTorch on a wide range of hardware.

The Modular AI Engine can help simplify your workflow and reduce your inference latency so you can scale your AI products.

We've incorporated best-in-class compiler and runtime technologies to create the world's fastest unified inference engine. 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.

Below, you can preview our APIs for Python and C, and our C++ API is coming soon!

[See our performance dashboard](#)

Python API

[Get started in Python](#)

[A walkthrough of the Python API, showing how to load and run a trained model.](#)

[AI Engine Python API](#)

[The AI Engine Python API reference.](#)

No matching items

C API

[Get started in C](#)

[A walkthrough of the C API, showing how to load and run a trained model.](#)

[AI Engine C API](#)

[The AI Engine C API reference.](#)

No matching items

Server integration

[Server integration overview](#)

[An introduction to using the AI Engine as a backend for Triton and TF Serving.](#)

[Triton serving demo](#)

[A Jupyter notebook that compares the performance of Triton with TF to Triton with Modular.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Why Mojo

A backstory and rationale for why we created the Mojo language.

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.

To see all the features we've built with Mojo so far, see the [Mojo programming manual](#).

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Get started with Mojo

Get the Mojo SDK or try coding in the Mojo Playground.

Mojo is now available for local development!

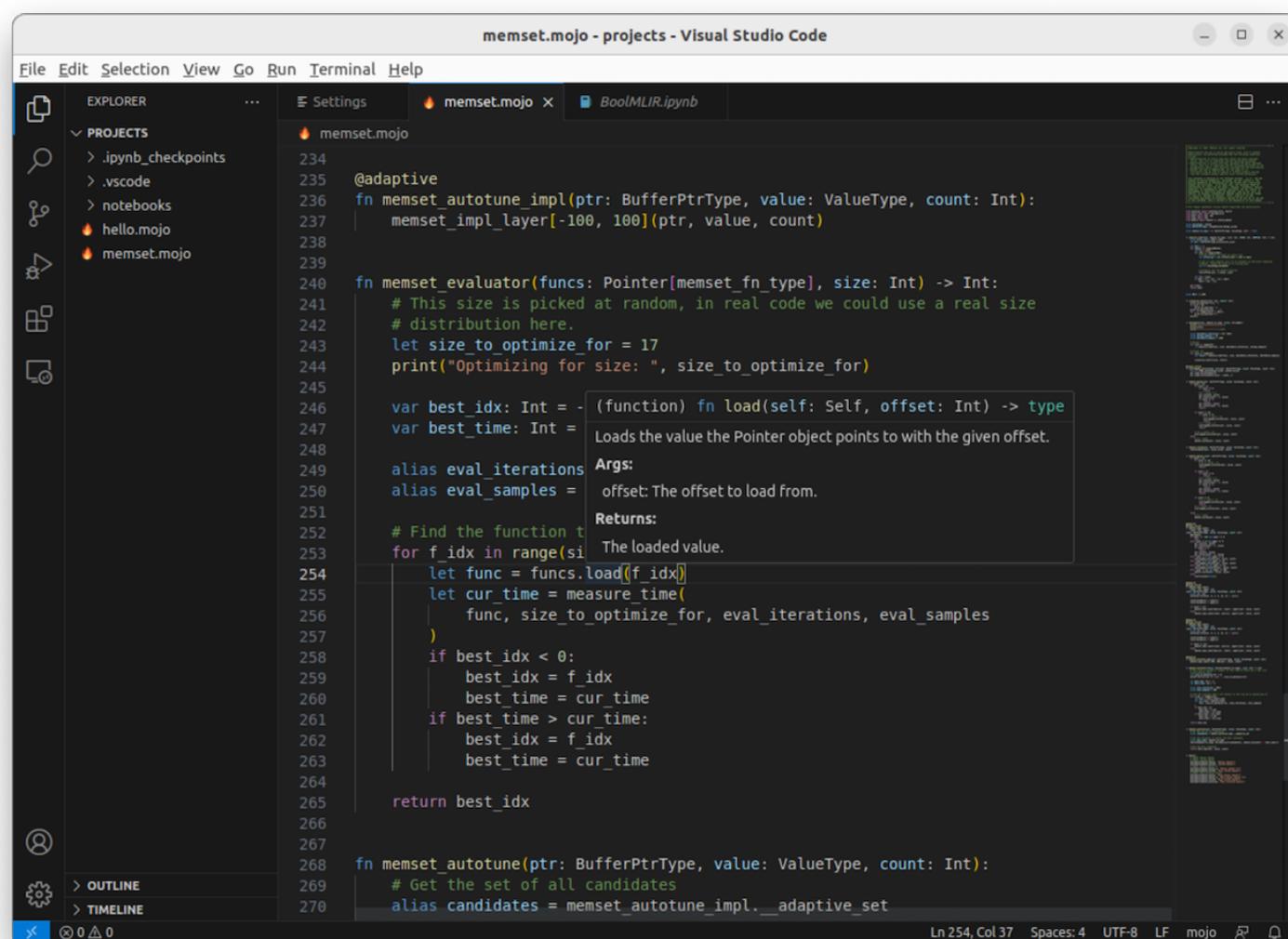
[Download Now](#) 

The Mojo SDK is currently available for Ubuntu Linux systems and macOS systems running on Apple silicon. Support for Windows is coming soon. Our setup guide also includes instructions about how to develop from Windows or Intel macOS using a container or remote Linux system. Alternatively, you can also experiment with Mojo using our web-based [Mojo Playground](#).

Get the Mojo SDK

The Mojo SDK includes everything you need for local Mojo development, including the Mojo standard library and the [Mojo command-line interface](#) (CLI). The Mojo CLI can start a REPL programming environment, compile and run Mojo source files, format source files, and more.

We've also published a [Mojo extension for Visual Studio Code](#) to provide a first-class developer experience with features like code completion, quick fixes, and hover help for Mojo APIs.



```
memset.mojo - projects - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER ... Settings memset.mojo x BoolMLIR.ipynb
memset.mojo
234
235 @adaptive
236 fn memset_autotune_impl(ptr: BufferPtrType, value: ValueType, count: Int):
237     memset_impl_layer[-100, 100](ptr, value, count)
238
239
240 fn memset_evaluator(funcs: Pointer[memset_fn_type], size: Int) -> Int:
241     # This size is picked at random, in real code we could use a real size
242     # distribution here.
243     let size_to_optimize_for = 17
244     print("Optimizing for size: ", size_to_optimize_for)
245
246     var best_idx: Int = -
247     var best_time: Int =
248
249     alias eval_iterations
250     alias eval_samples =
251
252     # Find the function to use
253     for f_idx in range(size):
254         let func = funcs.load(f_idx)
255         let cur_time = measure_time(
256             func, size_to_optimize_for, eval_iterations, eval_samples
257         )
258         if best_idx < 0:
259             best_idx = f_idx
260             best_time = cur_time
261         if best_time > cur_time:
262             best_idx = f_idx
263             best_time = cur_time
264
265     return best_idx
266
267
268 fn memset_autotune(ptr: BufferPtrType, value: ValueType, count: Int):
269     # Get the set of all candidates
270     alias candidates = memset_autotune_impl.__adaptive_set
```

System requirements

To use the Mojo SDK, you need a system that meets these specifications:

Linux:

- Ubuntu 20.04/22.04 LTS
- x86-64 CPU (with [SSE4.2 or newer](#)) and a minimum of 8 GiB memory
- Python 3.8 - 3.11
- g++ or clang++ C++ compiler

Mac:

- Apple silicon (M1 or M2 processor)
- macOS Monterey (12) or later
- Python 3.8 - 3.11
- Command-line tools for Xcode, or Xcode

Support for Windows will be added in a future release.

Install Mojo

The Mojo SDK is available through the [Modular CLI tool](#), which works like a package manager to install and update Mojo. Use the following link to log into the Modular developer console, where you can get the Modular CLI and then install Mojo:

[Download Now](#) 

Then get started with [Hello, world!](#)

Note: To help us improve Mojo, we collect some basic system information and crash reports. [Learn more.](#)

Update Mojo

Mojo is a work in progress and we will release regular updates to the Mojo language and SDK tools. For information about each release, see the [Mojo changelog](#).

To check your current Mojo version, use the `--version` option:

```
mojo --version
```

To update to the latest Mojo version, use the `modular update` command:

```
modular update mojo
```

Update the Modular CLI

We may also release updates to the `modular` tool. Run the following commands to update the CLI on your system.

Linux:

```
sudo apt update  
sudo apt install modular
```

Mac:

```
brew update  
brew upgrade modular
```

Develop in the Mojo Playground

Instead of downloading the Mojo SDK, you can also experiment with Mojo in our hosted Jupyter notebook environment called Mojo Playground. This is a hosted version of [JupyterLab](#) that's running our latest Mojo kernel.

To get access, just [log in to the Mojo Playground here](#).

The screenshot shows the JupyterLab interface for the Mojo Playground. On the left, there is a file browser window titled "HelloMojo.ipynb" showing files like "Mandelbrot.ipynb", "HelloMojo.ipynb", and "BoolMLIR.ipynb". The main area is a code editor titled "Parallelizing Mandelbrot" containing the following Mojo code:

```
[10]: from Functional import parallelize

def compute_mandelbrot_simd_parallel() -> Matrix:
    # create a matrix. Each element of the matrix corresponds to a pixel
    var result = Matrix(xn, yn)

    let dx = (xmax - xmin) / xn
    let dy = (ymax - ymin) / yn

    alias SIMD_width = dtype SIMD_width[DType.f32]()

    @parameter
    fn _process_row(row:Int):
        var y = ymin + dy*row
        var x = xmin
    @parameter
    fn _process_simd_element[simd_width:Int](col: Int):
        let c = ComplexGenericSIMD[DType.f32, SIMD_width](dx*ioata[simd_width, DType.f32]() + x,
                                                          SIMD[DType.f32, SIMD_width](y))
        result.store[simd_width](col, row, mandelbrot_kernel_simd[simd_width](c))
        x += SIMD_width*dx

    vectorize[simd_width, _process_simd_element](xn)

    parallelize[_process_row](yn)
    return result

make_plot(compute_mandelbrot_simd_parallel())
print("finished")
```

At the bottom of the interface, it says "Mode: Command" and "Ln 1, Col 1 Mandelbrot.ipynb".

What to expect

- The Mojo Playground is a [JupyterHub](#) environment in which you get a private volume associated with your account, so you can create your own notebooks and they'll be saved across sessions.
- We've included a handful of notebooks to show you Mojo basics and demonstrate its capabilities.
- The number of vCPU cores available in your cloud instance may vary, so baseline performance is not representative of the language. However, as you will see in the included Matmul.ipynb notebook, Mojo's relative performance over Python is significant.
- There might be some bugs. Please [report issues and feedback on GitHub](#).

Tips

- If you want to keep any edits to the included notebooks, **rename the notebook files**. These files will reset upon any server refresh or update, sorry. So if you rename the files, your changes will be safe.
- You can use `%%python` at the top of a notebook cell and write normal Python code. Variables, functions, and imports defined in a Python cell are available for access in subsequent Mojo cells.

Caveats

- Did we mention that the included notebooks will lose your changes?
Rename the files if you want to save your changes.
- The Mojo environment does not have network access, so you cannot install other tools or Python packages. However, we've included a variety of popular Python packages, such as `numpy`, `pandas`, and `matplotlib` (see how to [import Python modules](#)).
- Redefining implicit variables is not supported (variables without a `let` or `var` in front). If you'd like to redefine a variable across notebook cells, you must introduce the variable with `var` (`let` variables are immutable).
- You can't use global variables inside functions—they're only visible to other global variables.
- For a longer list of things that don't work yet or have pain-points, see the [Mojo roadmap and sharp edges](#).

Hello, world!

Learn to run your first Mojo program.

After you [install Mojo](#), you can use the [Mojo CLI](#) to build and compile Mojo programs. So let's create the classic starter program that prints "Hello, world!"

Before you start:

You must set the MODULAR_HOME and PATH environment variables, as described in the output when you ran `modular install mojo`. For example, if you're using bash or zsh, add the following lines to your configuration file (`.bash_profile`, `.bashrc`, or `.zshrc`):

```
export MODULAR_HOME="$HOME/.modular"
export PATH="$MODULAR_HOME/pkg/packages.modular.com_moj/bin:$PATH"
```

Then source the file you just updated, for example:

```
source ~/.bash_profile
```

If you have other issues during install, check our [known issues](#).

Run code in the REPL

First, let's try running some code in the Mojo [REPL](#), which allows you to write and run Mojo code directly in a command prompt:

1. To start a REPL session, type `mojo` in your terminal and press Enter.
2. Then type `print("Hello, world!")` and press Enter twice (a blank line is required to indicate the end of an expression).

That's it! For example:

```
$ mojo
Welcome to Mojo! □

Expressions are delimited by a blank line.
Type `:quit` to exit the REPL and `:mojo help` for further assistance.

1> print("Hello, world!")
2.
Hello, world!
```

You can write as much code as you want in the REPL. You can press Enter to start a new line and continue writing code, and when you want Mojo to evaluate the code, press Enter twice. If there's something to print, Mojo prints it and then returns the prompt to you.

The REPL is primarily useful for short experiments because the code isn't saved. So when you want to write a real program, you need to write the code in a `.mojo` source file.

Build and run Mojo source files

Now let's print "Hello, world" with a source file. Mojo source files are identified with either the `.mojo` or `.□` file extension.

You can quickly execute a Mojo file by passing it to the `mojo` command, or you can build a compiled executable with the `mojo build` command. Let's try both.

Run a Mojo file

First, write the Mojo code and execute it:

1. Create a file named `hello.mojo` (or `hello.jo`) and add the following code:

```
fn main():
    print("Hello, world!")
```

That's all you need. Save the file and return to your terminal.

2. Now run it with the `mojo` command:

```
mojo hello.mojo
```

It should immediately print the message:

```
Hello, world!
```

If this didn't work for you, double-check your code looks exactly like the code in step 1, and make sure you correctly [installed Mojo](#).

Build an executable binary

Now, build and run an executable:

1. Create a stand-alone executable with the `build` command:

```
mojo build hello.mojo
```

It creates the executable with the same name as the `.mojo` file, but you can change that with the `-o` option.

2. Then run the executable:

```
./hello
```

The executable runs on your system like any C or C++ executable.

Next steps

- If you're developing in VS Code, install the [Mojo extension](#) so you get syntax highlighting, code completion, diagnostics, and more.
- If you're new to Mojo, read the [Mojo language basics](#).
- If you want to package your code as a library, read about [Mojo modules and packages](#).
- If you want to explore some Mojo code, clone our repo to see some examples:

```
git clone https://github.com/modularml/mojo.git
```

Then open the `/examples` directory in your IDE to try our examples:

- The [code examples](#) offer a variety of demos with the standard library to help you learn Mojo's features and start your own projects.
- The [Mojo notebooks](#) are the same Jupyter notebooks we publish in the [Mojo Playground](#), which demonstrate a variety of language features. Now with the Mojo SDK, you can also run them in VS Code or in JupyterLab.
- For a deep dive into the language, check out the [Mojo programming manual](#).
- To see all the available Mojo APIs, check out the [Mojo standard library reference](#).

Note: The Mojo SDK is still in early development, but you can expect constant improvements to both the language and tools. Please see the [known issues](#) and [report any other issues on GitHub](#).

Mojo language basics

A short introduction to the Mojo language basics.

Mojo is a powerful programming language that's primarily designed for high-performance systems programming, so it has a lot in common with other systems languages like Rust and C++. Yet, Mojo is also designed to become a superset of Python, so a lot of language features and concepts you might know from Python translate nicely to Mojo.

For example, if you're in a REPL environment or Jupyter notebook (like this document), you can run top-level code just like Python:

```
print("Hello Mojo!")
```

```
Hello Mojo!
```

You don't normally see that with other systems programming languages.

Mojo preserves Python's dynamic features and language syntax, and it even allows you to import and run code from Python packages. However, it's important to know that Mojo is an entirely new language, not just a new implementation of Python with syntax sugar. Mojo takes the Python language to a whole new level, with systems programming features, strong type-checking, memory safety, next-generation compiler technologies, and more. Yet, it's still designed to be a simple language that's useful for general-purpose programming.

This page provides a gentle introduction to the Mojo language, and requires only a little programming experience. So let's get started!

If you're an experienced systems programmer and want a deep dive into the language, check out the [Mojo programming manual](#).

Language basics

First and foremost, Mojo is a compiled language and a lot of its performance and memory-safety features are derived from that fact. Mojo code can be ahead-of-time (AOT) or just-in-time (JIT) compiled.

Like other compiled languages, Mojo programs (.mojo or .mo files) require a `main()` function as the entry point to the program. For example:

```
fn main():
    var x: Int = 1
    x += 1
    print(x)
```

If you know Python, you might have expected the function name to be `def main()` instead of `fn main()`. Both actually work in Mojo, but using `fn` behaves a bit differently, as we'll discuss below.

Of course, if you're building a Mojo module (an API library), not a Mojo program, then your file doesn't need a `main()` function (because it will be imported by other programs that do have one).

Note: When you're writing code in a .mojo/.mo file, you can't run top-level code as shown on this page—all code in a Mojo program or module must be encased in a function or struct. However, top-level code does work in a REPL or Jupyter notebook (such as the [notebook for this page](#)).

Now let's explain the code in this `main()` function.

Syntax and semantics

This is simple: Mojo supports (or will support) all of Python's syntax and semantics. If you're not familiar with Python syntax, there are a ton of great resources online that can teach you.

For example, like Python, Mojo uses line breaks and indentation to define code blocks (not curly braces), and Mojo supports all of Python's control-flow syntax such as `if` conditions and `for` loops.

However, Mojo is still a work in progress, so there are some things from Python that aren't implemented in Mojo yet (see the [Mojo roadmap](#)). All the missing Python features will arrive in time, but Mojo already includes many features and capabilities beyond what's available in Python.

As such, the following sections will focus on some of the language features that are unique to Mojo (compared to Python).

Functions

Mojo functions can be declared with either `fn` (shown above) or `def` (as in Python). The `fn` declaration enforces strongly-typed and memory-safe behaviors, while `def` provides Python-style dynamic behaviors.

Both `fn` and `def` functions have their value, and it's important that you learn them both. However, for the purposes of this introduction, we're going to focus on `fn` functions only. For much more detail about both, see the [programming manual](#).

In the following sections, you'll learn how `fn` functions enforce strongly-typed and memory-safe behaviors in your code.

Variables

You can declare variables (such as `x` in the above `main()` function) with `var` to create a mutable value, or with `let` to create an immutable value.

If you change `var` to `let` in the `main()` function above and run it, you'll get a compiler error like this:

```
error: Expression [15]:7:5: expression must be mutable for in-place operator destination
      x += 1
      ^
```

That's because `let` makes the value immutable, so you can't increment it.

And if you delete `var` completely, you'll get an error because `fn` functions require explicit variable declarations (unlike Python-style `def` functions).

Finally, notice that the `x` variable has an explicit `Int` type specification. Declaring the type is not required for variables in `fn`, but it is desirable sometimes. If you omit it, Mojo infers the type, as shown here:

```
fn do_math():
    let x: Int = 1
    let y = 2
    print(x + y)

do_math()__
```

3

Function arguments and returns

Although types aren't required for variables declared in the function body, they are required for arguments and return values for an `fn` function.

For example, here's how to declare `Int` as the type for function arguments and the return value:

```
fn add(x: Int, y: Int) -> Int:  
    return x + y  
  
z = add(1, 2)  
print(z)<!--</pre>
```

3

Optional arguments and keyword arguments

You can also specify argument default values (also known as optional arguments), and pass values with keyword argument names. For example:

```
fn pow(base: Int, exp: Int = 2) -> Int:  
    return base ** exp  
  
# Uses default value for `exp`  
z = pow(3)  
print(z)  
  
# Uses keyword argument names (with order reversed)  
z = pow(exp=3, base=2)  
print(z)<!--</pre>
```

9
8

Note: Mojo currently includes only partial support for keyword arguments, so some features such as keyword-only arguments and variadic keyword arguments (e.g. `**kwargs`) are not supported yet.

Argument mutability and ownership

Mojo supports full value semantics and enforces memory safety with a robust value ownership model (similar to the Rust borrow checker). So the following is a quick introduction to you can share references to values through function arguments.

Notice that, above, `add()` doesn't modify `x` or `y`, it only reads the values. In fact, as written, the function *cannot* modify them because `fn` arguments are **immutable references**, by default.

In terms of argument conventions, this is called "borrowing," and although it's the default for `fn` functions, you can make it explicit with the `borrowed` declaration like this (this behaves exactly the same as the `add()` above):

```
fn add(borrowed x: Int, borrowed y: Int) -> Int:  
    return x + y<!--</pre>
```

If you want the arguments to be mutable, you need to declare the argument convention as `inout`. This means that changes made to the arguments *inside* the function are visible *outside* the function.

For example, this function is able to modify the original variables:

```
fn add_inout(inout x: Int, inout y: Int) -> Int:  
    x += 1  
    y += 1  
    return x + y
```

```
var a = 1  
var b = 2  
c = add_inout(a, b)  
print(a)  
print(b)  
print(c)<!--</pre>
```

2
3
5

Another option is to declare the argument as `owned`, which provides the function full ownership of the value (it's mutable and guaranteed unique). This way, the function can modify the value and not worry about affecting variables outside the function. For example:

```
fn set_fire(owned text: String) -> String:  
    text += "█"  
    return text
```

```
fn mojo():  
    let a: String = "mojo"  
    let b = set_fire(a)  
    print(a)  
    print(b)
```

```
mojo()█
```

```
mojo  
mojo█
```

In this case, Mojo makes a copy of `a` and passes it as the `text` argument. The original `a` string is still alive and well.

However, if you want to give the function ownership of the value and **do not** want to make a copy (which can be an expensive operation for some types), then you can add the `^` “transfer” operator when you pass `a` to the function. The transfer operator effectively destroys the local variable name—any attempt to call upon it later causes a compiler error.

Try it above by changing the call to `set_fire()` to look like this:

```
let b = set_fire(a^)█
```

You'll now get an error because the transfer operator effectively destroys the `a` variable, so when the following `print()` function tries to use `a`, that variable isn't initialized anymore.

If you delete `print(a)`, then it works fine.

These argument conventions are designed to provide systems programmers with total control for memory optimizations while ensuring safe access and timely deallocations—the Mojo compiler ensures that no two variables have mutable access to the same value at the same time, and the lifetime of each value is well-defined to strictly prevent any memory errors such as “use-after-free” and “double-free.”

Note: Currently, Mojo always makes a copy when a function returns a value.

Structures

You can build high-level abstractions for types (or “objects”) in a `struct`. A `struct` in Mojo is similar to a `class` in Python: they both support methods, fields, operator overloading, decorators for metaprogramming, etc. However, Mojo structs are completely static—they are bound at compile-time, so they do not allow dynamic dispatch or any runtime changes to the structure. (Mojo will also support classes in the future.)

For example, here's a basic struct:

```
struct MyPair:  
    var first: Int  
    var second: Int  
  
    fn __init__(inout self, first: Int, second: Int):  
        self.first = first  
        self.second = second  
  
    fn dump(self):  
        print(self.first, self.second)█
```

And here's how you can use it:

```
let mine = MyPair(2, 4)
mine.dump()__
```

```
2 4
```

If you're familiar with Python, then the `__init__()` method and the `self` argument should be familiar to you. If you're *not* familiar with Python, then notice that, when we call `dump()`, we don't actually pass a value for the `self` argument. The value for `self` is automatically provided with the current instance of the struct (it's used similar to the `this` name used in some other languages to refer to the current instance of the object/type).

For much more detail about structs and other special methods like `__init__()` (also known as "dunder" methods), see the [programming manual](#).

Python integration

Although Mojo is still a work in progress and is not a full superset of Python yet, we've built a mechanism to import Python modules as-is, so you can leverage existing Python code right away. Under the hood, this mechanism uses the CPython interpreter to run Python code, and thus it works seamlessly with all Python modules today.

For example, here's how you can import and use NumPy (you must have Python `numpy` installed):

```
from python import Python
let np = Python.import_module("numpy")
ar = np.arange(15).reshape(3, 5)
print(ar)
print(ar.shape)__
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
(3, 5)
```

Note: Mojo is not a feature-complete superset of Python yet. So, you can't always copy Python code and run it in Mojo. For more details on our plans, please refer to the [Mojo roadmap and sharp edges](#).

Caution: When you install Mojo, the installer searches your system for a version of Python to use with Mojo, and adds the path to the `modular.cfg` config file. If you change your Python version or switch virtual environments, Mojo will then be looking at the wrong Python library, which can cause problems such as errors when you import Python packages (Mojo says only An error occurred in Python—this is a separate [known issue](#)). The current solution is to override Mojo's path to the Python library, using the `MOJO PYTHON LIBRARY` environment variable. For instructions on how to find and set this path, see [this related issue](#).

Next steps

We hope this page covered enough of the basics to get you started. It's intentionally brief, so if you want more detail about any of the topics touched upon here, check out the [Mojo programming manual](#).

- If you want to package your code as a library, read about [Mojo modules and packages](#).
- If you want to explore some Mojo code, check out our [code examples on GitHub](#).
- To see all the available Mojo APIs, check out the [Mojo standard library reference](#).

Note: The Mojo SDK is still in early development. Some things are still rough, but you can expect constant changes and improvements to both the language and tools. Please see the [known issues](#) and [report any other issues on GitHub](#).

Mojo modules and packages

Learn how to package Mojo code for distribution and importing.

Mojo provides a packaging system that allows you to organize and compile code libraries into importables files. This page introduces the necessary concepts about how to organize your code into modules and packages (which is a lot like Python), and shows you how to create a packaged binary with the [mojo_package](#) command.

Mojo modules

To understand Mojo packages, you first need to understand Mojo modules. A Mojo module is a single Mojo source file that includes code suitable for use by other files that import it. For example, you can create a module to define a struct such as this one:

```
mymodule.mojo

struct MyPair:
    var first: Int
    var second: Int

fn __init__(inout self, first: Int, second: Int):
    self.first = first
    self.second = second

fn dump(self):
    print(self.first, self.second)○
```

Notice that this code has no `main()` function, so you can't execute `mymodule.mojo`. However, you can import this into another file with a `main()` function and use it there.

For example, here's how you can import `MyPair` into a file named `main.mojo` that's in the same directory as `mymodule.mojo`:

```
main.mojo

from mymodule import MyPair

fn main():
    let mine = MyPair(2, 4)
    mine.dump()○
```

Alternatively, you can import the whole module and then access its members through the module name. For example:

```
main.mojo

import mymodule

fn main():
    let mine = mymodule.MyPair(2, 4)
    mine.dump()○
```

You can also create an alias for an imported member with `as`, like this:

```
main.mojo

import mymodule as my

fn main():
    let mine = my.MyPair(2, 4)
    mine.dump()○
```

In this example, it only works when `mymodule.mojo` is in the same directory as `main.mojo`. Currently, you can't import `.mojo` files as modules if they reside in other directories. That is, unless you treat

the directory as a Mojo package, as described in the next section.

Note: A Mojo module may include a `main()` function and may also be executable, but that's generally not the practice and modules typically include APIs to be imported and used in other Mojo programs.

Mojo packages

A Mojo package is just a collection of Mojo modules in a directory that includes an `__init__.mojo` file. By organizing modules together in a directory, you can then import all the modules together or individually. Optionally, you can also compile the package into a `.mojopkg` or `.mojo` file that's easier to share.

You can import a package and its modules either directly from source files or from a compiled `.mojopkg`/`.mojo` file. It makes no real difference to Mojo which way you import a package. When importing from source files, the directory name works as the package name, whereas when importing from a compiled package, the filename is the package name (which you specify with the [mojo package](#) command—it can differ from the directory name).

For example, consider a project with these files:

```
main.mojo
mypackage/
  __init__.mojo
  mymodule.mojo
```

`mymodule.mojo` is the same code from examples above (with the `MyPair` struct) and `__init__.mojo` is empty.

In this case, the `main.mojo` file can now import `MyPair` through the package name like this:

```
main.mojo
from mypackage.mymodule import MyPair

fn main():
    let mine = my.MyPair(2, 4)
    mine.dump()
```

Notice that the `__init__.mojo` is crucial here. If you delete it, then Mojo doesn't recognize the directory as a package and it cannot import `mymodule`.

Then, let's say you don't want the `mypackage` source code in the same location as `main.mojo`. So, you can compile it into a package file like this:

```
mojo package mypackage -o mypack.mojopkg
```

Then the `mypackage` source can be moved somewhere else, and the project files now look like this:

```
main.mojo
mypack.mojopkg
```

Because we named the package file different from the directory, we need to fix the import statement and it all works the same:

```
main.mojo
from mypack.mymodule import MyPair
```

Note: If you want to rename your package, you cannot simply edit the `.mojopkg` or `.mojo` filename, because the package name is encoded in the file. You must instead run `mojo package` again to specify a new name.

The `__init__` file

As mentioned above, the `__init__.mojo` file is required to indicate that a directory should be treated as a Mojo package, and it can be empty.

Currently, top-level code is not supported in `.mojo` files, so unlike Python, you can't write code in `__init__.mojo` that executes upon import. You can, however, add structs and functions, which you can then import from the package name.

However, instead of adding APIs in the `__init__.mojo` file, you can import module members, which has the same effect by making your APIs accessible from the package name, instead of requiring the `<package_name>.<module_name>` notation.

For example, again let's say you have these files:

```
main.mojo
mypackage/
  __init__.mojo
  mymodule.mojo
```

Let's now add the following line in `__init__.mojo`:

```
__init__.mojo
from .mymodule import MyPair
```

That's all that's in there. Now, we can simplify the import statement in `main.mojo` like this:

```
main.mojo
from mypackage import MyPair
```

This feature explains why some members in the Mojo standard library can be imported from their package name, while others required the `<package_name>.<module_name>` notation. For example, the [functional](#) module resides in the `algorithm` package, so you can import members of that module (such as the `map()` function) like this:

```
from algorithm.functional import map
```

However, the `algorithm/__init__.mojo` file also includes these lines:

```
algorithm/__init__.mojo
from .functional import *
from .reduction import *
```

So you can actually import anything from `functional` or `reduction` simply by naming the package. That is, you can drop the functional name from the import statement, and it also works:

```
from algorithm import map
```

Note: Which modules in the standard library are imported to the package scope varies, and is subject to change. Refer to the [documentation for each module](#) to see how you can import its members.

Mojo programming manual

A tour of major Mojo language features with code examples.

Mojo is a programming language that is as easy to use as Python but with the performance of C++ and Rust. Furthermore, Mojo provides the ability to leverage the entire Python library ecosystem.

Mojo achieves this feat by utilizing next-generation compiler technologies with integrated caching, multithreading, and cloud distribution technologies. Furthermore, Mojo's autotuning and compile-time metaprogramming features allow you to write code that is portable to even the most exotic hardware.

More importantly, **Mojo allows you to leverage the entire Python ecosystem** so you can continue to use tools you are familiar with. Mojo is designed to become a **superset** of Python over time by preserving Python's dynamic features while adding new primitives for [systems programming](#). These new system programming primitives will allow Mojo developers to build high-performance libraries that currently require C, C++, Rust, CUDA, and other accelerator systems. By bringing together the best of dynamic languages and systems languages, we hope to provide a **unified** programming model that works across levels of abstraction, is friendly for novice programmers, and scales across many use cases from accelerators through to application programming and scripting.

This document is an introduction to the Mojo programming language, not a complete language guide. It assumes knowledge of Python and systems programming concepts. At the moment, Mojo is still a work in progress and the documentation is targeted to developers with systems programming experience. As the language grows and becomes more broadly available, we intend for it to be friendly and accessible to everyone, including beginner programmers. It's just not there today.

Using the Mojo compiler

With the [Mojo SDK](#), you can run a Mojo program from a terminal just like you can with Python. So if you have a file named `hello.mojo` (or `hello.□`—yes, the file extension can be an emoji!), just type `mojo hello.mojo`:

```
$ cat hello.□
def main():
    print("hello world")
    for x in range(9, 0, -3):
        print(x)
$ mojo hello.□
hello world
9
6
3
$ □
```

Again, you can use either the `.□` or `.mojo` suffix.

For more details about the Mojo compiler tools, see the [mojo CLI docs](#).

Basic systems programming extensions

Given our goal of compatibility and Python's strength with high-level applications and dynamic APIs, we don't have to spend much time explaining how those portions of the language work. On the other hand, Python's support for systems programming is mainly delegated to C, and we want to provide a single system that is great in that world. As such, this section breaks down each major component and feature and describes how to use them with examples.

let and var declarations

Inside a `def` in Mojo, you may assign a value to a name and it implicitly creates a function scope variable just like in Python. This provides a very dynamic and low-ceremony way to write code, but it is a challenge for two reasons:

1. Systems programmers often want to declare that a value is immutable for type-safety and performance.
2. They may want to get an error if they mistype a variable name in an assignment.

To support this, Mojo provides scoped runtime value declarations: `let` is immutable, and `var` is mutable. These values use lexical scoping and support name shadowing:

```
def your_function(a, b):
    let c = a
    # Uncomment to see an error:
    # c = b # error: c is immutable

    if c != b:
        let d = b
        print(d)
```

```
your_function(2, 3) ↴
```

```
3
```

`let` and `var` declarations support type specifiers as well as patterns, and late initialization:

```
def your_function():
    let x: Int = 42
    let y: Float64 = 17.0

    let z: Float32
    if x != 0:
        z = 1.0
    else:
        z = foo()
    print(z)

def foo() -> Float32:
    return 3.14
```

```
your_function() ↴
```

```
1.0
```

Note that `let` and `var` are completely optional when in a `def` function (you can instead use implicitly declared values, just like Python), but they're required for all variables in an `fn` function.

Also beware that when using Mojo in a REPL environment (such as this notebook), top-level variables (variables that live outside a function or struct) are treated like variables in a `def`, so they allow implicit value type declarations (they do not require `var` or `let` declarations, nor type declarations). This matches the Python REPL behavior.

struct types

Mojo is based on MLIR and LLVM, which offer a cutting-edge compiler and code generation system used in many programming languages. This lets us have better control over data organization, direct access to data fields, and other ways to improve performance. An important feature of modern systems programming languages is the ability to build high-level and safe abstractions on top of these complex, low-level operations without any performance loss. In Mojo, this is provided by the `struct` type.

A `struct` in Mojo is similar to a Python `class`: they both support methods, fields, operator overloading, decorators for metaprogramming, etc. Their differences are as follows:

- Python classes are dynamic: they allow for dynamic dispatch, monkey-patching (or “swizzling”), and dynamically binding instance properties at runtime.
- Mojo structs are static: they are bound at compile-time (you cannot add methods at runtime). Structs allow you to trade flexibility for performance while being safe and easy to use.

Here's a simple definition of a struct:

```
struct MyPair:
    var first: Int
    var second: Int

    # We use 'fn' instead of 'def' here - we'll explain that soon
    fn __init__(inout self, first: Int, second: Int):
        self.first = first
        self.second = second

    fn __lt__(self, rhs: MyPair) -> Bool:
        return self.first < rhs.first or
            (self.first == rhs.first and
             self.second < rhs.second)
```

Syntactically, the biggest difference compared to a Python class is that all instance properties in a struct **must** be explicitly declared with a var or let declaration.

In Mojo, the structure and contents of a “struct” are set in advance and can't be changed while the program is running. Unlike in Python, where you can add, remove, or change attributes of an object on the fly, Mojo doesn't allow that for structs. This means you can't use del to remove a method or change its value in the middle of running the program.

However, the static nature of struct has some great benefits! It helps Mojo run your code faster. The program knows exactly where to find the struct's information and how to use it without any extra steps or delays.

Mojo's structs also work really well with features you might already know from Python, like operator overloading (which lets you change how math symbols like + and - work with your own data). Furthermore, *all* the “standard types” (like Int, Bool, String and even Tuple) are made using structs. This means they're part of the standard set of tools you can use, rather than being hardwired into the language itself. This gives you more flexibility and control when writing your code.

If you're wondering what the inout means on the self argument: this indicates that the argument is mutable and changes made inside the function are visible to the caller. For details, see below about [inout arguments](#).

Int vs int

In Mojo, you might notice that we use Int (with a capital “I”), which is different from Python's int (with a lowercase “i”). This difference is on purpose, and it's actually a good thing!

In Python, the int type can handle really big numbers and has some extra features, like checking if two numbers are the same object. But this comes with some extra baggage that can slow things down. Mojo's Int is different. It's designed to be simple, fast, and tuned for your computer's hardware to handle quickly.

We made this choice for two main reasons:

1. We want to give programmers who need to work closely with computer hardware (systems programmers) a transparent and reliable way to interact with hardware. We don't want to rely on fancy tricks (like JIT compilers) to make things faster.

2. We want Mojo to work well with Python without causing any issues. By using a different name (Int instead of int), we can keep both types in Mojo without changing how Python's int works.

As a bonus, Int follows the same naming style as other custom data types you might create in Mojo. Additionally, Int is a struct that's included in Mojo's standard set of tools.

Strong type checking

Even though you can still use flexible types like in Python, Mojo lets you use strict type checking. Type-checking can make your code more predictable, manageable, and secure.

One of the primary ways to employ strong type checking is with Mojo's struct type. A struct definition in Mojo defines a compile-time-bound name, and references to that name in a type context are treated as a strong specification for the value being defined. For example, consider the following code that uses the MyPair struct shown above:

```
def pair_test() -> Bool:  
    let p = MyPair(1, 2)  
    # Uncomment to see an error:  
    # return p < 4 # gives a compile time error  
    return True
```

If you uncomment the first return statement and run it, you'll get a compile-time error telling you that 4 cannot be converted to MyPair, which is what the right-hand-side of `_lt_()` requires (in the MyPair definition).

This is a familiar experience when working with systems programming languages, but it's not how Python works. Python has syntactically identical features for [MyPy](#) type annotations, but they are not enforced by the compiler: instead, they are hints that inform static analysis. By tying types to specific declarations, Mojo can handle both the classical type annotation hints and strong type specifications without breaking compatibility.

Type checking isn't the only use-case for strong types. Since we know the types are accurate, we can optimize the code based on those types, pass values in registers, and be as efficient as C for argument passing and other low-level details. This is the foundation of the safety and predictability guarantees Mojo provides to systems programmers.

Overloaded functions and methods

Like Python, you can define functions in Mojo without specifying argument data types and Mojo will handle them dynamically. This is nice when you want expressive APIs that just work by accepting arbitrary inputs and let dynamic dispatch decide how to handle the data. However, when you want to ensure type safety, as discussed above, Mojo also offers full support for overloaded functions and methods.

This allows you to define multiple functions with the same name but with different arguments. This is a common feature seen in many languages, such as C++, Java, and Swift.

When resolving a function call, Mojo tries each candidate and uses the one that works (if only one works), or it picks the closest match (if it can determine a close match), or it reports that the call is ambiguous if it can't figure out which one to pick. In the latter case, you can resolve the ambiguity by adding an explicit cast on the call site.

Let's look at an example:

```
struct Complex:  
    var re: Float32  
    var im: Float32  
  
fn __init__(inout self, x: Float32):  
    """Construct a complex number given a real number."""  
    self.re = x
```

```

self.im = 0.0

fn __init__(inout self, r: Float32, i: Float32):
    """Construct a complex number given its real and imaginary components."""
    self.re = r
    self.im = i

```

You can overload methods in structs and classes and overload module-level functions.

Mojo doesn't support overloading solely on result type, and doesn't use result type or contextual type information for type inference, keeping things simple, fast, and predictable. Mojo will never produce an "expression too complex" error, because its type-checker is simple and fast by definition.

Again, if you leave your argument names without type definitions, then the function behaves just like Python with dynamic types. As soon as you define a single argument type, Mojo will look for overload candidates and resolve function calls as described above.

Although we haven't discussed parameters yet (they're different from function arguments), you can also [overload functions and methods based on parameters](#).

fn definitions

The extensions above are the cornerstone that provides low-level programming and provide abstraction capabilities, but many systems programmers prefer more control and predictability than what `def` in Mojo provides. To recap, `def` is defined by necessity to be very dynamic, flexible and generally compatible with Python: arguments are mutable, local variables are implicitly declared on first use, and scoping isn't enforced. This is great for high level programming and scripting, but is not always great for systems programming. To complement this, Mojo provides an `fn` declaration which is like a "strict mode" for `def`.

Alternative: instead of using a new keyword like `fn`, we could instead add a modifier or decorator like `@strict def`. However, we need to take new keywords anyway and there is little cost to doing so. Also, in practice in systems programming domains, `fn` is used all the time so it probably makes sense to make it first class.

As far as a caller is concerned, `fn` and `def` are interchangeable: there is nothing a `def` can provide that a `fn` cannot (and vice versa). The difference is that a `fn` is more limited and controlled on the *inside* of its body (alternatively: pedantic and strict). Specifically, `fns` have a number of limitations compared to `def` functions:

1. Argument values default to being immutable in the body of the function (like a `let`), instead of mutable (like a `var`). This catches accidental mutations, and permits the use of non-copyable types as arguments.
2. Argument values require a type specification (except for `self` in a method), catching accidental omission of type specifications. Similarly, a missing return type specifier is interpreted as returning `None` instead of an unknown return type. Note that both can be explicitly declared to return `object`, which allows one to opt-in to the behavior of a `def` if desired.
3. Implicit declaration of local variables is disabled, so all locals must be declared. This catches name typos and dovetails with the scoping provided by `let` and `var`.
4. Both support raising exceptions, but this must be explicitly declared on a `fn` with the `raises` keyword.

Programming patterns will vary widely across teams, and this level of strictness will not be for everyone. We expect that folks who are used to C++ and already use MyPy-style type annotations in Python to prefer the use of `fns`, but higher level programmers and ML researchers to continue to use `def`. Mojo allows you to freely intermix `def` and `fn` declarations, e.g.

implementing some methods with one and others with the other, and allows each team or programmer to decide what is best for their use-case.

For more about argument behavior in Mojo functions, see the section below about [Argument passing control and memory ownership](#).

The `__copyinit__`, `__moveinit__`, and `__takeinit__` special methods

Mojo supports full “value semantics” as seen in languages like C++ and Swift, and it makes defining simple aggregates of fields very easy with the [@value decorator](#).

For advanced use cases, Mojo allows you to define custom constructors (using Python’s existing `__init__` special method), custom destructors (using the existing `__del__` special method) and custom copy and move constructors using the `__copyinit__`, `__moveinit__` and `__takeinit__` special methods.

These low-level customization hooks can be useful when doing low level systems programming, e.g. with manual memory management. For example, consider a dynamic string type that needs to allocate memory for the string data when constructed and destroy it when the value is destroyed:

```
from memory.unsafe import Pointer

struct HeapArray:
    var data: Pointer[Int]
    var size: Int

    fn __init__(inout self, size: Int, val: Int):
        self.size = size
        self.data = Pointer[Int].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, val)

    fn __del__(owned self):
        self.data.free()

    fn dump(self):
        print_no_newline("[")
        for i in range(self.size):
            if i > 0:
                print_no_newline(", ")
            print_no_newline(self.data.load(i))
        print("]")
```

This array type is implemented using low level functions to show a simple example of how this works. However, if you try to copy an instance of `HeapArray` with the `=` operator, you might be surprised:

```
var a = HeapArray(3, 1)
a.dump() # Should print [1, 1, 1]
# Uncomment to see an error:
# var b = a # ERROR: Vector doesn't implement __copyinit__

var b = HeapArray(4, 2)
b.dump() # Should print [2, 2, 2, 2]
a.dump() # Should print [1, 1, 1]

[1, 1, 1]
[2, 2, 2, 2]
[1, 1, 1]
```

If you uncomment the line to copy `a` into `b`, you’ll see that Mojo doesn’t allow you to make a copy of our array: `HeapArray` contains an instance of `Pointer` (which is equivalent to a low-level C pointer), and Mojo doesn’t know what kind of data it points to or how to copy it. More generally, some types (like atomic numbers) cannot be copied or moved around because their address provides an **identity** just like a class instance does.

In this case, we do want our array to be copyable. To enable this, we have to implement the `__copyinit__` special method, which is conventionally implemented like this:

```
struct HeapArray:  
    var data: Pointer[Int]  
    var size: Int  
  
    fn __init__(inout self, size: Int, val: Int):  
        self.size = size  
        self.data = Pointer[Int].alloc(self.size)  
        for i in range(self.size):  
            self.data.store(i, val)  
  
    fn __copyinit__(inout self, other: Self):  
        self.size = other.size  
        self.data = Pointer[Int].alloc(self.size)  
        for i in range(self.size):  
            self.data.store(i, other.data.load(i))  
  
    fn __del__(owned self):  
        self.data.free()  
  
    fn dump(self):  
        print_no_newline("[")  
        for i in range(self.size):  
            if i > 0:  
                print_no_newline(", ")  
            print_no_newline(self.data.load(i))  
        print("]")
```

With this implementation, our code above works correctly and the `b = a` copy produces a logically distinct instance of the array with its own lifetime and data:

```
var a = HeapArray(3, 1)  
a.dump()    # Should print [1, 1, 1]  
# This is no longer an error:  
var b = a  
  
b.dump()    # Should print [1, 1, 1]  
a.dump()    # Should print [1, 1, 1]
```

```
[1, 1, 1]  
[1, 1, 1]  
[1, 1, 1]
```

Mojo also supports the `__moveinit__` method which allows both Rust-style moves (which transfers a value from one place to another when the source lifetime ends) and the `__takeinit__` method for C++-style moves (where the contents of a value is logically transferred out of the source, but its destructor is still run), and allows defining custom move logic. For more detail, see the [Value Lifecycle](#) section below.

Mojo provides full control over the lifetime of a value, including the ability to make types copyable, move-only, and not-movable. This is more control than languages like Swift and Rust offer, which require values to at least be movable. If you are curious how existing can be passed into the `__copyinit__` method without itself creating a copy, check out the section on [Borrowed arguments](#) below.

Argument passing control and memory ownership

In both Python and Mojo, much of the language revolves around function calls: a lot of the (apparently) built-in behaviors are implemented in the standard library with “dunder” (double-underscore) methods. Inside these magic functions is where a lot of memory ownership is determined through argument passing.

Let's review some details about how Python and Mojo pass arguments:

- All values passed into a *Python* `def` function use reference semantics. This means the function can modify mutable objects passed into it and those changes are visible outside the function. However, the behavior is sometimes surprising for the uninitiated, because you can change the object that an argument points to and that change is not visible outside the function.
- All values passed into a *Mojo* `def` function use value semantics by default. Compared to Python, this is an important difference: A *Mojo* `def` function receives a copy of all arguments—it can modify arguments inside the function, but the changes are **not** visible outside the function.
- All values passed into a *Mojo* [fn function](#) are immutable references by default. This means the function can read the original object (it is *not* a copy), but it cannot modify the object at all.

This convention for immutable argument passing in a *Mojo* `fn` is called “borrowing.” In the following sections, we’ll explain how you can change the argument passing behavior in *Mojo*, for both `def` and `fn` functions.

Why argument conventions are important

In *Python*, all fundamental values are references to objects—as described above, a *Python* function can modify the original object. Thus, *Python* developers are used to thinking about everything as reference semantic. However, at the *CPython* or machine level, you can see that the references themselves are actually passed *by-copy*—*Python* copies a pointer and adjusts reference counts.

This *Python* approach provides a comfortable programming model for most people, but it requires all values to be heap-allocated (and results are occasionally surprising results due to reference sharing). *Mojo* classes (TODO: will) follow the same reference-semantic approach for most objects, but this isn’t practical for simple types like integers in a systems programming context. In these scenarios, we want the values to live on the stack or even in hardware registers. As such, *Mojo* structs are always inlined into their container, whether that be as the field of another type or into the stack frame of the containing function.

This raises some interesting questions: How do you implement methods that need to mutate `self` of a structure type, such as `__iadd__`? How does `let` work, and how does it prevent mutation? How are the lifetimes of these values controlled to keep *Mojo* a memory-safe language?

The answer is that the *Mojo* compiler uses dataflow analysis and type annotations to provide full control over value copies, aliasing of references, and mutation control. These features are similar in many ways to features in the *Rust* language, but they work somewhat differently in order to make *Mojo* easier to learn, and they integrate better into the *Python* ecosystem without requiring a massive annotation burden.

In the following sections, you’ll learn about how you can control memory ownership for objects passed into *Mojo* `fn` functions.

Immutable arguments (borrowed)

A borrowed object is an **immutable reference** to an object that a function receives, instead of receiving a copy of the object. So the callee function has full read-and-execute access to the object, but it cannot modify it (the caller still has exclusive “ownership” of the object).

For example, consider this struct that we don’t want to copy when passing around instances of it:

```
# Don't worry about this code yet. It's just needed for the function below.
# It's a type so expensive to copy around so it does not have a
# __copyinit__ method.
struct SomethingBig:
    var id_number: Int
```

```

var huge: HeapArray
fn __init__(inout self, id: Int):
    self.huge = HeapArray(1000, 0)
    self.id_number = id

# self is passed by-reference for mutation as described above.
fn set_id(inout self, number: Int):
    self.id_number = number

# Arguments like self are passed as borrowed by default.
fn print_id(self): # Same as: fn print_id(borrowed self):
    print(self.id_number)_

```

When passing an instance of `SomethingBig` to a function, it's necessary to pass a reference because `SomethingBig` cannot be copied (it has no `_copyinit_` method). And, as mentioned above, `fn` arguments are immutable references by default, but you can explicitly define it with the `borrowed` keyword as shown in the `use_something_big()` function here:

```

fn use_something_big(borrowed a: SomethingBig, b: SomethingBig):
    """'a' and 'b' are both immutable, because 'borrowed' is the default."""
    a.print_id()
    b.print_id()

let a = SomethingBig(10)
let b = SomethingBig(20)
use_something_big(a, b)_

```

10

20

This default applies to all arguments uniformly, including the `self` argument of methods. This is much more efficient when passing large values or when passing expensive values like a reference-counted pointer (which is the default for Python/Mojo classes), because the copy constructor and destructor don't have to be invoked when passing the argument.

Because the default argument convention for `fn` functions is borrowed, Mojo has simple and logical code that does the right thing by default. For example, we don't want to copy or move all of `SomethingBig` just to invoke the `print_id()` method, or when calling `use_something_big()`.

This borrowed argument convention is similar in some ways to passing an argument by `const&` in C++, which avoids a copy of the value and disables mutability in the callee. However, the borrowed convention differs from `const&` in C++ in two important ways:

1. The Mojo compiler implements a borrow checker (similar to Rust) that prevents code from dynamically forming mutable references to a value when there are immutable references outstanding, and it prevents multiple mutable references to the same value. You are allowed to have multiple borrows (as the call to `use_something_big` does above) but you cannot pass something by mutable reference and borrow at the same time. (TODO: Not currently enabled).
2. Small values like `Int`, `Float`, and `SIMD` are passed directly in machine registers instead of through an extra indirection (this is because they are declared with the [@register_passable_decorator](#)). This is a [significant performance enhancement](#) when compared to languages like C++ and Rust, and moves this optimization from every call site to being declarative on a type.

Similar to Rust, Mojo's borrow checker enforces the exclusivity of invariants. The major difference between Rust and Mojo is that Mojo does not require a sigil on the caller side to pass by borrow. Also, Mojo is more efficient when passing small values, and Rust defaults to moving values instead of passing them around by borrow. These policy and syntax decisions allow Mojo to provide an easier-to-use programming model.

Mutable arguments (`inout`)

On the other hand, if you define an `fn` function and want an argument to be mutable, you must declare the argument as mutable with the `inout` keyword.

Tip: When you see `inout`, it means any changes made to the argument **inside** the function are visible **outside** the function.

Consider the following example, in which the `_iadd_` function (which implements the in-place add operation such as `x += 2`) tries to modify `self`:

```
struct MyInt:  
    var value: Int  
  
    fn __init__(inout self, v: Int):  
        self.value = v  
  
    fn __copyinit__(inout self, other: MyInt):  
        self.value = other.value  
  
    # self and rhs are both immutable in __add__.  
    fn __add__(self, rhs: MyInt) -> MyInt:  
        return MyInt(self.value + rhs.value)  
  
    # ... but this cannot work for __iadd__  
    # Uncomment to see the error:  
    #fn __iadd__(self, rhs: Int):  
    #    self = self + rhs # ERROR: cannot assign to self! □
```

If you uncomment the `__iadd__()` method, you'll get a compiler error.

The problem here is that `self` is immutable because this is a Mojo `fn` function, so it can't change the internal state of the argument (the default argument convention is borrowed). The solution is to declare that the argument is mutable by adding the `inout` keyword on the `self` argument name:

```
struct MyInt:  
    var value: Int  
  
    fn __init__(inout self, v: Int):  
        self.value = v  
  
    fn __copyinit__(inout self, other: MyInt):  
        self.value = other.value  
  
    # self and rhs are both immutable in __add__.  
    fn __add__(self, rhs: MyInt) -> MyInt:  
        return MyInt(self.value + rhs.value)  
  
    # ... now this works:  
    fn __iadd__(inout self, rhs: Int):  
        self = self + rhs □
```

Now the `self` argument is mutable in the function and any changes are visible in the caller, so we can perform in-place addition with `MyInt`:

```
var x: MyInt = 42  
x += 1  
print(x.value) # prints 43 as expected  
  
# However...  
let y = x  
# Uncomment to see the error:  
# y += 1 # ERROR: Cannot mutate 'let' value □
```

If you uncomment the last line above, mutation of the `let` value fails because it isn't possible to form a mutable reference to an immutable value (`let` makes the variable immutable).

Of course, you can declare multiple `inout` arguments. For example, you can define and use a swap function like this:

```
fn swap(inout lhs: Int, inout rhs: Int):
    let tmp = lhs
    lhs = rhs
    rhs = tmp

var x = 42
var y = 12
print(x, y) # Prints 42, 12
swap(x, y)
print(x, y) # Prints 12, 42
```

```
42 12
12 42
```

A very important aspect of this system is that it all composes correctly.

Notice that we don't call this argument passing "by reference." Although the `inout` convention is conceptually the same, we don't call it by-reference passing because the implementation may actually pass values using pointers.

Transfer arguments (`owned` and `^`)

The final argument convention that Mojo supports is the `owned` argument convention. This convention is used for functions that want to take exclusive ownership over a value, and it is often used with the postfix `^` operator.

For example, imagine you're working with a move-only type like a unique pointer:

```
# This is not really a unique pointer, we just model its behavior here
# to serve the examples below.
struct UniquePointer:
    var ptr: Int

    fn __init__(inout self, ptr: Int):
        self.ptr = ptr

    fn __moveinit__(inout self, owned existing: Self):
        self.ptr = existing.ptr

    fn __del__(owned self):
        self.ptr = 0
```

While the `borrow` convention makes it easy to work with this unique pointer without ceremony, at some point you might want to transfer ownership to some other function. This is a situation where you want to use the `^` "transfer" operator with your movable type.

The `^` operator ends the lifetime of a value binding and transfers the value ownership to something else (in the following example, ownership is transferred to the `take_ptr()` function). To support this, you can define functions as taking `owned` arguments. For example, you can define `take_ptr()` to take ownership of an argument as follows:

```
fn take_ptr(owned p: UniquePointer):
    print("take_ptr")
    print(p.ptr)

fn use_ptr(borrowed p: UniquePointer):
    print("use_ptr")
    print(p.ptr)

fn work_with_unique_ptrs():
    let p = UniquePointer(100)
    use_ptr(p)    # Pass to borrowing function.
    take_ptr(p^)  # Pass ownership of the `p` value to another function.

    # Uncomment to see an error:
    # use_ptr(p) # ERROR: p is no longer valid here!
```

```
work_with_unique_ptrs()
```

```
use_ptr  
100  
take_ptr  
100
```

Notice that if you uncomment the second call to `use_ptr()`, you get an error because the `p` value has been transferred to the `take_ptr()` function and, thus, the `p` value is destroyed.

Because it is declared `owned`, the `take_ptr()` function knows it has unique access to the value. This is very important for things like unique pointers, and it's useful when you want to avoid copies.

For example, you will notably see the `owned` convention on destructors and on consuming move initializers. For example, our `HeapArray` struct defined earlier uses `owned` in its `__del__()` method, because you need to own a value to destroy it (or to steal its parts, in the case of a move constructor).

Comparing def and fn argument passing

Mojo's `def` function is essentially just sugar for the `fn` function:

- A `def` argument without an explicit type annotation defaults to `Object`.
- A `def` argument without a convention keyword (such as `inout` or `owned`) is passed by implicit copy into a mutable var with the same name as the argument. (This requires that the type have a `__copyinit__` method.)

For example, these two functions have the same behavior:

```
def example(inout a: Int, b: Int, c):  
    # b and c use value semantics so they're mutable in the function  
    ...  
  
fn example(inout a: Int, b_in: Int, c_in: Object):  
    # b_in and c_in are immutable references, so we make mutable shadow copies  
    var b = b_in  
    var c = c_in  
    ...
```

The shadow copies typically add no overhead, because references for small types like `Object` are cheap to copy. The expensive part is adjusting the reference count, but that's eliminated by a move optimization.

Python integration

It's easy to use Python modules you know and love in Mojo. You can import any Python module into your Mojo program and create Python types from Mojo types.

Importing Python modules

To import a Python module in Mojo, just call `Python.import_module()` with the module name:

```
from python import Python  
  
# This is equivalent to Python's `import numpy as np`  
let np = Python.import_module("numpy")  
  
# Now use numpy as if writing in Python  
array = np.array([1, 2, 3])  
print(array)  
[1 2 3]
```

Yes, this imports Python NumPy, and you can import *any other Python module*.

Currently, you cannot import individual members (such as a single Python class or function)—you must import the whole Python module and then access members through the module name.

Mojo types in Python

Mojo primitive types implicitly convert into Python objects. Today we support lists, tuples, integers, floats, booleans, and strings.

For example, given this Python function that prints Python types:

```
%%python
def type_printer(my_list, my_tuple, my_int, my_string, my_float):
    print(type(my_list))
    print(type(my_tuple))
    print(type(my_int))
    print(type(my_string))
    print(type(my_float))
```

You can pass the Python function Mojo types with no problem:

```
type_printer([0, 3], (False, True), 4, "orange", 3.4)
<class 'list'>
<class 'tuple'>
<class 'int'>
<class 'str'>
<class 'float'>
```

Notice that in a Jupyter notebook, the Python function declared above is automatically available to any Mojo code in following code cells.

Mojo doesn't have a standard Dictionary yet, so it is not yet possible to create a Python dictionary from a Mojo dictionary. You can work with Python dictionaries in Mojo though! To create a Python dictionary, use the `dict` method:

```
from python import Python
from python.object import PythonObject

var dictionary = Python.dict()
dictionary["fruit"] = "apple"
dictionary["starch"] = "potato"

var keys: PythonObject = ["fruit", "starch", "protein"]
var N: Int = keys.__len__().__index__()
print(N, "items")

for i in range(N):
    if Python.is_type(dictionary.get(keys[i]), Python.none()):
        print(keys[i], "is not in dictionary")
    else:
        print(keys[i], "is included")
```

3 items
fruit is included
starch is included
protein is not in dictionary

Importing local Python modules

If you have some local Python code you want to use in Mojo, just add the directory to the Python path and then import the module.

For example, suppose you have a Python file named `mypython.py`:

```
import numpy as np
```

```
def my_algorithm(a, b):
    array_a = np.random.rand(a, a)
    return array_a + b
```

Here's how you can import it and use it in a Mojo file:

```
from python import Python
Python.add_to_path("path/to/module")
let mypython = Python.import_module("mypython")

let c = mypython.my_algorithm(2, 3)
print(c)
```

There's no need to worry about memory management when using Python in Mojo. Everything just works because Mojo was designed for Python from the beginning.

Parameterization: compile-time metaprogramming

One of Python's most amazing features is its extensible runtime metaprogramming features. This has enabled a wide range of libraries and provides a flexible and extensible programming model that Python programmers everywhere benefit from. Unfortunately, these features also come at a cost: because they are evaluated at runtime, they directly impact run-time efficiency of the underlying code. Because they are not known to the IDE, it is difficult for IDE features like code completion to understand them and use them to improve the developer experience.

Outside the Python ecosystem, static metaprogramming is also an important part of development, enabling the development of new programming paradigms and advanced libraries. There are many examples of prior art in this space, with different tradeoffs, for example:

1. Preprocessors (e.g. C preprocessor, Lex/YACC, etc) are perhaps the heaviest handed. They are fully general but the worst in terms of developer experience and tools integration.
2. Some languages (like Lisp and Rust) support (sometimes “hygienic”) macro expansion features, enabling syntactic extension and boilerplate reduction with somewhat better tooling integration.
3. Some older languages like C++ have very large and complex metaprogramming languages (templates) that are a dual to the *runtime* language. These are notably difficult to learn and have poor compile times and error messages.
4. Some languages (like Swift) build many features into the core language in a first-class way to provide good ergonomics for common cases at the expense of generality.
5. Some newer languages like Zig integrate a language interpreter into the compilation flow, and allow the interpreter to reflect over the AST as it is compiled. This allows many of the same features as a macro system with better extensibility and generality.

For Modular's work in AI, high-performance machine learning kernels, and accelerators, we need high abstraction capabilities provided by advanced metaprogramming systems. We needed high-level zero-cost abstractions, expressive libraries, and large-scale integration of multiple variants of algorithms. We want library developers to be able to extend the system, just like they do in Python, providing an extensible developer platform.

That said, we are not willing to sacrifice developer experience (including compile times and error messages) nor are we interested in building a parallel language ecosystem that is difficult to teach. We can learn from these previous systems but also have new technologies to build on top of, including MLIR and fine-grained language-integrated caching technologies.

As such, Mojo supports compile-time metaprogramming built into the compiler as a separate stage of compilation—after parsing, semantic analysis, and IR generation, but before lowering to

target-specific code. It uses the same host language for runtime programs as it does for metaprograms, and leverages MLIR to represent and evaluate these programs predictably.

Let's take a look at some simple examples.

About “parameters”: Python developers use the words “arguments” and “parameters” fairly interchangeably for “things that are passed into functions.” We decided to reclaim “parameter” and “parameter expression” to represent a compile-time value in Mojo, and continue to use “argument” and “expression” to refer to runtime values. This allows us to align around words like “parameterized” and “parametric” for compile-time metaprogramming.

Defining parameterized types and functions

You can parameterize structs and functions by specifying parameter names and types in square brackets (using an extended version of the [PEP695 syntax](#)). Unlike argument values, parameter values are known at compile-time, which enables an additional level of abstraction and code reuse, plus compiler optimizations such as [autotuning](#).

For instance, let's look at a [SIMD](#) type, which represents a low-level vector register in hardware that holds multiple instances of a scalar data-type. Hardware accelerators are constantly introducing new vector data types, and even CPUs may have 512-bit or longer SIMD vectors. In order to access the SIMD instructions on these processors, the data must be shaped into the proper SIMD width (data type) and length (vector size).

However, it's not feasible to define all the different SIMD variations with Mojo's built-in types. So, Mojo's SIMD type (defined as a struct) exposes the common SIMD operations in its methods, and makes the SIMD data type and size values parametric. This allows you to directly map your data to the SIMD vectors on any hardware.

Here is a cut-down (non-functional) version of Mojo's SIMD type definition:

```
struct SIMD[type: DType, size: Int]:  
    var value: ... # Some low-level MLIR stuff here  
  
    # Create a new SIMD from a number of scalars  
    fn __init__(inout self, *elems: SIMD[type, 1]): ...  
  
    # Fill a SIMD with a duplicated scalar value.  
    @staticmethod  
    fn splat(x: SIMD[type, 1]) -> SIMD[type, size]: ...  
  
    # Cast the elements of the SIMD to a different elt type.  
    fn cast[target: DType](self) -> SIMD[target, size]: ...  
  
    # Many standard operators are supported.  
    fn __add__(self, rhs: Self) -> Self: ...
```

Defining each SIMD variant with parameters is great for code reuse because the SIMD type can express all the different vector variants statically, instead of requiring the language to pre-define every variant.

Because SIMD is a parameterized type, the self argument in its functions carries those parameters—the full type name is SIMD[type, size]. Although it's valid to write this out (as shown in the return type of `splat()`), this can be verbose, so we recommend using the `Self` type (from [PEP673](#)) like the `__add__` example does.

Overloading on parameters

Functions and methods can be overloaded on their parameter signatures. The overload resolution logic filters for candidates according to the following rules, in order of precedence:

1. Candidates with the minimal number of implicit conversions (in both arguments and parameters).

2. Candidates without variadic arguments.
3. Candidates without variadic parameters.
4. Candidates with the shortest parameter signature.
5. Non-@staticmethod candidates (over @staticmethod ones, if available).

If there is more than one candidate after applying these rules, the overload resolution fails. For example:

```
@register_passable("trivial")
struct MyInt:
    """A type that is implicitly convertible to `Int`."""
    var value: Int

    @always_inline("nodebug")
    fn __init__(_a: Int) -> Self:
        return Self {value: _a}

fn foo[x: MyInt, a: Int]():
    print("foo[x: MyInt, a: Int]()")

fn foo[x: MyInt, y: MyInt]():
    print("foo[x: MyInt, y: MyInt]()")

fn bar[a: Int](b: Int):
    print("bar[a: Int](b: Int)")

fn bar[a: Int](*b: Int):
    print("bar[a: Int](*b: Int)")

fn bar[*a: Int](b: Int):
    print("bar[*a: Int](b: Int)")

fn parameter_overloads[a: Int, b: Int, x: MyInt]():
    # `foo[x: MyInt, a: Int]()` is called because it requires no implicit
    # conversions, whereas `foo[x: MyInt, y: MyInt]()` requires one.
    foo[x, a]()

    # `bar[a: Int](b: Int)` is called because it does not have variadic
    # arguments or parameters.
    bar[a](b)

    # `bar[*a: Int](b: Int)` is called because it has variadic parameters.
    bar[a, a, a](b)

parameter_overloads[1, 2, MyInt(3)]()

struct MyStruct:
    fn __init__(inout self):
        pass

    fn foo(inout self):
        print("calling instance method")

    @staticmethod
    fn foo():
        print("calling static method")

fn test_static_overload():
    var a = MyStruct()
    # `foo(inout self)` takes precedence over a static method.
    a.foo()
```

```
foo[x: MyInt, a: Int]()
bar[a: Int](b: Int)
bar[*a: Int](b: Int)
```

Using parameterized types and functions

You can instantiate parametric types and functions by passing values to the parameters in square brackets. For example, for the SIMD type above, `type` specifies the data type and `size` specifies the length of the SIMD vector (it must be a power of 2):

```

# Make a vector of 4 floats.
let small_vec = SIMD[DType.float32, 4](1.0, 2.0, 3.0, 4.0)

# Make a big vector containing 1.0 in float16 format.
let big_vec = SIMD[DType.float16, 32].splat(1.0)

# Do some math and convert the elements to float32.
let bigger_vec = (big_vec+big_vec).cast[DType.float32]()

# You can write types out explicitly if you want of course.
let bigger_vec2 : SIMD[DType.float32, 32] = bigger_vec

print('small_vec type:', small_vec.element_type, 'length:', len(small_vec))
print('bigger_vec2 type:', bigger_vec2.element_type, 'length:', len(bigger_vec2))
```

small_vec type: float32 length: 4
bigger_vec2 type: float32 length: 32

Note that the `cast()` method also needs a parameter to specify the type you want from the `cast` (the method definition above expects a target parametric value). Thus, just like how the `SIMD` struct is a generic type definition, the `cast()` method is a generic method definition that gets instantiated at compile-time instead of runtime, based on the parameter value.

The code above shows the use of concrete types (that is, it instantiates `SIMD` using known type values), but the major power of parameters comes from the ability to define parametric algorithms and types (code that uses the parameter values). For example, here's how to define a parametric algorithm with `SIMD` that is type- and width-agnostic:

```

from math import sqrt

fn rsqrt[dt: DType, width: Int](x: SIMD[dt, width]) -> SIMD[dt, width]:
    return 1 / sqrt(x)

print(rsqrt[DType.float16, 4](42))
```

[0.154296875, 0.154296875, 0.154296875, 0.154296875]

Notice that the `x` argument is actually a `SIMD` type based on the function parameters. The runtime program can use the value of parameters, because the parameters are resolved at compile-time before they are needed by the runtime program (but compile-time parameter expressions cannot use runtime values).

The Mojo compiler is also smart about type inference with parameters. Note that the above function is able to call the parametric `sqrt[1]()` function without specifying the parameters—the compiler infers its parameters based on the parametric `x` value passed into it, as if you wrote `sqrt[dt, width](x)` explicitly. Also note that `rsqrt()` chose to define its first parameter named `width` even though the `SIMD` type names it `size`, and there is no problem.

Using default parameter values

Just like how you can specify [default argument values](#) (in function arguments), you can also specify default values for parameters (in both function and struct parameters).

For example, here's a function with two parameters, each with a default value:

```

fn foo[a: Int = 3, msg: StringLiteral = "woof"]():
    print(msg, a)

fn use_defaults():
    foo()           # prints 'woof 3'
    foo[5]()        # prints 'woof 5'
    foo[7, "meow"]() # prints 'meow 7'
```

Recall that Mojo can infer parameter values in a parametric function, based on the parametric values attached to an argument value (see the `rsqrt[]()` example above). If the parametric function also has a default value defined, then the inferred parameter type takes precedence.

For example, in the following code, we update the parametric `foo[]()` function to take an argument with a parametric type. Although the function has a default parameter value for `a`, Mojo instead uses the inferred a parameter value from the `bar` argument (as written, the default a value can never be used, but this is just for demonstration purposes):

```
@value
struct Bar[v: Int]:
    pass

fn foo[a: Int = 3, msg: StringLiteral = "woof"](bar: Bar[a]):
    print(msg, a)

fn use_inferred():
    foo(Bar[9]()) # prints 'woof 9' □
```

And here's an example of default parameters in a struct:

```
@value
struct DefaultParams[msg: StringLiteral = "woof"]:
    alias message = msg

fn use_struct_default():
    print(DefaultParams[]().message)      # prints 'woof'
    print(DefaultParams["meow"]().message) # prints 'meow' □
```

Parameter expressions are just Mojo code

A parameter expression is any code expression (such as `a+b`) that occurs where a parameter is expected. Parameter expressions support operators and function calls, just like runtime code, and all parameter types use the same type system as the runtime program (such as `Int` and `DType`).

Because parameter expressions use the same grammar and types as runtime Mojo code, you can use many “dependent type” features. For example, you might want to define a helper function to concatenate two SIMD vectors:

```
fn concat[ty: DType, len1: Int, len2: Int](
    lhs: SIMD[ty, len1], rhs: SIMD[ty, len2]) -> SIMD[ty, len1+len2]:

    var result = SIMD[ty, len1 + len2]()
    for i in range(len1):
        result[i] = SIMD[ty, 1](lhs[i])
    for j in range(len2):
        result[len1 + j] = SIMD[ty, 1](rhs[j])
    return result

let a = SIMD[DType.float32, 2](1, 2)
let x = concat[DType.float32, 2, 2](a, a)

print('result type:', x.element_type, 'length:', len(x)) □
result type: float32 length: 4
```

Note how the resulting length is the sum of the input vector lengths, and you can express that with a simple `+` operation. For a more complex example, take a look at the [SIMD.shuffle\(\)](#) method in the standard library: it takes two input SIMD values, a vector shuffle mask as a list, and returns a SIMD that matches the length of the shuffle mask.

Powerful compile-time programming

While simple expressions are useful, sometimes you want to write imperative compile-time logic with control flow. For example, the `isclose()` function in the Mojo Math module uses exact equality for integers but “close” comparison for floating-point. You can even do compile-time recursion. For instance, here is an example “tree reduction” algorithm that sums all elements of a vector recursively into a scalar:

```

fn slice[ty: DType, new_size: Int, size: Int]{
    x: SIMD[ty, size], offset: Int) -> SIMD[ty, new_size]:
    var result = SIMD[ty, new_size]()
    for i in range(new_size):
        result[i] = SIMD[ty, 1](x[i + offset])
    return result

fn reduce_add[ty: DType, size: Int](x: SIMD[ty, size]) -> Int:
    @parameter
    if size == 1:
        return x[0].to_int()
    elif size == 2:
        return x[0].to_int() + x[1].to_int()

    # Extract the top/bottom halves, add them, sum the elements.
    alias half_size = size // 2
    let lhs = slice[ty, half_size, size](x, 0)
    let rhs = slice[ty, half_size, size](x, half_size)
    return reduce_add[ty, half_size](lhs + rhs)

let x = SIMD[DType.index, 4](1, 2, 3, 4)
print(x)
print("Elements sum:", reduce_add[DType.index, 4](x))
```

[1, 2, 3, 4]
Elements sum: 10

This makes use of the `@parameter if` feature, which is an `if` statement that runs at compile-time. It requires that its condition be a valid parameter expression, and ensures that only the live branch of the `if` statement is compiled into the program.

Mojo types are just parameter expressions

While we've shown how you can use parameter expressions within types, type annotations can themselves be arbitrary expressions (just like in Python). Types in Mojo have a special metatype type, allowing type-parametric algorithms and functions to be defined.

For example, we can create a simplified `Array` that supports arbitrary types of the elements (via the `AnyType` parameter):

```

struct Array[T: AnyType]:
    var data: Pointer[T]
    var size: Int

    fn __init__(inout self, size: Int, value: T):
        self.size = size
        self.data = Pointer[T].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, value)

    fn __getitem__(self, i: Int) -> T:
        return self.data.load(i)

    fn __del__(owned self):
        self.data.free()

var v = Array[Float32](4, 3.14)
print(v[0], v[1], v[2], v[3])
```

3.1400001049041748 3.1400001049041748 3.1400001049041748 3.1400001049041748

Notice that the `T` parameter is being used as the formal type for the `value` arguments and the return type of the `__getitem__` function. Parameters allow the `Array` type to provide different APIs based on the different use-cases.

There are many other cases that benefit from more advanced use of parameters. For example, you can execute a closure `N` times in parallel, feeding in a value from the context, like this:

```

fn parallelize[func: fn (Int) -> None](num_work_items: Int):
    # Not actually parallel: see the 'algorithm' module for real implementation.
```

```
for i in range(num_work_items):
    func(i)
```

Another example where this is important is with variadic generics, where an algorithm or data structure may need to be defined over a list of heterogeneous types such as for a tuple:

```
struct Tuple[*Ts: AnyType]:
    var _storage : *Ts
```

And although we don't have enough metatype helpers in place yet, we should be able to write something like this in the future (though overloading is still a better way to handle this):

```
struct Array[T: AnyType]:
    fn __getitem__[IndexType: AnyType](self, idx: IndexType)
        -> (ArraySlice[T] if issubclass(IndexType, Range) else T):
    ...
```

alias: named parameter expressions

It is very common to want to *name* compile-time values. Whereas `var` defines a runtime value, and `let` defines a runtime constant, we need a way to define a compile-time temporary value. For this, Mojo uses an `alias` declaration.

For example, the `DTType` struct implements a simple enum using aliases for the enumerators like this (the actual `DTType` implementation details vary a bit):

```
struct DTType:
    var value : UI8
    alias invalid = DTType(0)
    alias bool = DTType(1)
    alias int8 = DTType(2)
    alias uint8 = DTType(3)
    alias int16 = DTType(4)
    alias int16 = DTType(5)
    ...
    alias float32 = DTType(15)
```

This allows clients to use `DTType.float32` as a parameter expression (which also works as a runtime value) naturally. Note that this is invoking the runtime constructor for `DTType` at compile-time.

Types are another common use for `alias`. Because types are compile-time expressions, it is handy to be able to do things like this:

```
alias Float16 = SIMD[DTType.float16, 1]
alias UInt8 = SIMD[DTType.uint8, 1]

var x : Float16 # FLoat16 works like a "typedef"
```

Like `var` and `let`, aliases obey scope, and you can use local aliases within functions as you'd expect.

By the way, both `None` and `AnyType` are defined as [type aliases](#).

Autotuning / Adaptive compilation

Mojo parameter expressions allow you to write portable parametric algorithms like you can do in other languages, but when writing high-performance code you still have to pick concrete values to use for the parameters. For example, when writing high-performance numeric algorithms, you might want to use memory tiling to accelerate the algorithm, but the dimensions to use depend highly on the available hardware features, the sizes of the cache, what gets fused into the kernel, and many other fiddly details.

Even vector length can be difficult to manage, because the vector length of a typical machine depends on the datatype, and some datatypes like `bfloat16` don't have full support on all implementations. Mojo helps by providing an `autotune` function in the standard library. For

example if you want to write a vector-length-agnostic algorithm to a buffer of data, you might write it like this:

```
from autotune import autotune, search
from benchmark import Benchmark
from memory.unsafe import DTypePointer
from algorithm import vectorize

fn buffer_elementwise_add_impl[
    dt: DType
](lhs: DTypePointer[dt], rhs: DTypePointer[dt], result: DTypePointer[dt], N: Int):
    """Perform elementwise addition of N elements in RHS and LHS and store
    the result in RESULT.
    """
    @parameter
    fn add_simd[size: Int](idx: Int):
        let lhs_simd = lhssimd_load[size](idx)
        let rhs_simd = rhssimd_load[size](idx)
        resultsimd_store[size](idx, lhs_simd + rhs_simd)

    # Pick vector length for this dtype and hardware
    alias vector_len = autotune(1, 4, 8, 16, 32)

    # Use it as the vectorization length
    vectorize[vector_len, add_simd](N)

fn elementwise_evaluator[dt: DType](
    fns: Pointer[fn (DTypePointer[dt], DTypePointer[dt], DTypePointer[dt], Int) -> None],
    num: Int,
) -> Int:
    # Benchmark the implementations on N = 64.
    alias N = 64
    let lhs = DTypePointer[dt].alloc(N)
    let rhs = DTypePointer[dt].alloc(N)
    let result = DTypePointer[dt].alloc(N)

    # Fill with ones.
    for i in range(N):
        lhs.store(i, 1)
        rhs.store(i, 1)

    # Find the fastest implementation.
    var best_idx: Int = -1
    var best_time: Int = -1
    for i in range(num):
        @parameter
        fn wrapper():
            fns.load(i)(lhs, rhs, result, N)
        let cur_time = Benchmark(1).run[wrapper]()
        if best_idx < 0 or best_time > cur_time:
            best_idx = i
            best_time = cur_time
        print("time[", i, "] =", cur_time)
    print("selected:", best_idx)
    return best_idx

fn buffer_elementwise_add[
    dt: DType
](lhs: DTypePointer[dt], rhs: DTypePointer[dt], result: DTypePointer[dt], N: Int):
    # Forward declare the result parameter.
    alias best_impl: fn(DTypePointer[dt], DTypePointer[dt], DTypePointer[dt], Int) -> None

    # Perform search!
    search[
        fn(DTypePointer[dt], DTypePointer[dt], DTypePointer[dt], Int) -> None,
        buffer_elementwise_add_impl[dt],
        elementwise_evaluator[dt] -> best_impl
    ]()

    # Call the select implementation
    best_impl(lhs, rhs, result, N)
```

We can now call our function as usual:

```

let N = 32
let a = DTypePointer[DType.float32].alloc(N)
let b = DTypePointer[DType.float32].alloc(N)
let res = DTypePointer[DType.float32].alloc(N)
# Initialize arrays with some values
for i in range(N):
    a.store(i, 2.0)
    b.store(i, 40.0)
    res.store(i, -1)

buffer_elementwise_add[DType.float32](a, b, res, N)
print(a.load(10), b.load(10), res.load(10))
```

```

time[ 0 ] = 23
time[ 1 ] = 6
time[ 2 ] = 4
time[ 3 ] = 3
time[ 4 ] = 4
selected: 3
2.0 40.0 42.0
```

When compiling instantiations of this code, Mojo forks compilation of this algorithm and decides which value to use by measuring what works best in practice for the target hardware. It evaluates the different values of the `vector_len` expression and picks the fastest one according to a user-defined performance evaluator. Because it measures and evaluates each option individually, it might pick a different vector length for `float32` than for `int8`, for example. This simple feature is pretty powerful—going beyond simple integer constants—because functions and types are also parameter expressions.

Notice that the search for the best vector length is performed by the [search\(\)](#) function. `search()` takes an evaluator and a forked function and returns the fastest implementation selected by the evaluator as a parameter result. For a deeper dive on this topic, check out the notebooks about [Matrix Multiplication](#) and [Fast Memset in Mojo](#).

Autotuning is an inherently exponential technique that benefits from internal implementation details of the Mojo compiler stack (particularly MLIR, integrated caching, and distribution of compilation). This is also a power-user feature and needs continued development and iteration over time.

“Value Lifecycle”: Birth, life and death of a value

At this point, you should understand the core semantics and features for Mojo functions and types, so we can now discuss how they fit together to express new types in Mojo.

Many existing languages express design points with different tradeoffs: C++, for example, is very powerful but often accused of “getting the defaults wrong” which leads to bugs and mis-features. Swift is easy to work with, but has a less predictable model that copies values a lot and is dependent on an “ARC optimizer” for performance. Rust started with strong value ownership goals to satisfy its borrow checker, but relies on values being movable, which makes it challenging to express custom move constructors and can put a lot of stress on `memcpy` performance. In Python, everything is a reference to a class, so it never really faces issues with types.

For Mojo, we’ve learned from these existing systems, and we aim to provide a model that’s very powerful while still easy to learn and understand. We also don’t want to require “best effort” and difficult-to-predict optimization passes built into a “sufficiently smart” compiler.

To explore these issues, we look at different value classifications and the relevant Mojo features that go into expressing them, and build from the bottom-up. We use C++ as the primary comparison point in examples because it is widely known, but we occasionally reference other languages if they provide a better comparison point.

Types that cannot be instantiated

The most bare-bones type in Mojo is one that doesn't allow you to create instances of it: these types have no initializer at all, and if they have a destructor, it will never be invoked (because there cannot be instances to destroy):

```
struct NoInstances:  
    var state: Int # Pretty useless  
  
    alias my_int = Int  
  
    @staticmethod  
    fn print_hello():  
        print("hello world")
```

Mojo types do not get default constructors, move constructors, member-wise initializers or anything else by default, so it is impossible to create an instance of this `NoInstances` type. In order to get them, you need to define an `__init__` method or use a decorator that synthesizes an initializer. As shown, these types can be useful as "namespaces" because you can refer to static members like `NoInstances.my_int` or `NoInstances.print_hello()` even though you cannot instantiate an instance of the type.

Non-movable and non-copyable types

If we take a step up the ladder of sophistication, we'll get to types that can be instantiated, but once they are pinned to an address in memory, they cannot be implicitly moved or copied. This can be useful to implement types like atomic operations (such as `std::atomic` in C++) or other types where the memory address of the value is its identity and is critical to its purpose:

```
struct Atomic:  
    var state: Int  
  
    fn __init__(inout self, state: Int = 0):  
        self.state = state  
  
    fn __iadd__(inout self, rhs: Int):  
        #...atomic magic...  
  
    fn get_value(self) -> Int:  
        return atomic_load_int(self.state)
```

This class defines an initializer but no copy or move constructors, so once it is initialized it can never be moved or copied. This is safe and useful because Mojo's ownership system is fully "address correct" - when this is initialized onto the stack or in the field of some other type, it never needs to move.

Note that Mojo's approach controls only the built-in move operations, such as `a = b` copies and the [^transfer operator](#). One useful pattern you can use for your own types (like `Atomic` above) is to add an explicit `copy()` method (a non-“dunder” method). This can be useful to make explicit copies of an instance when it is known safe to the programmer.

Unique “move-only” types

If we take one more step up the ladder of capabilities, we will encounter types that are "unique" - there are many examples of this in C++, such as types like `std::unique_ptr` or even a `FileDescriptor` type that owns an underlying POSIX file descriptor. These types are pervasive in languages like Rust, where copying is discouraged, but "move" is free. In Mojo, you can implement these kinds of moves by defining the `__moveinit__` method to take ownership of a unique type. For example:

```
# This is a simple wrapper around POSIX-style fcntl.h functions.  
struct FileDescriptor:  
    var fd: Int  
  
    # This is how we move our unique type.  
    fn __moveinit__(inout self, owned existing: Self):  
        self.fd = existing.fd
```

```

# This takes ownership of a POSIX file descriptor.
fn __init__(inout self, fd: Int):
    self.fd = fd

fn __init__(inout self, path: String):
    # Error handling omitted, call the open(2) syscall.
    self = FileDescriptor(open(path, ...))

fn __del__(owned self):
    close(self.fd)  # pseudo code, call close(2)

fn dup(self) -> Self:
    # Invoke the dup(2) system call.
    return Self(dup(self.fd))

fn read(...): ...
fn write(...): ..._
```

The consuming move constructor (`__moveinit__`) takes ownership of an existing `FileDescriptor`, and moves its internal implementation details over to a new instance. This is because instances of `FileDescriptor` may exist at different locations, and they can be logically moved around—stealing the body of one value and moving it into another.

Here is an egregious example that will invoke `__moveinit__` multiple times:

```

fn egregious_moves(owned fd1: FileDescriptor):
    # fd1 and fd2 have different addresses in memory, but the
    # transfer operator moves unique ownership from fd1 to fd2.
    let fd2 = fd1^

    # Do it again, a use of fd2 after this point will produce an error.
    let fd3 = fd2^

    # We can do this all day...
    let fd4 = fd3^
    fd4.read(...)

    # fd4.__del__() runs here_
```

Note how ownership of the value is transferred between various values that own it, using the postfix-`^` “transfer” operator, which destroys a previous binding and transfer ownership to a new constant. If you are familiar with C++, the simple way to think about the transfer operator is like `std::move`, but in this case, we can see that it is able to move things without resetting them to a state that can be destroyed: in C++, if your move operator failed to change the old value’s `fd` instance, it would get closed twice.

Mojo tracks the liveness of values and allows you to define custom move constructors. This is rarely needed, but extremely powerful when it is. For example, some types like the [llvm::SmallVector_type](#) use the “inline storage” optimization technique, and they may want to be implemented with an “inner pointer” into their instance. This is a well-known trick to reduce pressure on the malloc memory allocator, but it means that a “move” operation needs custom logic to update the pointer when that happens.

With Mojo, this is as simple as implementing a custom `__moveinit__` method. This is something that is also easy to implement in C++ (though, with boilerplate in the cases where you don’t need custom logic) but is difficult to implement in other popular memory-safe languages.

One additional note is that while the Mojo compiler provides good predictability and control, it is also very sophisticated. It reserves the right to eliminate temporaries and the corresponding copy/move operations. If this is inappropriate for your type, you should use explicit methods like `copy()` instead of the dunder methods.

Types that support a “taking move”

One challenge with memory-safe languages is that they need to provide a predictable programming model around what the compiler is able to track, and static analysis in a compiler is inherently limited. For example, while it is possible for a compiler to understand that the two

array accesses in the first example below are to different array elements, it is (in general) impossible to reason about the second example (this is C++ code):

```
std::pair<T, T> getValues1(MutableArray<T> &array) {
    return { std::move(array[0]), std::move(array[1]) };
}
std::pair<T, T> getValues2(MutableArray<T> &array, size_t i, size_t j) {
    return { std::move(array[i]), std::move(array[j]) };
}
```

The problem here is that there is simply no way (looking at just the function body above) to know or prove that the dynamic values of *i* and *j* are not the same. While it is possible to maintain dynamic state to track whether individual elements of the array are live, this often causes significant runtime expense (even when move/transfers are not used), which is something that Mojo and other systems programming languages are not keen to do. There are a variety of ways to deal with this, including some pretty complicated solutions that aren't always easy to learn.

Mojo takes a pragmatic approach to let Mojo programmers get their job done without having to work around its type system. As seen above, it doesn't force types to be copyable, movable, or even constructable, but it does want types to express their full contract, and it wants to enable fluent design patterns that programmers expect from languages like C++. The (well known) observation here is that many objects have contents that can be "taken away" without needing to disable their destructor, either because they have a "null state" (like an optional type or nullable pointer) or because they have a null value that is efficient to create and a no-op to destroy (e.g. `std::vector` can have a null pointer for its data).

To support these use-cases, the [^transfer operator](#) supports arbitrary LValues, and when applied to one, it invokes the "taking move constructor," which is spelled `_takeinit_`. This constructor must set up the new value to be in a live state, and it can mutate the old value, but it must put the old value into a state where its destructor still works. For example, if we want to put our `FileDescriptor` into a vector and move out of it, we might choose to extend it to know that -1 is a sentinel which means that it is "null". We can implement this like so:

```
# This is a simple wrapper around POSIX-style fcntl.h functions.
struct FileDescriptor:
    var fd: Int

    # This is the new key capability.
    fn _takeinit_(inout self, inout existing: Self):
        self.fd = existing.fd
        existing.fd = -1 # neutralize 'existing'.

    fn __moveinit__(inout self, owned existing: Self): # as above
    fn __init__(inout self, fd: Int): # as above
    fn __init__(inout self, path: String): # as above

    fn __del__(owned self):
        if self.fd != -1:
            close(self.fd) # pseudo code, call close(2)
```

Notice how the "stealing move" constructor takes the file descriptor from an existing value and mutates that value so that its destructor won't do anything. This technique has tradeoffs and is not the best for every type. We can see that it adds one (inexpensive) branch to the destructor because it has to check for the sentinel case. It is also generally considered bad form to make types like this nullable because a more general feature like an `Optional[T]` type is a better way to handle this.

Furthermore, we plan to implement `Optional[T]` in Mojo itself, and `Optional` needs this functionality. We also believe that the library authors understand their domain problem better than language designers do, and generally prefer to give library authors full power over that domain. As such you can choose (but don't have to) to make your types participate in this behavior in an opt-in way.

Copyable types

The next step up from movable types are copyable types. Copyable types are also very common - programmers generally expect things like strings and arrays to be copyable, and every Python Object reference is copyable - by copying the pointer and adjusting the reference count.

There are many ways to implement copyable types. One can implement reference semantic types like Python or Java, where you propagate shared pointers around, one can use immutable data structures that are easily shareable because they are never mutated once created, and one can implement deep value semantics through lazy copy-on-write as Swift does. Each of these approaches has different tradeoffs, and Mojo takes the opinion that while we want a few common sets of collection types, we can also support a wide range of specialized ones that focus on particular use cases.

In Mojo, you can do this by implementing the `__copyinit__` method. Here is an example of that using a simple `String` (in pseudo-code):

```
struct MyString:  
    var data: Pointer[UI8]  
  
    # StringRef is a pointer + length and works with StringLiteral.  
    def __init__(inout self, input: StringRef):  
        self.data = ...  
  
    # Copy the string by deep copying the underlying malloc'd data.  
    def __copyinit__(inout self, existing: Self):  
        self.data = strdup(existing.data)  
  
    # This isn't required, but optimizes unneeded copies.  
    def __moveinit__(inout self, owned existing: Self):  
        self.data = existing.data  
  
    def __del__(owned self):  
        free(self.data.address)  
  
    def __add__(self, rhs: MyString) -> MyString: ...
```

This simple type is a pointer to a “null-terminated” string data allocated with `malloc`, using old-school C APIs for clarity. It implements the `__copyinit__`, which maintains the invariant that each instance of `MyString` owns its underlying pointer and frees it upon destruction. This implementation builds on tricks we’ve seen above, and implements a `__moveinit__` constructor, which allows it to completely eliminate temporary copies in some common cases. You can see this behavior in this code sequence:

```
fn test_my_string():  
    var s1 = MyString("hello ")  
  
    var s2 = s1      # s2.__copyinit__(s1) runs here  
  
    print(s1)  
  
    var s3 = s1^    # s3.__moveinit__(s1) runs here  
  
    print(s2)  
    # s2.__del__() runs here  
    print(s3)  
    # s3.__del__() runs here
```

In this case, you can see both why a copy constructor is needed: without one, the duplication of the `s1` value into `s2` would be an error - because you cannot have two live instances of the same non-copyable type. The move constructor is optional but helps the assignment into `s3`: without it, the compiler would invoke the copy constructor from `s1`, then destroy the old `s1` instance. This is logically correct but introduces extra runtime overhead.

Mojo destroys values eagerly, which allows it to transform copy+destroy pairs into single move operations, which can lead to much better performance than C++ without requiring the need for pervasive micromanagement of `std::move`.

Trivial types

The most flexible types are ones that are just “bags of bits”. These types are “trivial” because they can be copied, moved, and destroyed without invoking custom code. Types like these are arguably the most common basic type that surrounds us: things like integers and floating point values are all trivial. From a language perspective, Mojo doesn’t need special support for these, it would be perfectly fine for type authors to implement these things as no-ops, and allow the inliner to just make them go away.

There are two reasons that approach would be suboptimal: one is that we don’t want the boilerplate of having to define a bunch of methods on trivial types, and second, we don’t want the compile-time overhead of generating and pushing around a bunch of function calls, only to have them inline away to nothing. Furthermore, there is an orthogonal concern, which is that many of these types are trivial in another way: they are tiny, and should be passed around in the registers of a CPU, not indirectly in memory.

As such, Mojo provides a struct decorator that solves all of these problems. You can implement a type with the `@register_passable("trivial")` decorator, and this tells Mojo that the type should be copyable and movable but that it has no user-defined logic for doing this. It also tells Mojo to prefer to pass the value in CPU registers, which can lead to efficiency benefits.

TODO: This decorator is due for reconsideration. Lack of custom logic copy/move/destroy logic and “passability in a register” are orthogonal concerns and should be split. This former logic should be subsumed into a more general `@value("trivial")` decorator, which is orthogonal from `@register_passable`.

`@value` decorator

Mojo’s [value lifecycle](#) provides simple and predictable hooks that give you the ability to express exotic low-level things like Atomic correctly. This is great for control and for a simple programming model, but most structs are simple aggregations of other types, and we don’t want to write a lot of boilerplate for them. To solve this, Mojo provides a `@value` decorator for structs that synthesizes a lot of boilerplate for you.

You can think of `@value` as an extension of Python’s [@dataclass](#) that also handles Mojo’s `__moveinit__` and `__copyinit__` methods.

The `@value` decorator takes a look at the fields of your type, and generates some members that are missing. For example, consider a simple struct like this:

```
@value
struct MyPet:
    var name: String
    var age: Int
```

Mojo will notice that you do not have a member-wise initializer, a move constructor, or a copy constructor, and it will synthesize these for you as if you had written:

```
struct MyPet:
    var name: String
    var age: Int

fn __init__(inout self, owned name: String, age: Int):
    self.name = name^
    self.age = age

fn __copyinit__(inout self, existing: Self):
    self.name = existing.name
    self.age = existing.age

fn __moveinit__(inout self, owned existing: Self):
    self.name = existing.name^
    self.age = existing.age
```

When you add the `@value` decorator, Mojo synthesizes each of these special methods only when it doesn't exist. You can override the behavior of one or more by defining your own version. For example, it is fairly common to want a custom copy constructor but use the default member-wise and move constructor.

The arguments to `__init__` are all passed as owned arguments since the struct takes ownership and stores the value. This is a useful micro-optimization and enables the use of move-only types. Trivial types like `Int` are also passed as owned values, but since that doesn't mean anything for them, we elide the marker and the transfer operator (^) for clarity.

Note: If your type contains any [move-only](#) fields, Mojo will not generate a copy constructor because it cannot copy those fields. Further, the `@value` decorator only works on types whose members are copyable and/or movable. If you have something like `Atomic` in your struct, then it probably isn't a value type, and you don't want these members anyway.

Also notice that the `MyPet` struct above doesn't include the `__del__()` destructor—Mojo also synthesizes this, but it doesn't require the `@value` decorator (see the section below about [destructors](#)).

There is no way to suppress the generation of specific methods or customize generation at this time, but we can add arguments to the `@value` generator to do this if there is demand.

Behavior of destructors

Any struct in Mojo can have a destructor (a `__del__()` method), which is automatically run when the value's lifetime ends (typically the point at which the value is last used). For example, a simple string might look like this (in pseudo code):

```
@value
struct MyString:
    var data: Pointer[UInt8]

    def __init__(inout self, input: StringRef): ...
    def __add__(self, rhs: String) -> MyString: ...
    def __del__(owned self):
        free(self.data.address)
```

Mojo destroys values like `MyString` (it calls the `__del__()` destructor) using an **“As Soon As Possible”** (ASAP) policy that runs after every call. Mojo does *not* wait until the end of the code block to destroy unused values. Even in an expression like `a+b+c+d`, Mojo destroys the intermediate expressions eagerly, as soon as they are no longer needed—it does not wait until the end of the statement.

The Mojo compiler automatically invokes the destructor when a value is dead and provides strong guarantees about when the destructor is run. Mojo uses static compiler analysis to reason about your code and decide when to insert calls to the destructor. For example:

```
fn use_strings():
    var a = String("hello a")
    let b = String("hello b")
    print(a)
    # a.__del__() runs here for "hello a"

    print(b)
    # b.__del__() runs here

    a = String("temporary a")
    # a.__del__() runs here because "temporary a" is never used

    # Other stuff happens here

    a = String("final a")
    print(a)
    # a.__del__() runs again here for "final a"
```

```
use_strings() →
```

```
hello a  
hello b  
final a
```

In the code above, you'll see that the `a` and `b` values are created early on, and each initialization of a value is matched with a call to a destructor. Notice that `a` is destroyed multiple times—once for each time it receives a new value.

Now, this might be surprising to a C++ programmer, because it's different from the [RAII pattern](#) in which C++ destroys values at the end of a scope. Mojo also follows the principle that values acquire resources in a constructor and release resources in a destructor, but eager destruction in Mojo has a number of strong advantages over scope-based destruction in C++:

- The Mojo approach eliminates the need for types to implement re-assignment operators, like `operator=(const T&)` and `operator=(T&&)` in C++, making it easier to define types and eliminating a concept.
- Mojo does not allow mutable references to overlap with other mutable references or with immutable borrows. One major way that it provides a predictable programming model is by making sure that references to objects die as soon as possible, avoiding confusing situations where the compiler thinks a value could still be alive and interfere with another value, but that isn't clear to the user.
- Destroying values at last-use composes nicely with “move” optimization, which transforms a “copy+del” pair into a “move” operation, a generalization of C++ move optimizations like NRVO (named return value optimization).
- Destroying values at end-of-scope in C++ is problematic for some common patterns like tail recursion because the destructor calls happen after the tail call. This can be a significant performance and memory problem for certain functional programming patterns.

Importantly, Mojo's eager destruction also works well within Python-style `def` functions to provide destruction guarantees (without a garbage collector) at a fine-grain level—recall that Python doesn't really provide scopes beyond a function, so C++-style destruction in Mojo would be a lot less useful.

Note: Mojo also supports the Python-style [with statement](#), which provides more deliberately-scoped access to resources.

The Mojo approach is more similar to how Rust and Swift work, because they both have strong value ownership tracking and provide memory safety. One difference is that their implementations require the use of a [dynamic “drop flag”](#)—they maintain hidden shadow variables to keep track of the state of your values to provide safety. These are often optimized away, but the Mojo approach eliminates this overhead entirely, making the generated code faster and avoiding ambiguity.

Field-sensitive lifetime management

In addition to Mojo's lifetime analysis being fully control-flow aware, it is also fully field-sensitive (each field of a structure is tracked independently). That is, Mojo separately keeps track of whether a “whole object” is fully or only partially initialized/destroyed.

For example, consider this code:

```
@value  
struct TwoStrings:  
    var str1: String  
    var str2: String  
  
fn use_two_strings():
```

```
var ts = TwoStrings("foo", "bar")
print(ts.str1)
# ts.str1.__del__() runs here

# Other stuff happens here

ts.str1 = String("hello") # Overwrite ts.str1
print(ts.str1)
# ts.__del__() runs here
```

use_two_strings() 

```
foo
hello
```

Note that the `ts.str1` field is destroyed almost immediately, because Mojo knows that it will be overwritten down below. You can also see this when using the [transfer operator](#), for example:

```
fn consume(owned arg: String):
    pass

fn use(arg: TwoStrings):
    print(arg.str1)

fn consume_and_use_two_strings():
    var ts = TwoStrings("foo", "bar")
    consume(ts.str1^)
    # ts.str1.__moveinit__() runs here

    # ts is now only partially initialized here!

    ts.str1 = String("hello") # All together now
    use(ts)                 # This is ok
    # ts.__del__() runs here
```

consume_and_use_two_strings() 

```
hello
```

Notice that the code transfers ownership of the `str1` field: for the duration of `other_stuff()`, the `str1` field is completely uninitialized because ownership was transferred to `consume()`. Then `str1` is reinitialized before it is used by the `use()` function (if it weren't, Mojo would reject the code with an uninitialized field error).

Mojo's rule on this is powerful and intentionally straight-forward: fields can be temporarily transferred, but the "whole object" must be constructed with the aggregate type's initializer and destroyed with the aggregate destructor. This means that it isn't possible to create an object by initializing only its fields, nor is it possible to tear down an object by destroying only its fields. For example, this code does not compile:

```
fn consume_and_use_two_strings():
    let ts = TwoStrings("foo", "bar") # ts is initialized
    # Uncomment to see an error:
    # consume(ts.str1^)
    # Because `ts` is not used anymore, it should be destroyed here, but
    # the object is not whole, preventing the overall value from being destroyed

    let ts2 : TwoStrings # ts2 type is declared but not initialized
    ts2.str1 = String("foo")
    ts2.str2 = String("bar") # Both the member are initialized
    # Uncomment to see an error:
    # use(ts2) # Error: 'ts2' isn't fully initialized 
```

While we could allow patterns like this to happen, we reject this because a value is more than a sum of its parts. Consider a `FileDescriptor` that contains a POSIX file descriptor as an integer value: there is a big difference between destroying the integer (a no-op) and destroying the `FileDescriptor` (it might call the `close()` system call). Because of this, we require all full-value initialization to go through initializers and be destroyed with their full-value destructor.

For what it's worth, Mojo does internally have an equivalent of the Rust [mem::forget](#) function, which explicitly disables a destructor and has a corresponding internal feature for "blessing" an object, but they aren't exposed for user consumption at this point.

Field lifetimes in `_init_`

The behavior of an `_init_` method works almost like any other method—there is a small bit of magic: it knows that the fields of an object are uninitialized, but it believes the full object is initialized. This means that you can use `self` as a whole object as soon as all the fields are initialized:

```
fn use(arg: TwoStrings2):
    pass

struct TwoStrings2:
    var str1: String
    var str2: String

    fn __init__(inout self, cond: Bool, other: String):
        self.str1 = String()
        if cond:
            self.str2 = other
            use(self) # Safe to use immediately!
            # self.str2.__del__(): destroyed because overwritten below.

        self.str2 = self.str1
        use(self) # Safe to use immediately! □
```

Similarly, it's safe for initializers in Mojo to completely overwrite `self`, such as by delegating to other initializers:

```
struct TwoStrings3:
    var str1: String
    var str2: String

    fn __init__(inout self):
        self.str1 = String()
        self.str2 = String()

    fn __init__(inout self, one: String):
        self = TwoStrings3() # Delegate to the basic init
        self.str1 = one □
```

Field lifetimes of owned arguments in `_moveinit_` and `_del_`

A final bit of magic exists for the `owned` arguments of a `_moveinit_()` move initializer and a `_del_()` destructor. To recap, these method signatures look like this:

```
struct TwoStrings:
    ...
    fn __moveinit__(inout self, owned existing: Self):
        # Initializes a new `self` by consuming the contents of `existing`
    fn __del__(owned self):
        # Destroys all resources in `self` □
```

These methods face an interesting but obscure problem: both methods are in charge of dismantling the `owned` `existing`/`self` value. That is, `_moveinit_()` destroys sub-elements of `existing` in order to transfer ownership to a new instance, while `_del_()` implements the deletion logic for its `self`. As such, they both want to own and transform elements of the `owned` value, and they definitely don't want the `owned` value's destructor to also run (in the case of the `_del_()` method, that would turn into an infinite loop).

To solve this problem, Mojo handles these two methods specially by assuming that their whole values are destroyed upon reaching any return from the method. This means that the whole object may be used before the field values are transferred. For example, this works as you expect:

```

fn consume(owned str: String):
    print('Consumed', str)

struct TwoStrings4:
    var str1: String
    var str2: String

    fn __init__(inout self, one: String):
        self.str1 = one
        self.str2 = String("bar")

    fn __moveinit__(inout self, owned existing: Self):
        self.str1 = existing.str1
        self.str2 = existing.str2

    fn __del__(owned self):
        self.dump() # Self is still whole here
        # Mojo calls self.str2.__del__() since str2 isn't used anymore
        consume(self.str1^)
        # str1 has now been transferred;
        # `self.__del__()` is not called (avoiding an infinite loop).

    fn dump(inout self):
        print('str1:', self.str1)
        print('str2:', self.str2)

fn use_two_strings():
    let two_strings = TwoStrings4("foo")

# We use a function call to ensure the `two_strings` ownership is enforced
# (Currently, ownership is not enforced for top-level code in notebooks)
use_two_strings()_

```

str1: foo
str2: bar
Consumed foo

You should not generally have to think about this, but if you have logic with inner pointers into members, you may need to keep them alive for some logic within the destructor or move initializer itself. You can do this by assigning to the `_discard` pattern:

```

fn __del__(owned self):
    self.dump() # Self is still whole here

    consume(self.str1^)
    self.str2 =
    # self.str2.__del__(): Mojo destroys str2 after its last use.__

```

In this case, if `consume()` implicitly refers to some value in `str2` somehow, this will ensure that `str2` isn't destroyed until the last use when it is accessed by the `_discard` pattern.

Defining the `__del__` destructor

You should define the `__del__()` method to perform any kind of cleanup the type requires. Usually, that includes freeing memory for any fields that are not trivial or destructible—Mojo automatically destroys any trivial and destructible types as soon as they're not used anymore.

For example, consider this struct:

```

struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, owned name: String, age: Int):
        self.name = name^
        self.age = age_

```

There's no need to define the `__del__()` method because `String` is a destructible (it has its own `__del__()` method) and Mojo destroys it as soon as it's no longer used (which is exactly when the

MyPet instance is no longer used), and `Int` is a [trivial type](#) and Mojo reclaims this memory also as soon as possible (although a little differently, without need for a `__del__()` method).

Whereas, the following struct must define the `__del__()` method to free the memory allocated for its `Pointer`:

```
struct Array[Type: AnyType]:  
    var data: Pointer[Type]  
    var size: Int  
  
    fn __init__(inout self, size: Int, value: Type):  
        self.size = size  
        self.data = Pointer[Type].alloc(self.size)  
        for i in range(self.size):  
            self.data.store(i, value)  
  
    fn __del__(owned self):  
        self.data.free()□
```

Lifetimes

TODO: Explain how returning references work, tied into lifetimes which dovetail with parameters. This is not enabled yet.

Type traits

This is a feature very much like Rust traits or Swift protocols or Haskell type classes. Note, this is not implemented yet.

Advanced/Obscure Mojo features

This section describes power-user features that are important for building the bottom-est level of the standard library. This level of the stack is inhabited by narrow features that require experience with compiler internals to understand and utilize effectively.

`@register_passable` struct decorator

The default model for working with values is they live in memory, so they have an identity, which means they are passed indirectly to and from functions (equivalently, they are passed “by reference” at the machine level). This is great for types that cannot be moved, and is a safe default for large objects or things with expensive copy operations. However, it is inefficient for tiny things like a single integer or floating point number.

To solve this, Mojo allows structs to opt-in to being passed in a register instead of passing through memory with the `@register_passable` decorator. You’ll see this decorator on types like `Int` in the standard library:

```
@register_passable("trivial")  
struct Int:  
    var value: __mlir_type.`!pop.scalar<index>`  
  
    fn __init__(value: __mlir_type.`!pop.scalar<index>`) -> Self:  
        return Self {value: value}  
    ...□
```

The basic `@register_passable` decorator does not change the fundamental behavior of a type: it still needs to have a `__copyinit__` method to be copyable, may still have a `__init__` and `__del__` methods, etc. The major effect of this decorator is on internal implementation details: `@register_passable` types are typically passed in machine registers (subject to the details of the underlying architecture).

There are only a few observable effects of this decorator to the typical Mojo programmer:

1. `@register_passable` types are not able to hold instances of types that are not themselves `@register_passable`.
2. Instances of `@register_passable` types do not have predictable identity, and so the `self` pointer is not stable/predictable (e.g. in hash tables).
3. `@register_passable` arguments and result are exposed to C and C++ directly, instead of being passed by-pointer.
4. The `__init__` and `__copyinit__` methods of this type are implicitly static (like `__new__` in Python) and return their results by-value instead of taking `inout self`.

We expect that this decorator will be used pervasively on core standard library types, but is safe to ignore for general application level code.

The `Int` example above actually uses the “trivial” variant of this decorator. It changes the passing convention as described above but also disallows copy and move constructors and destructors (synthesizing them all trivially).

TODO: Trivial needs to be decoupled to its own decorator since it applies to memory types as well.

@always_inline decorator

`@always_inline("nodebug")`: same thing but without debug information so you don't step into the + method on `Int`.

@parameter decorator

The `@parameter` decorator can be placed on nested functions that capture runtime values to create “parametric” capturing closures. This is an unsafe feature in Mojo, because we do not currently model the lifetimes of capture-by-reference. A particular aspect of this feature is that it allows closures that capture runtime values to be passed as parameter values.

Magic operators

C++ code has a number of magic operators that intersect with value lifecycle, things like “placement new”, “placement delete” and “operator=” that reassign over an existing value. Mojo is a safe language when you use all its language features and compose on top of safe constructs, but of any stack is a world of C-style pointers and rampant unsafety. Mojo is a pragmatic language, and since we are interested in both interoperating with C/C++ and in implementing safe constructs like `String` directly in Mojo itself, we need a way to express unsafe things.

The Mojo standard library `Pointer[element_type]` type is implemented with an underlying `!kgen.pointer<element_type>` type in MLIR, and we desire a way to implement these C++-equivalent unsafe constructs in Mojo. Eventually, these will migrate to all being methods on the `Pointer` type, but until then, some need to be exposed as built-in operators.

Direct access to MLIR

Mojo provides full access to the MLIR dialects and ecosystem. Please take a look at the [Low level IR in Mojo](#) to learn how to use the `__mlir_type`, `__mlir_op`, and `__mlir_type` constructs. All of the built-in and standard library APIs are implemented by just calling the underlying MLIR constructs, and in doing so, Mojo effectively serves as syntax sugar on top of MLIR.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo[] notebooks

All the Jupyter notebooks we've created for the Mojo Playground.

The following pages are rendered from the Jupyter notebooks that are [available on GitHub](#) and in the [Mojo Playground](#).

[Low-level IR in Mojo](#)

[Learn how to use low-level primitives to define your own boolean type in Mojo.](#)

[Mandelbrot in Mojo with Python plots](#)

[Learn how to write high-performance Mojo code and import Python packages.](#)

[Matrix multiplication in Mojo](#)

[Learn how to leverage Mojo's various functions to write a high-performance matmul.](#)

[Fast memset in Mojo](#)

[Learn how to use Mojo's autotuning to quickly write a memset function.](#)

[Ray tracing in Mojo](#)

[Learn how to draw 3D graphics with ray-traced lighting using Mojo.](#)

No matching items

Low-level IR in Mojo

Learn how to use low-level primitives to define your own boolean type in Mojo.

Mojo is a high-level programming language with an extensive set of modern features. Mojo also provides you, the programmer, access to all of the low-level primitives that you need to write powerful – yet zero-cost – abstractions.

These primitives are implemented in [MLIR](#), an extensible intermediate representation (IR) format for compiler design. Many different programming languages and compilers translate their source programs into MLIR, and because Mojo provides direct access to MLIR features, this means Mojo programs can enjoy the benefits of each of these tools.

Going one step further, Mojo’s unique combination of zero-cost abstractions with MLIR interoperability means that Mojo programs can take full advantage of *anything* that interfaces with MLIR. While this isn’t something normal Mojo programmers may ever need to do, it’s an extremely powerful capability when extending a system to interface with a new datatype, or an esoteric new accelerator feature.

To illustrate these ideas, we’ll implement a boolean type in Mojo below, which we’ll call `OurBool`. We’ll make extensive use of MLIR, so let’s begin with a short primer.

What is MLIR?

MLIR is an intermediate representation of a program, not unlike an assembly language, in which a sequential set of instructions operate on in-memory values.

More importantly, MLIR is modular and extensible. MLIR is composed of an ever-growing number of “dialects.” Each dialect defines operations and optimizations: for example, the [‘math’ dialect](#) provides mathematical operations such as sine and cosine, the [‘amdgpu’ dialect](#) provides operations specific to AMD processors, and so on.

Each of MLIR’s dialects can interoperate with the others. This is why MLIR is said to unlock heterogeneous compute: as newer, faster processors and architectures are developed, new MLIR dialects are implemented to generate optimal code for those environments. Any new MLIR dialect can be translated seamlessly into other dialects, so as more get added, all existing MLIR becomes more powerful.

This means that our own custom types, such as the `OurBool` type we’ll create below, can be used to provide programmers with a high-level, Python-like interface. But “under the covers,” Mojo and MLIR will optimize our convenient, high-level types for each new processor that appears in the future.

There’s much more to write about why MLIR is such a revolutionary technology, but let’s get back to Mojo and defining the `OurBool` type. There will be opportunities to learn more about MLIR along the way.

Defining the `OurBool` type

We can use Mojo’s `struct` keyword to define a new type `OurBool`:

```
struct OurBool:  
    var value: __mlir_type.i1
```

A boolean can represent 0 or 1, “true” or “false.” To store this information, `OurBool` has a single member, called `value`. Its type is represented *directly in MLIR*, using the MLIR builtin type [i1](#). In fact, you can use any MLIR type in Mojo, by prefixing the type name with `__mlir_type`.

As we'll see below, representing our boolean value with `i1` will allow us to utilize all of the MLIR operations and optimizations that interface with the `i1` type – and there are many of them!

Having defined `OurBool`, we can now declare a variable of this type:

```
let a: OurBool
```

Leveraging MLIR

Naturally, we might next try to create an instance of `OurBool`. Attempting to do so at this point, however, results in an error:

```
let a = OurBool() # error: 'OurBool' does not implement an '__init__' method
```

As in Python, `__init__` is a [special method](#) that can be defined to customize the behavior of a type. We can implement an `__init__` method that takes no arguments, and returns an `OurBool` with a “false” value.

```
struct OurBool:  
    var value: __mlir_type.i1  
  
    fn __init__(inout self):  
        self.value = __mlir_op.`index.bool.constant`[  
            value=__mlir_attr.`false`,  
        ]()
```

To initialize the underlying `i1` value, we use an MLIR operation from its [‘index’ dialect](#), called `index.bool.constant`.

MLIR’s ‘index’ dialect provides us with operations for manipulating builtin MLIR types, such as the `i1` we use to store the value of `OurBool`. The `index.bool.constant` operation takes a true or false compile-time constant as input, and produces a runtime output of type `i1` with the given value.

So, as shown above, in addition to any MLIR type, Mojo also provides direct access to any MLIR operation via the `__mlir_op` prefix, and to any attribute via the `__mlir_attr` prefix. MLIR attributes are used to represent compile-time constants.

As you can see above, the syntax for interacting with MLIR isn’t always pretty: MLIR attributes are passed in between square brackets [...], and the operation is executed via a parentheses suffix (...), which can take runtime argument values. However, most Mojo programmers will not need to access MLIR directly, and for the few that do, this “ugly” syntax gives them superpowers: they can define high-level types that are easy to use, but that internally plug into MLIR and its powerful system of dialects.

We think this is very exciting, but let’s bring things back down to earth: having defined an `__init__` method, we can now create an instance of our `OurBool` type:

```
let b = OurBool()
```

Value semantics in Mojo

We can now instantiate `OurBool`, but using it is another story:

```
let a = OurBool()  
let b = a # error: 'OurBool' does not implement the '__copyinit__' method
```

Mojo uses “value semantics” by default, meaning that it expects to create a copy of a when assigning to `b`. However, Mojo doesn’t make any assumptions about *how* to copy `OurBool`, or its underlying `i1` value. The error indicates that we should implement a `__copyinit__` method, which would implement the copying logic.

In our case, however, `OurBool` is a very simple type, with only one “trivially copyable” member. We can use a decorator to tell the Mojo compiler that, saving us the trouble of defining our own `_copyinit_` boilerplate. Trivially copyable types must implement an `_init_` method that returns an instance of themselves, so we must also rewrite our initializer slightly.

```
@register_passable("trivial")
struct OurBool:
    var value: __mlir_type.i1

    fn __init__() -> Self:
        return Self {
            value: __mlir_op.`index.bool.constant`[
                value=__mlir_attr.`false`,
            ]()
        }
    }
```

We can now copy `OurBool` as we please:

```
let c = OurBool()
let d = c
```

Compile-time constants

It's not very useful to have a boolean type that can only represent “false.” Let’s define compile-time constants that represent true and false `OurBool` values.

First, let’s define another `_init_` constructor for `OurBool` that takes its `i1` value as an argument:

```
@register_passable("trivial")
struct OurBool:
    var value: __mlir_type.i1

    # ...

    fn __init__(value: __mlir_type.i1) -> Self:
        return Self {value: value}
```

This allows us to define compile-time constant `OurBool` values, using the `alias` keyword. First, let’s define `OurTrue`:

```
alias OurTrue = OurBool(__mlir_attr.`true`)
```

Here we’re passing in an MLIR compile-time constant value of `true`, which has the `i1` type that our new `_init_` constructor expects. We can use a slightly different syntax for `OurFalse`:

```
alias OurFalse: OurBool = __mlir_attr.`false`
```

`OurFalse` is declared to be of type `OurBool`, and then assigned an `i1` type – in this case, the `OurBool` constructor we added is called implicitly.

With `true` and `false` constants, we can also simplify our original `_init_` constructor for `OurBool`. Instead of constructing an MLIR value, we can simply return our `OurFalse` constant:

```
alias OurTrue = OurBool(__mlir_attr.`true`)
alias OurFalse: OurBool = __mlir_attr.`false`

@register_passable("trivial")
struct OurBool:
    var value: __mlir_type.i1

    # We can simplify our no-argument constructor:
    fn __init__() -> Self:
        return OurFalse

    fn __init__(value: __mlir_type.i1) -> Self:
        return Self {value: value}
```

Note also that we can define `OurTrue` before we define `OurBool`. The Mojo compiler is smart enough to figure this out.

With these constants, we can now define variables with both true and false values of `OurBool`:

```
let e = OurTrue
let f = OurFalse
```

Implementing `__bool__`

Of course, the reason booleans are ubiquitous in programming is because they can be used for program control flow. However, if we attempt to use `OurBool` in this way, we get an error:

```
let a = OurTrue
if a: print("It's true!") # error: 'OurBool' does not implement the '__bool__' method
```

When Mojo attempts to execute our program, it needs to be able to determine whether to print "It's true!" or not. It doesn't yet know that `OurBool` represents a boolean value – Mojo just sees a struct that is 1 bit in size. However, Mojo also provides interfaces that convey boolean qualities, which are the same as those used by Mojo's standard library types, like `Bool`. In practice, this means Mojo gives you full control: any type that's packaged with the language's standard library is one for which you could define your own version.

In the case of our error message, Mojo is telling us that implementing a `__bool__` method on `OurBool` would signify that it has boolean qualities.

Thankfully, `__bool__` is simple to implement: Mojo's standard library and builtin types are all implemented on top of MLIR, and so the builtin `Bool` type also defines a constructor that takes an `i1`, just like `OurBool`:

```
alias OurTrue = OurBool(__mlir_attr.`true`)
alias OurFalse: OurBool = __mlir_attr.`false`  
  
@register_passable("trivial")
struct OurBool:
    var value: __mlir_type.i1
    # ...
    fn __init__(value: __mlir_type.i1) -> Self:
        return Self {value: value}
    # Our new method converts `OurBool` to `Bool`:
    fn __bool__(self) -> Bool:
        return Bool(self.value)
```

Now we can use `OurBool` anywhere we can use the builtin `Bool` type:

```
let g = OurTrue
if g: print("It's true!")
```

It's true!

Avoiding type conversion with `__mlir_i1__`

Our `OurBool` type is looking great, and by providing a conversion to `Bool`, it can be used anywhere the builtin `Bool` type can. But in the last section we promised you "full control," the ability to define your own version of any type built into Mojo or its standard library. Surely `Bool` doesn't implement `__bool__` to convert itself into `Bool`?

Indeed it doesn't: when Mojo evaluates a conditional expression, it actually attempts to convert it to an MLIR `i1` value, by searching for the special interface method `__mlir_i1__`. (The automatic conversion to `Bool` occurs because `Bool` is known to implement the `__mlir_i1__` method.)

Again, Mojo is designed to be extensible and modular. By implementing all the special methods `Bool` does, we can create a type that can replace it entirely. Let's do so by implementing `_mlir_i1_` on `OurBool`:

```
alias OurTrue = OurBool(_mlir_attr.`true`)
alias OurFalse: OurBool = _mlir_attr.`false`  
  
@register_passable("trivial")
struct OurBool:
    var value: _mlir_type.i1  
  
    fn __init__(value: _mlir_type.i1) -> Self:
        return Self {value: value}  
  
    # ...  
  
    # Our new method converts `OurBool` to `i1`:
    fn __mlir_i1__(self) -> _mlir_type.i1:
        return self.value
```

We can still use `OurBool` in conditionals just as we did before:

```
let h = OurTrue
if h: print("No more Bool conversion!")
```

No more Bool conversion!

But this time, no conversion to `Bool` occurs. You can try adding `print` statements to the `_bool_` and `_mlir_i1_` methods, or even removing the `_bool_` method entirely, to see for yourself.

Adding functionality with MLIR

There are many more ways we can improve `OurBool`. Many of those involve implementing special methods, some of which you may recognize from Python, and some which are specific to Mojo. For example, we can implement inversion of a `OurBool` value by adding a `_invert_` method. We can also add an `_eq_` method, which allows two `OurBool` to be compared with the `==` operator.

What sets Mojo apart is the fact that we can implement each of these using MLIR. To implement `_eq_`, for example, we use the `index.casts` operation to cast our `i1` values to the MLIR index dialect's `index` type, and then the `index.cmp` operation to compare them for equality. And with the `_eq_` method implemented, we can then implement `_invert_` in terms of `_eq_`:

```
alias OurTrue = OurBool(_mlir_attr.`true`)
alias OurFalse: OurBool = _mlir_attr.`false`  
  
@register_passable("trivial")
struct OurBool:
    var value: _mlir_type.i1  
  
    fn __init__(value: _mlir_type.i1) -> Self:
        return Self {value: value}  
  
    # ...  
  
    fn __mlir_i1__(self) -> _mlir_type.i1:
        return self.value  
  
    fn __eq__(self, rhs: OurBool) -> Self:
        let lhsIndex = _mlir_op.`index.casts`[_type=_mlir_type.index]{
            self.value
        }
        let rhsIndex = _mlir_op.`index.casts`[_type=_mlir_type.index]{
            rhs.value
        }
        return Self{
            _mlir_op.`index.cmp`[
                pred=_mlir_attr.`#index<cmp_predicate_eq>`
```

```
](lhsIndex, rhsIndex)
)

fn __invert__(self) -> Self:
    return OurFalse if self == OurTrue else OurTrue
```

This allows us to use the `~` operator with `OurBool`:

```
let i = OurFalse
if ~i: print("It's false!")
```

It's false!

This extensible design is what allows even “built in” Mojo types like `Bool`, `Int`, and even `Tuple (!!)` to be implemented in the Mojo standard library in terms of MLIR, rather than hard-coded into the Mojo language. This also means that there’s almost nothing that those types can achieve that user-defined types cannot.

By extension, this means that the incredible performance that Mojo unlocks for machine learning workflows isn’t due to some magic being performed behind a curtain – you can define your own high-level types that, in their implementation, use low-level MLIR to achieve unprecedented speed and control.

The promise of modularity

As we’ve seen, Mojo’s integration with MLIR allows Mojo programmers to implement zero-cost abstractions on par with Mojo’s own builtin and standard library types.

MLIR is open-source and extensible: new dialects are being added all the time, and those dialects then become available to use in Mojo. All the while, Mojo code gets more powerful and more optimized for new hardware – with no additional work necessary by Mojo programmers.

What this means is that your own custom types, whether those be `OurBool` or `OurTensor`, can be used to provide programmers with an easy-to-use and unchanging interface. But behind the scenes, MLIR will optimize those convenient, high-level types for the computing environments of tomorrow.

In other words: Mojo isn’t magic, it’s modular.

Mandelbrot in Mojo with Python plots

Learn how to write high-performance Mojo code and import Python packages.

Not only is Mojo great for writing high-performance code, but it also allows us to leverage the huge Python ecosystem of libraries and tools. With seamless Python interoperability, Mojo can use Python for what it's good at, especially GUIs, without sacrificing performance in critical code. Let's take the classic Mandelbrot set algorithm and implement it in Mojo.

This tutorial shows two aspects of Mojo. First, it shows that Mojo can be used to develop fast programs for irregular applications. It also shows how we can leverage Python for visualizing the results.

► Code

First set some parameters, you can try changing these to see different results:

```
alias width = 960
alias height = 960
alias MAX_ITERS = 200

alias min_x = -2.0
alias max_x = 0.6
alias min_y = -1.5
alias max_y = 1.5
```

The core [Mandelbrot](#) algorithm involves computing an iterative complex function for each pixel until it "escapes" the complex circle of radius 2, counting the number of iterations to escape:

$$z_{i+1} = z_i^2 + c$$

```
# Compute the number of steps to escape.
def mandelbrot_kernel(c: ComplexFloat64) -> Int:
    z = c
    for i in range(MAX_ITERS):
        z = z * z + c
        if z.squared_norm() > 4:
            return i
    return MAX_ITERS

def compute_mandelbrot() -> Tensor[float_type]:
    # create a matrix. Each element of the matrix corresponds to a pixel
    t = Tensor[float_type](height, width)

    dx = (max_x - min_x) / width
    dy = (max_y - min_y) / height

    y = min_y
    for row in range(height):
        x = min_x
        for col in range(width):
            t[Index(row, col)] = mandelbrot_kernel(ComplexFloat64(x, y))
            x += dx
        y += dy
    return t
```

Plotting the number of iterations to escape with some color gives us the canonical Mandelbrot set plot. To render it we can directly leverage Python's `matplotlib` right from Mojo!

First install the required libraries:

```
%%python
from importlib.util import find_spec
import sys
import subprocess
```

```
if not find_spec("matplotlib"):
    print("Matplotlib not found, installing...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "matplotlib"])

if not find_spec("numpy"):
    print("Numpy not found, installing...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "numpy"])

def show_plot(tensor: Tensor[float_type]):
    alias scale = 10
    alias dpi = 64

    np = Python.import_module("numpy")
    plt = Python.import_module("matplotlib.pyplot")
    colors = Python.import_module("matplotlib.colors")

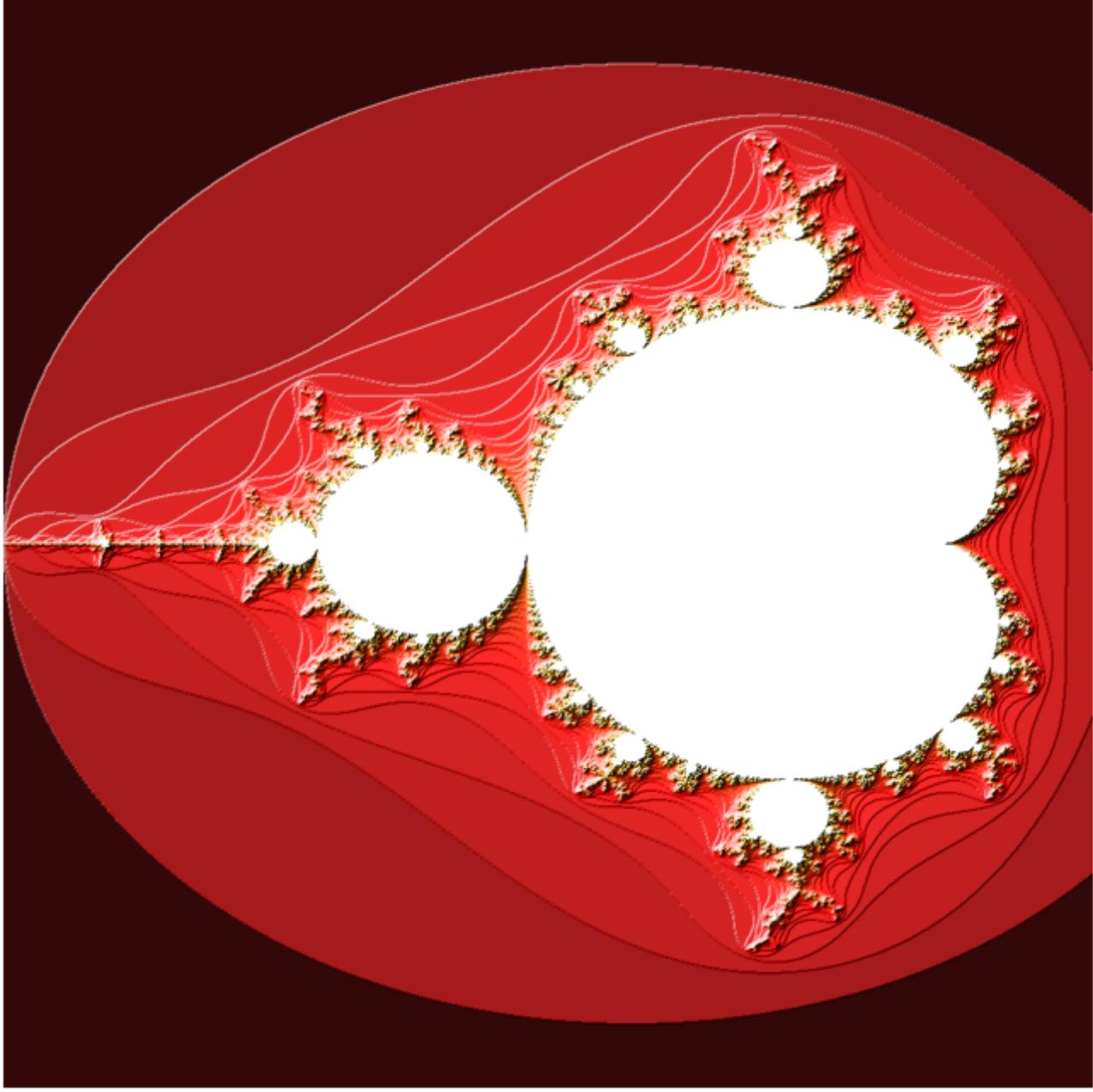
    numpy_array = np.zeros((height, width), np.float64)

    for row in range(height):
        for col in range(width):
            numpy_array.itemset((col, row), tensor[col, row])

    fig = plt.figure(1, [scale, scale * height // width], dpi)
    ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], False, 1)
    light = colors.LightSource(315, 10, 0, 1, 1, 0)

    image = light.shade(numpy_array, plt.cm.hot, colors.PowerNorm(0.3), "hsv", 0, 0, 1.5)
    plt.imshow(image)
    plt.axis("off")
    plt.show()

show_plot(compute_mandelbrot())
```



Vectorizing Mandelbrot

We showed a naive implementation of the Mandelbrot algorithm, but there are two things we can do to speed it up. We can early-stop the loop iteration when a pixel is known to have escaped, and we can leverage Mojo's access to hardware by vectorizing the loop, computing multiple pixels simultaneously. To do that we will use the `vectorize` higher order generator.

We start by defining our main iteration loop in a vectorized fashion

```
fn mandelbrot_kernel SIMD[
    simd_width: Int
](c: ComplexSIMD[float_type, simd_width]) -> SIMD[float_type, simd_width]:
    """A vectorized implementation of the inner mandelbrot computation."""
    let cx = c.re
    let cy = c.im
    var x = SIMD[float_type, simd_width](0)
    var y = SIMD[float_type, simd_width](0)
    var y2 = SIMD[float_type, simd_width](0)
    var iters = SIMD[float_type, simd_width](0)

    var t: SIMD[DTType.bool, simd_width] = True
    for i in range(MAX_ITERS):
        if not t.reduce_or():
            break
        y2 = y*y
        y = x.fma(y + y, cy)
        t = x.fma(x, y2) <= 4
```

```

x = x.fma(x, cx - y2)
iters = t.select(iters + 1, iters)
return iters

```

The above function is parameterized on the `simd_width` and processes `simd_width` pixels. It only escapes once all pixels within the vector lane are done. We can use the same iteration loop as above, but this time we vectorize within each row instead. We use the `vectorize` generator to make this a simple function call.

```

fn vectorized():
    let t = Tensor[float_type](height, width)

    @parameter
    fn worker(row: Int):
        let scale_x = (max_x - min_x) / width
        let scale_y = (max_y - min_y) / height

    @parameter
    fn compute_vector[simd_width: Int](col: Int):
        """Each time we operate on a `simd_width` vector of pixels."""
        let cx = min_x + (col + iota[float_type, simd_width]()) * scale_x
        let cy = min_y + row * scale_y
        let c = ComplexSIMD[float_type, simd_width](cx, cy)
        t.data().simd_store[simd_width](row * width + col, mandelbrot_kernel SIMD[simd_width](c))

    # Vectorize the call to compute_vector where call gets a chunk of pixels.
    vectorize[simd_width, compute_vector](width)

@parameter
fn bench[simd_width: Int]():
    for row in range(height):
        worker(row)

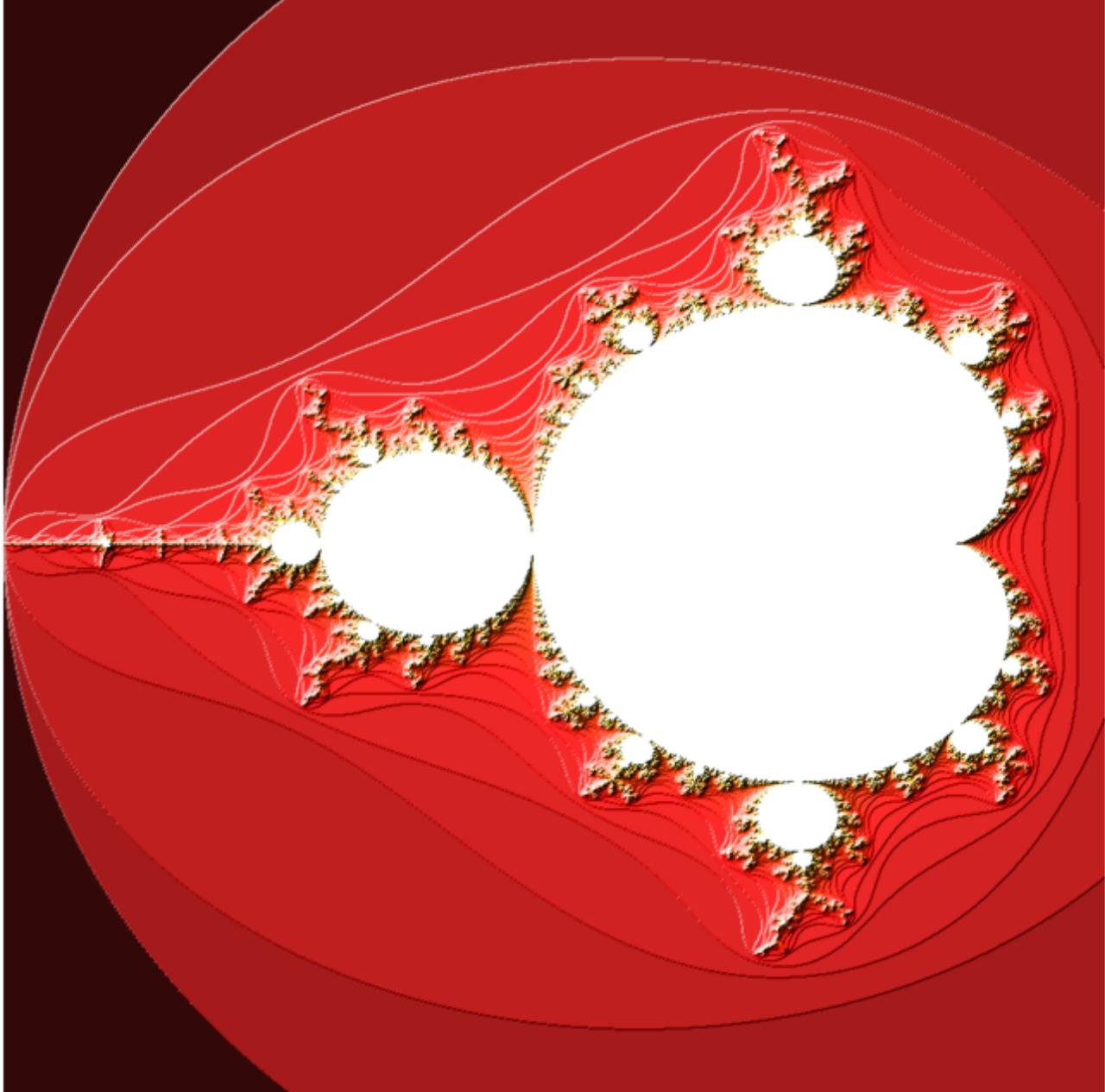
let vectorized = Benchmark().run[bench[simd_width]]() / 1e6
print("Vectorized", ":", vectorized, "ms")

try:
    _ = show_plot(t)
except e:
    print("failed to show plot:", e)

vectorized()

```

Vectorized : 12.177345000000001 ms



Parallelizing Mandelbrot

While the vectorized implementation above is efficient, we can get better performance by parallelizing on the cols. This again is simple in Mojo using the `parallelize` higher order function. Only the function that performs the invocation needs to change.

```
fn parallelized():
    let t = Tensor[float_type](height, width)

    @parameter
    fn worker(row: Int):
        let scale_x = (max_x - min_x) / width
        let scale_y = (max_y - min_y) / height

    @parameter
    fn compute_vector[simd_width: Int](col: Int):
        """Each time we operate on a `simd_width` vector of pixels."""
        let cx = min_x + (col + iota[float_type, simd_width]()) * scale_x
        let cy = min_y + row * scale_y
        let c = ComplexSIMD[float_type, simd_width](cx, cy)
        t.data().simd_store[simd_width](row * width + col, mandelbrot_kernel SIMD[simd_width](c))

    # Vectorize the call to compute_vector where call gets a chunk of pixels.
    vectorize[simd_width, compute_vector](width)

    @parameter
    fn bench_parallel[simd_width: Int]():
```

```

parallelize[worker](height, height)

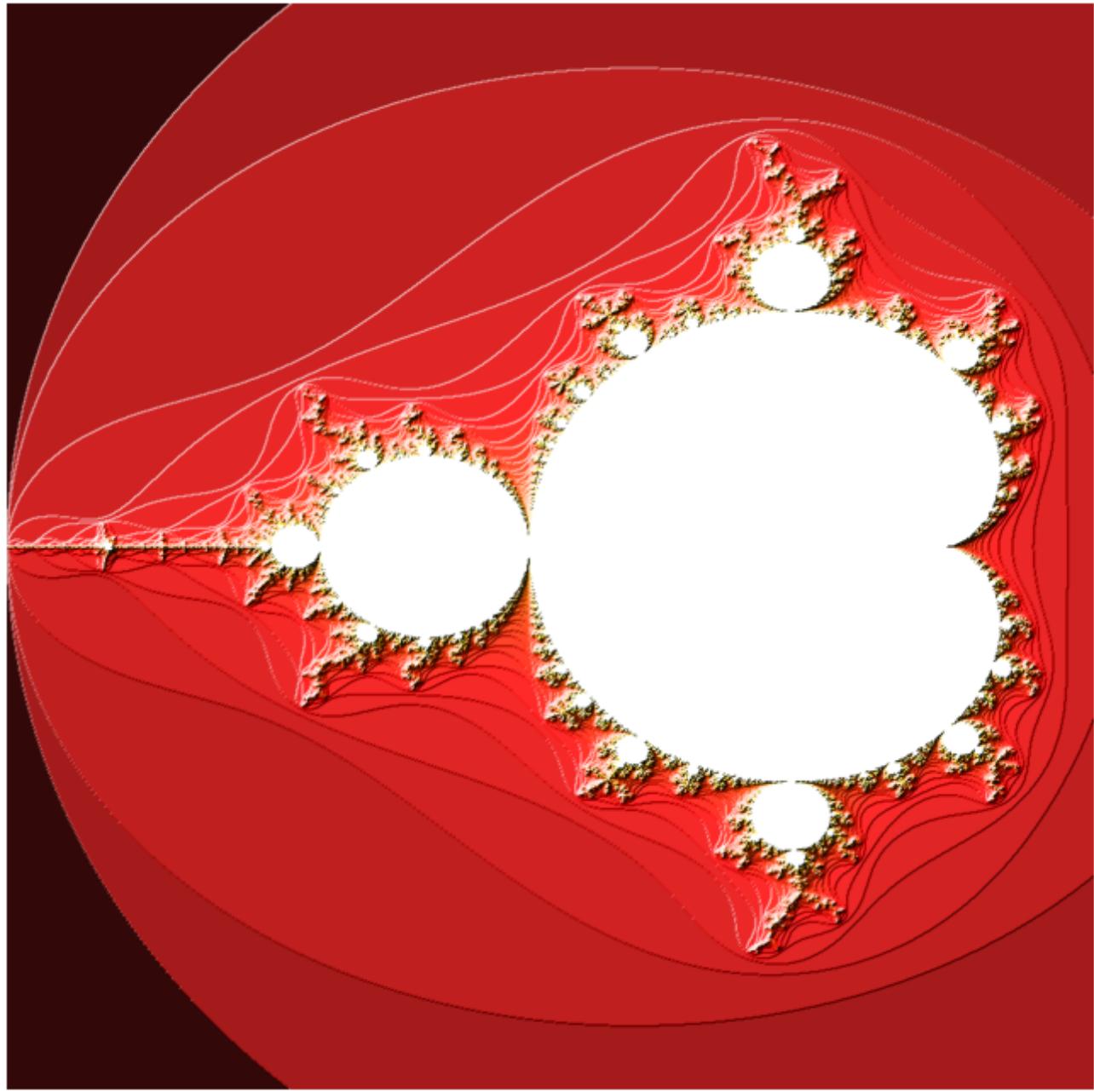
let parallelized = Benchmark().run[bench_parallel[simd_width]]() / 1e6
print("Parallelized:", parallelized, "ms")

try:
    _ = show_plot(t)
except e:
    print("failed to show plot:", e)

parallelized()_

```

Parallelized: 1.4245639999999999 ms



Benchmarking

In this section we increase the size to 4096x4096 and run 1000 iterations for a larger test to stress the CPU

```

fn compare():
    let t = Tensor[float_type](height, width)

    @parameter
    fn worker(row: Int):
        let scale_x = (max_x - min_x) / width
        let scale_y = (max_y - min_y) / height

    @parameter

```

```

fn compute_vector[simd_width: Int](col: Int):
    """Each time we operate on a `simd_width` vector of pixels."""
    let cx = min_x + (col + iota[float_type, simd_width]()) * scale_x
    let cy = min_y + row * scale_y
    let c = ComplexSIMD[float_type, simd_width](cx, cy)
    t.data().simd_store[simd_width](row * width + col, mandelbrot_kernel SIMD[simd_width](c))

# Vectorize the call to compute_vector where call gets a chunk of pixels.
vectorize[simd_width, compute_vector](width)

@parameter
fn bench[simd_width: Int]():
    for row in range(height):
        worker(row)

let vectorized = Benchmark().run[bench[simd_width]]() / 1e6
print("Number of threads:", num_cores())
print("Vectorized:", vectorized, "ms")

# Parallelized
@parameter
fn bench_parallel[simd_width: Int]():
    parallelize[worker](height, height)

let parallelized = Benchmark().run[bench_parallel[simd_width]]() / 1e6
print("Parallelized:", parallelized, "ms")
print("Parallel speedup:", vectorized / parallelized)

_ = t # Make sure tensor isn't destroyed before benchmark is finished

compare()

```

Number of threads: 16
 Vectorized: 12.171849 ms
 Parallelized: 1.3043979999999999 ms
 Parallel speedup: 9.3313919524562294

Matrix multiplication in Mojo

Learn how to leverage Mojo's various functions to write a high-performance matmul.

This notebook describes how to write a matrix multiplication (matmul) algorithm in Mojo. We will start with a pure Python implementation, transition to a naive implementation that is essentially a copy of the Python one, then add types, then continue the optimizations by vectorizing, tiling, and parallelizing the implementation.

First, let's define matrix multiplication. Given two dense matrices A and B of dimensions $M \times K$ and $K \times N$ respectively, we want to compute their dot product $C = A \cdot B$ (also known as matmul). The dot product $C + = A \cdot B$ is defined by

$$C_{i,j} += \sum_{k \in [0 \dots K]} A_{i,k} B_{k,j}$$

Please take look at our [blog](#) post on matmul and why it is important for ML and DL workloads.

The format of this notebook is to start with an implementation which is identical to that of Python (effectively renaming the file extension), then look at how adding types to the implementation helps performance before extending the implementation by leveraging the vectorization and parallelization capabilities available on modern hardware. Throughout the execution, we report the GFlops achieved.

Python Implementation

Let's first implement matmul in Python directly from the definition.

```
%%python
def matmul_python(C, A, B):
    for m in range(C.rows):
        for k in range(A.cols):
            for n in range(C.cols):
                C[m, n] += A[m, k] * B[k, n]
```

Let's benchmark our implementation using 128 by 128 square matrices and report the achieved GFlops.

Install numpy if it's not already:

```
%%python
from importlib.util import find_spec
import sys
import subprocess

if not find_spec("numpy"):
    print("Numpy not found, installing...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "numpy"])

%%python
from timeit import timeit
import numpy as np

class Matrix:
    def __init__(self, value, rows, cols):
        self.value = value
        self.rows = rows
        self.cols = cols

    def __getitem__(self, idxs):
        return self.value[idxs[0]][idxs[1]]

    def __setitem__(self, idxs, value):
        self.value[idxs[0]][idxs[1]] = value

def benchmark_matmul_python(M, N, K):
    A = Matrix(list(np.random.rand(M, K)), M, K)
    B = Matrix(list(np.random.rand(K, N)), K, N)
    C = Matrix(list(np.zeros((M, N))), M, N)
    secs = timeit(lambda: matmul_python(C, A, B), number=2)/2
    gflops = ((2*M*N*K)/secs) / 1e9
    print(gflops, "GFLOP/s")
    return gflops

python_gflops = benchmark_matmul_python(128, 128, 128).to_float64()
0.0018947946413032505 GFLOP/s
```

Importing the Python implementation to Mojo

Using Mojo is as simple as Python. First, let's include that modules from the Mojo stdlib that we are going to use:

► Import utilities and define Matrix (click to show/hide)

Then, we can copy and paste our Python code. Mojo is a superset of Python, so the same Python code will run as Mojo code

```
# This exactly the same Python implementation,  
# but is infact Mojo code!  
def matmul_untyped(C, A, B):  
    for m in range(C.rows):  
        for k in range(A.cols):  
            for n in range(C.cols):  
                C[m, n] += A[m, k] * B[k, n]
```

We can then benchmark the implementation. As before we use a 128 by 128 matrix

```
fn matrix_getitem(self: object, i: object) raises -> object:  
    return self.value[i]  
  
fn matrix_setitem(self: object, i: object, value: object) raises -> object:  
    self.value[i] = value  
    return None  
  
fn matrix_append(self: object, value: object) raises -> object:  
    self.value.append(value)  
    return None  
  
fn matrix_init(rows: Int, cols: Int) raises -> object:  
    let value = object([])  
    return object(  
        Attr("value", value), Attr("__getitem__", matrix_getitem), Attr("__setitem__", matrix_setitem),  
        Attr("rows", rows), Attr("cols", cols), Attr("append", matrix_append),  
    )  
  
def benchmark_matmul_untyped(M: Int, N: Int, K: Int, python_gflops: Float64):  
    C = matrix_init(M, N)  
    A = matrix_init(M, K)  
    B = matrix_init(K, N)  
    for i in range(M):  
        c_row = object([])  
        b_row = object([])  
        a_row = object([])  
        for j in range(N):  
            c_row.append(0.0)  
            b_row.append(random_float64(-5, 5))  
            a_row.append(random_float64(-5, 5))  
        C.append(c_row)  
        B.append(b_row)  
        A.append(a_row)  
  
    @parameter  
    fn test_fn():  
        try:  
            _ = matmul_untyped(C, A, B)  
        except:  
            pass  
  
    let secs = Float64(Benchmark().run[test_fn]() / 1_000_000_000  
    = (A, B, C)  
    let gflops = ((2*M*N*K)/secs) / 1e9  
    let speedup : Float64 = gflops / python_gflops  
    print(gflops, "GFLOP/s, a", speedup.value, "x speedup over Python")  
  
benchmark_matmul_untyped(128, 128, 128, python_gflops)  
0.0092210218211624933 GFLOP/s, a 4.866501952327785 x speedup over Python
```

Note the huge speedup with no effort that we have gotten.

Adding types to the Python implementation

The above program, while achieving better performance than Python, is still not the best we can get from Mojo. If we tell Mojo the types of the inputs, it can optimize much of the code away and reduce dispatching costs (unlike Python, which only uses types for type checking, Mojo exploits type info for performance optimizations as well).

To do that, let's first define a `Matrix` struct. The `Matrix` struct contains a data pointer along with size fields. While the `Matrix` struct can be parametrized on any data type, here we set the data type to be `Float32` for conciseness.

```
alias type = DType.float32  
  
struct Matrix:  
    var data: DTypePointer[type]  
    var rows: Int  
    var cols: Int  
  
    # Initialize zeroeing all values  
    fn __init__(inout self, rows: Int, cols: Int):  
        self.data = DTypePointer[type].alloc(rows * cols)
```

```

memset_zero(self.data, rows * cols)
self.rows = rows
self.cols = cols

# Initialize taking a pointer, don't set any elements
fn __init__(inout self, rows: Int, cols: Int, data: DTypePointer[DType.float32]):
    self.data = data
    self.rows = rows
    self.cols = cols

## Initialize with random values
@staticmethod
fn rand(rows: Int, cols: Int) -> Self:
    let data = DTypePointer[type].alloc(rows * cols)
    rand(data, rows * cols)
    return Self(rows, cols, data)

fn __getitem__(self, y: Int, x: Int) -> Float32:
    return self.load[1](y, x)

fn __setitem__(self, y: Int, x: Int, val: Float32):
    return self.store[1](y, x, val)

fn load[nelts: Int](self, y: Int, x: Int) -> SIMD[DType.float32, nelts]:
    return self.datasimd_load[nelts](y * self.cols + x)

fn store[nelts: Int](self, y: Int, x: Int, val: SIMD[DType.float32, nelts]):
    return self.datasimd_store[nelts](y * self.cols + x, val)

```

Note that we implement `getitem` and `setitem` in terms of `load` and `store`. For the naive implementation of `matmul` it does not make a difference, but we will utilize this later in a more optimized vectorized version of `matmul`.

With the above `Matrix` type we can effectively copy and paste the Python implementation and just add type annotations:

```

# Note that C, A, and B have types.
fn matmul_naive(C: Matrix, A: Matrix, B: Matrix):
    for m in range(C.rows):
        for k in range(A.cols):
            for n in range(C.cols):
                C[m, n] += A[m, k] * B[k, n]

```

We are going to benchmark the implementations as we improve, so let's write a helper function that will do that for us:

```

alias M = 1024
alias N = 1024
alias K = 1024

@always_inline
fn benchmark[
    func: fn (Matrix, Matrix, Matrix) -> None](base_gflops: Float64):
    var C = Matrix(M, N)
    var A = Matrix.rand(M, K)
    var B = Matrix.rand(K, N)

    @always_inline
    @parameter
    fn test_fn():
        _ = func(C, A, B)

    let secs = Float64(Benchmark().run[test_fn]() / 1_000_000_000
    # Prevent the matrices from being freed before the benchmark run
    A.data.free()
    B.data.free()
    C.data.free()
    let gflops = ((2 * M * N * K) / secs) / 1e9
    let speedup: Float64 = gflops / base_gflops
    # print(gflops, "GFLOP/s", speedup, " speedup")
    print(gflops, "GFLOP/s, a", speedup.value, "x speedup over Python")

```

Benchmarking shows significant speedups. We increase the size of the matrix to 512 by 512, since Mojo is much faster than Python.

```

benchmark[matmul_naive](python_gflops)
3.6469022982798753 GFLOP/s, a 1924.695277674796 x speedup over Python

```

Adding type annotations gives a huge improvement compared to the original untyped version.

Vectorizing the inner most loop

We can do better than the above implementation by utilizing the vector instructions. Rather than assuming a vector width, we query the SIMD width of the specified `DType` using `simd_width`. This makes our code portable as we transition to other hardware. Leverage SIMD instructions is as easy as:

```

# Mojo has SIMD vector types, we can vectorize the Matmul code as follows.
# nelts = number of float32 elements that can fit in SIMD register
alias nelts = simdwidthof[DType.float32]()
fn matmul_vectorized_0(C: Matrix, A: Matrix, B: Matrix):

```

```

for m in range(C.rows):
    for k in range(A.cols):
        for nv in range(0, C.cols, nelts):
            C.store[nelts](m,nv, C.load[nelts](m,nv) + A[m,k] * B.load[nelts](k,nv))

    # Handle remaining elements with scalars.
    for n in range(nelts*(C.cols//nelts), C.cols):
        C[m,n] += A[m,k] * B[k,n]

```

We can benchmark the above implementation. Note that many compilers can detect naive loops and perform optimizations on them. Mojo, however, allows you to be explicit and precisely control what optimizations are applied.

```
benchmark[matmul_vectorized_0](python_gflops)
12.535805674163893 GFLOP/s, a 6615.9178419154141 x speedup over Python
```

Vectorization is a common optimization, and Mojo provides a higher-order function that performs vectorization for you. The `vectorize` function takes a vector width and a function which is parameteric on the vector width and is going to be evaluated in a vectorized manner.

```

# Simplify the code by using the builtin vectorize function
from algorithm import vectorize
fn matmul_vectorized_1(C: Matrix, A: Matrix, B: Matrix):
    for m in range(C.rows):
        for k in range(A.cols):
            @parameter
            fn dot[nelts : Int](n : Int):
                C.store[nelts](m,n, C.load[nelts](m,n) + A[m,k] * B.load[nelts](k,n))
            vectorize[nelts, dot](C.cols)

```

There is only a slight difference in terms of performance between the two implementations:

```
benchmark[matmul_vectorized_1](python_gflops)
12.541864071124047 GFLOP/s, a 6619.1152316631433 x speedup over Python
```

Parallelizing Matmul

With Mojo we can easily run code in parallel with the `parallelize` function.

Let's modify our matmul implementation and make it multi-threaded (for simplicity, we only parallelize on the M dimension).

In `parallelize` below we're overpartitioning by distributing the work more evenly among processors. This ensures they all have something to work on even if some tasks finish before others, or some processors are stragglers. Intel and Apple now have separate performance and efficiency cores and this mitigates the problems that can cause.

```

# Parallelize the code by using the builtin parallelize function
from algorithm import parallelize
fn matmul_parallelized(C: Matrix, A: Matrix, B: Matrix):
    @parameter
    fn calc_row(m: Int):
        for k in range(A.cols):
            @parameter
            fn dot[nelts : Int](n : Int):
                C.store[nelts](m,n, C.load[nelts](m,n) + A[m,k] * B.load[nelts](k,n))
            vectorize[nelts, dot](C.cols)
        parallelize[calc_row](C.rows, C.rows)

```

We can benchmark the parallel matmul implementation.

```
benchmark[matmul_parallelized](python_gflops)
107.98687292023116 GFLOP/s, a 56991.333290850547 x speedup over Python
```

Tiling Matmul

Tiling is an optimization performed for matmul to increase cache locality. The idea is to keep sub-matrices resident in the cache and increase the reuse. The `tile` function itself can be written in Mojo as:

```

from algorithm import Static2DTileUnitFunc as Tile2DFunc

# Perform 2D tiling on the iteration space defined by end_x and end_y.
fn tile[tiled_fn: Tile2DFunc, tile_x: Int, tile_y: Int](end_x: Int, end_y: Int):
    # Note: this assumes that ends are multiples of the tiles.
    for y in range(0, end_y, tile_y):
        for x in range(0, end_x, tile_x):
            tiled_fn[tile_x, tile_y](x, y)

```

The above will perform 2 dimensional tiling over a 2D iteration space defined to be between $([0, end_x], [0, end_y])$. Once we define it above, we can use it within our matmul kernel. For simplicity we choose 4 as the tile height and since we also want to vectorize we use $4 * nelts$ as the tile width (since we vectorize on the columns).

```
# Use the above tile function to perform tiled matmul.
fn matmul_tiled_parallelized(C: Matrix, A: Matrix, B: Matrix):
    @parameter
    fn calc_row(m: Int):
        @parameter
        fn calc_tile[tile_x: Int, tile_y: Int](x: Int, y: Int):
            for k in range(y, y + tile_y):
                @parameter
                fn dot[nelts : Int](n : Int):
                    C.store[nelts](m, n+x, C.load[nelts](m, n+x) + A[m, k] * B.load[nelts](k, n+x))
                    vectorize[nelts, dot](tile_x)

            # We hardcode the tile factor to be 4.
            alias tile_size = 4
            tile[calc_tile, nelts * tile_size, tile_size](A.cols, C.cols)

    parallelize[calc_row](C.rows, C.rows)
```

Again, we can benchmark the tiled parallel matmul implementation:

```
benchmark[matmul_tiled_parallelized](python_gflops)
136.92271321866085 GFLOP/s, a 72262.560930869338 x speedup over Python
```

One source of overhead in the above implementation is the fact that we are not unrolling the loops introduced by vectorize of the dot function. We can do that via the vectorize_unroll higher-order function in Mojo:

```
# Unroll the vectorized loop by a constant factor.
from algorithm import vectorize_unroll
fn matmul_tiled_unrolled_parallelized(C: Matrix, A: Matrix, B: Matrix):
    @parameter
    fn calc_row(m: Int):
        @parameter
        fn calc_tile[tile_x: Int, tile_y: Int](x: Int, y: Int):
            for k in range(y, y + tile_y):
                @parameter
                fn dot[nelts : Int](n : Int):
                    C.store[nelts](m, n+x, C.load[nelts](m, n+x) + A[m, k] * B.load[nelts](k, n+x))

            # Vectorize by nelts and unroll by tile_x/nelts
            # Here unroll factor is 4
            vectorize_unroll[nelts, tile_x//nelts, dot](tile_x)

        alias tile_size = 4
        tile[calc_tile, nelts*tile_size, tile_size](A.cols, C.cols)

    parallelize[calc_row](C.rows, C.rows)
```

Again, we can benchmark the new tiled parallel matmul implementation with unrolled and vectorized inner loop:

```
benchmark[matmul_tiled_unrolled_parallelized](python_gflops)
173.70727621843133 GFLOP/s, a 91676.043636557093 x speedup over Python
```

Searching for the tile_factor

```
from autotune import autotune, search
from time import now
from memory.unsafe import Pointer

alias matmul_fn_sig_type = fn(Matrix, Matrix, Matrix) -> None
```

The choice of the tile factor can greatly impact the performance of the full matmul, but the optimal tile factor is highly hardware-dependent, and is influenced by the cache configuration and other hard-to-model effects. We want to write portable code without having to know everything about the hardware, so we can ask Mojo to automatically select the best tile factor using autotuning.

```
# Autotune the tile size used in the matmul.
@adaptive
fn matmul_autotune_impl(C: Matrix, A: Matrix, B: Matrix, /):
    @parameter
    fn calc_row(m: Int):
        @parameter
        fn calc_tile[tile_x: Int, tile_y: Int](x: Int, y: Int):
            for k in range(y, y + tile_y):
                @parameter
                fn dot[nelts : Int](n : Int):
                    C.store[nelts](m, n+x, C.load[nelts](m, n+x) + A[m, k] * B.load[nelts](k, n+x))
                    vectorize_unroll[nelts, tile_x // nelts, dot](tile_x)

            # Instead of hardcoding to tile_size = 4, search for the fastest
            # tile size by evaluating this function as tile size varies.
            alias tile_size = autotune(1, 2, 4, 8, 16, 32)
            tile[calc_tile, nelts * tile_size, tile_size](A.cols, C.cols)

    parallelize[calc_row](C.rows, C.rows)
```

This will generate multiple candidates for the matmul function. To teach Mojo how to find the best tile factor, we provide an evaluator function Mojo can use to assess each candidate.

```
fn matmul_evaluator(funcs: Pointer[matmul_fn_sig_type], size: Int) -> Int:  
    print("matmul_evaluator, number of candidates: ", size)  
  
    let eval_begin: Int = now()  
  
    # This size is picked at random, in real code we could use a real size  
    # distribution here.  
    let M = 512  
    let N = 512  
    let K = 512  
    print("Optimizing for size:", M, "x", N, "x", K)  
  
    var best_idx: Int = -1  
    var best_time: Int = -1  
  
    alias eval_iterations = 10  
    alias eval_samples = 10  
  
    var C = Matrix(M, N)  
    var A = Matrix(M, K)  
    var B = Matrix(K, N)  
    let Cptr = Pointer[Matrix].address_of(C).address  
    let Aptr = Pointer[Matrix].address_of(A).address  
    let Bptr = Pointer[Matrix].address_of(B).address  
  
    # Find the function that's the fastest on the size we're optimizing for  
    for f_idx in range(size):  
        let func = funcs.load(f_idx)  
  
        @always_inline  
        @parameter  
        fn wrapper():  
            func(C, A, B)  
        let cur_time = Benchmark(1, 100_000, 500_000_000, 1000_000_000).run[wrapper]()  
  
        if best_idx < 0:  
            best_idx = f_idx  
            best_time = cur_time  
        if best_time > cur_time:  
            best_idx = f_idx  
            best_time = cur_time  
  
    let eval_end: Int = now()  
    # Prevent matrices from being destroyed before we finished benchmarking them.  
    A.data.free()  
    B.data.free()  
    C.data.free()  
    print("Time spent in matmul_evaluator, ms:", (eval_end - eval_begin) // 1000000)  
    print("Best candidate idx:", best_idx)  
    return best_idx
```

Finally, we need to define an entry function that would simply call the best candidate.

```
fn matmul_autotune(C: Matrix, A: Matrix, B: Matrix):  
    alias best_impl: matmul_fn_sig_type  
    search[  
        matmul_fn_sig_type,  
        VariadicList(matmul_autotune_impl.__adaptive_set),  
        matmul_evaluator -> best_impl  
    ]()  
    # Run the best candidate  
    return best_impl(C, A, B)
```

Let's benchmark our new implementation:

```
benchmark[matmul_autotune](python_gflops)  
  
matmul_evaluator, number of candidates: 6  
Optimizing for size: 512 x 512 x 512  
Time spent in matmul_evaluator, ms: 8098  
Best candidate idx: 5  
201.48323262066052 GFLOP/s, a 106335.12900483991 x speedup over Python
```

Tile and accumulate in registers

Perform 2D tiling on the iteration space defined by end_x and end_y, parallelizing over y.

```
fn tile_parallel[  
    tiled_fn: Tile2DFunc, tile_x: Int, tile_y: Int  
](end_x: Int, end_y: Int):  
    # Note: this assumes that ends are multiples of the tiles.  
    @parameter  
    fn row(yo: Int):  
        let y = tile_y * yo  
        for x in range(0, end_x, tile_x):  
            tiled_fn[tile_x, tile_y](x, y)
```

```
parallelize[row](end_y // tile_y, 512) □
```

Tile the output and accumulate in registers. This strategy means we can compute tile_i * tile_j values of output for only reading tile_i + tile_j input values.

```
from memory import stack_allocation

fn accumulate_registers(C: Matrix, A: Matrix, B: Matrix):
    @parameter
    fn calc_tile[tile_j: Int, tile_i: Int](jo: Int, io: Int):
        # Allocate the tile of accumulators on the stack.
        var accumulators = Matrix(tile_i, tile_j, stack_allocation[tile_i * tile_j, DType.float32]())

    for k in range(0, A.cols):
        @parameter
        fn calc_tile_row[i: Int]():
            @parameter
            fn calc_tile_cols[nelts: Int](j: Int):
                accumulators.store[nelts](i, j, accumulators.load[nelts](i, j) + A[io + i, k] * B.load[nelts](k, jo + j))

            vectorize_unroll[nelts, tile_j // nelts, calc_tile_cols](tile_j)

        unroll[tile_i, calc_tile_row]()

    # Copy the local tile to the output
    for i in range(tile_i):
        for j in range(tile_j):
            C[io + i, jo + j] = accumulators[i, j]

alias tile_i = 4
alias tile_j = nelts*4
tile_parallel[calc_tile, tile_j, tile_i](C.cols, C.rows) □
```

```
benchmark[accumulate_registers](python_gflops) □
```

584.69820167517651 GFLOP/s, a 308581.30423728545 x speedup over Python

Fast memset in Mojo

Learn how to use Mojo's autotuning to quickly write a memset function.

In this tutorial we will implement a memset version optimized for small sizes using Mojo's autotuning feature.

The idea behind the implementation is based on Nadav Rotem's work [\[1\]](#), and is also well-described in [\[2\]](#).

We briefly summarize the approach below.

High-level overview

For the best memset performance we want to use the widest possible register width for the memory access. For instance, if we want to store 19 bytes, we want to use vector width 16 and use two overlapping stores. To store 9 bytes, we would want to use two 8-byte stores.

However, before we get to actually doing stores, we need to perform size checks to make sure that we're in the right range. I.e. we want to use 8 bytes stores for sizes 8-16, 16 bytes stores for sizes 16-32, etc.

The order in which we do the size checks significantly affects performance and ideally we would like to run as few checks as possible for the sizes that occur most often. I.e. if most of the sizes we see are 16-32, then we want to first check if it's within that range before we check if it's in 8-16 or some other range.

This results in a number of different comparison "trees" that can be used to perform the size checks, and in this tutorial we use Mojo's autotuning to pick the most optimal one given the distribution of input data.

Implementation

We will start as we always start - with imports and type aliases.

```
from autotune import autotune_fork, search
from utils.list import VariadicList
from math import min, max
from memory.unsafe import DTypePointer, Pointer
from time import now
from memory import memset as stdlib_memset

alias ValueType = UInt8
alias BufferPtrType = DTypePointer[DType.uint8]

alias memset_fn_type = fn(BufferPtrType, ValueType, Int) -> None
```

Now let's add some auxiliary functions. We will use them to benchmark various memset implementations and visualize results.

```
fn measure_time(
    func: memset_fn_type, size: Int, ITERS: Int, SAMPLES: Int
) -> Int:
    alias alloc_size = 1024 * 1024
    let ptr = BufferPtrType.alloc(alloc_size)

    var best = -1
    for sample in range(SAMPLES):
        let tic = now()
        for iter in range(ITERS):
            # Offset pointer to shake up cache a bit
            let offset_ptr = ptr.offset((iter * 128) & 1024)
```

```

# Just in case compiler will try to outsmart us and avoid repeating
# memset, change the value we're filling with
let v = ValueType(iter&255)

# Actually call the memset function
func(offset_ptr, v.value, size)

let toc = now()
if best < 0 or toc - tic < best:
    best = toc - tic

ptr.free()
return best

alias MULT = 2_000

fn visualize_result(size: Int, result: Int):
    print_no_newline("Size: ")
    if size < 10:
        print_no_newline(" ")
    print_no_newline(size, " | ")
    for _ in range(result // MULT):
        print_no_newline("*")
    print()

fn benchmark(func: memset_fn_type, title: StringRef):
    print("\n====")
    print(title)
    print("-----\n")

    alias benchmark_iterations = 30 * MULT
    alias warmup_samples = 10
    alias benchmark_samples = 1000

    # Warmup
    for size in range(35):
        _ = measure_time(
            func, size, benchmark_iterations, warmup_samples
        )

    # Actual run
    for size in range(35):
        let result = measure_time(
            func, size, benchmark_iterations, benchmark_samples
        )
        visualize_result(size, result)_

```

Reproducing results from the paper

Let's implement a memset version from the paper in Mojo and compare it against the system memset.

```

@always_inline
fn overlapped_store[
    width: Int
](ptr: BufferPtrType, value: ValueType, count: Int):
    let v = SIMD.splat[DType.uint8, width](value)
    ptrsimd_store[width](v)
    ptrsimd_store[width](count - width, v)

fn memset_manual(ptr: BufferPtrType, value: ValueType, count: Int):
    if count < 32:
        if count < 5:
            if count == 0:
                return
            # 0 < count <= 4
            ptr.store(0, value)
            ptr.store(count - 1, value)
            if count <= 2:

```

```

        return
ptr.store(1, value)
ptr.store(count - 2, value)
return

if count <= 16:
    if count >= 8:
        # 8 <= count < 16
        overlapped_store[8](ptr, value, count)
        return
    # 4 < count < 8
    overlapped_store[4](ptr, value, count)
    return

# 16 <= count < 32
overlapped_store[16](ptr, value, count)
else:
    # 32 < count
    memset_system(ptr, value, count)

fn memset_system(ptr: BufferPtrType, value: ValueType, count: Int):
    stdlib_memset(ptr, value.value, count) →

```

Let's benchmark our version of `memset` vs the standard `memset`.

Note: We're optimizing `memset` for tiniest sizes and benchmarking that properly is tricky. The notebook environment makes it even harder, and while we tried our best to tune the notebook to demonstrate the performance difference, it is hard to guarantee that the results will be stable from run to run.

```

benchmark(memset_manual, "Manual memset")
benchmark(memset_system, "System memset") →

```

=====

Manual memset

Size:	0	*****
Size:	1	*****
Size:	2	*****
Size:	3	*****
Size:	4	*****
Size:	5	*****
Size:	6	*****
Size:	7	*****
Size:	8	*****
Size:	9	*****
Size:	10	*****
Size:	11	*****
Size:	12	*****
Size:	13	*****
Size:	14	*****
Size:	15	*****
Size:	16	*****
Size:	17	*****
Size:	18	*****
Size:	19	*****
Size:	20	*****
Size:	21	*****
Size:	22	*****
Size:	23	*****
Size:	24	*****
Size:	25	*****
Size:	26	*****
Size:	27	*****
Size:	28	*****
Size:	29	*****
Size:	30	*****
Size:	31	*****
Size:	32	*****
Size:	33	*****

```

Size: 34 | ****
=====
System memset
-----
Size: 0 | ****
Size: 1 | ****
Size: 2 | ****
Size: 3 | ****
Size: 4 | ****
Size: 5 | ****
Size: 6 | ****
Size: 7 | ****
Size: 8 | ****
Size: 9 | ****
Size: 10 | ****
Size: 11 | ****
Size: 12 | ****
Size: 13 | ****
Size: 14 | ****
Size: 15 | ****
Size: 16 | ****
Size: 17 | ****
Size: 18 | ****
Size: 19 | ****
Size: 20 | ****
Size: 21 | ****
Size: 22 | ****
Size: 23 | ****
Size: 24 | ****
Size: 25 | ****
Size: 26 | ****
Size: 27 | ****
Size: 28 | ****
Size: 29 | ****
Size: 30 | ****
Size: 31 | ****
Size: 32 | ****
Size: 33 | ****
Size: 34 | ****

```

Tweaking the implementation for different sizes

We can see that it's already much faster for small sizes. That version was specifically optimized for a certain input size distribution, e.g. we can see that sizes 8-16 and 0-4 work fastest.

But what if in **our use case** the distribution is different? Let's imagine that in our case the most common sizes are 16-32 - is this version the most optimal version we can use then? The answer is obviously "no", and we can easily tweak the implementation to work better for these sizes - we just need to move the corresponding check closer to the beginning of the function. E.g. like so:

```

fn memset_manual_2(ptr: BufferPtrType, value: ValueType, count: Int):
    if count < 32:
        if count >= 16:
            # 16 <= count < 32
            overlapped_store[16](ptr, value, count)
            return

        if count < 5:
            if count == 0:
                return
            # 0 < count <= 4
            ptr.store(0, value)
            ptr.store(count - 1, value)
            if count <= 2:
                return
            ptr.store(1, value)
            ptr.store(count - 2, value)
            return

    if count >= 8:

```

```
# 8 <= count < 16
overlapped_store[8](ptr, value, count)
return
# 4 < count < 8
overlapped_store[4](ptr, value, count)

e:
# 32 < count
memset_system(ptr, value, count) underline
```

Let's check the performance of this version

```
benchmark(memset_manual_2, "Manual memset v2")
benchmark(memset_system, "Mojo system memset")
```

=====
=====

Size: 0 | *****
Size: 1 | *****
Size: 2 | *****
Size: 3 | *****
Size: 4 | *****
Size: 5 | *****
Size: 6 | *****
Size: 7 | *****
Size: 8 | *****
Size: 9 | *****
Size: 10 | *****
Size: 11 | *****
Size: 12 | *****
Size: 13 | *****
Size: 14 | *****
Size: 15 | *****
Size: 16 | *****
Size: 17 | *****
Size: 18 | *****
Size: 19 | *****
Size: 20 | *****
Size: 21 | *****
Size: 22 | *****
Size: 23 | *****
Size: 24 | *****
Size: 25 | *****
Size: 26 | *****
Size: 27 | *****
Size: 28 | *****
Size: 29 | *****
Size: 30 | *****
Size: 31 | *****
Size: 32 | *****
Size: 33 | *****
Size: 34 | *****

=====
=====

Size: 0	*****
Size: 1	*****
Size: 2	*****
Size: 3	*****
Size: 4	*****
Size: 5	*****
Size: 6	*****
Size: 7	*****
Size: 8	*****
Size: 9	*****
Size: 10	*****
Size: 11	*****
Size: 12	*****
Size: 13	*****

```

Size: 14 | ****
Size: 15 | ****
Size: 16 | ****
Size: 17 | ****
Size: 18 | ****
Size: 19 | ****
Size: 20 | ****
Size: 21 | ****
Size: 22 | ****
Size: 23 | ****
Size: 24 | ****
Size: 25 | ****
Size: 26 | ****
Size: 27 | ****
Size: 28 | ****
Size: 29 | ****
Size: 30 | ****
Size: 31 | ****
Size: 32 | ****
Size: 33 | ****
Size: 34 | ****

```

The performance is now much better on the 16-32 sizes!

The problem is that we had to manually re-write the code. Wouldn't it be nice if it was done automatically?

In Mojo this is possible (and quite easy) - we can generate multiple implementations and let the compiler pick the fastest one for us evaluating them on sizes we want!

Mojo implementation

Let's dive into that.

The first thing we need to do is to generate all possible candidates. To do that we will need to iteratively generate size checks to understand what size for the overlapping store we can use. Once we localize the size interval, we just call the overlapping store of the corresponding size.

To express this we will implement an adaptive function `memset_impl_layer` two parameters designating the current interval of possible size values. When we generate a new size check, we split that interval into two parts and recursively call the same functions on those two parts. Once we reach the minimal intervals, we will call the corresponding `overlapped_store` function.

This first implementation covers minimal interval cases:

```

@adaptive
@always_inline
fn memset_impl_layer[
    lower: Int, upper: Int
](ptr: BufferPtrType, value: ValueType, count: Int):
    @parameter
    if lower == -100 and upper == 0:
        pass
    elif lower == 0 and upper == 4:
        ptr.store(0, value)
        ptr.store(count - 1, value)
        if count <= 2:
            return
        ptr.store(1, value)
        ptr.store(count - 2, value)
    elif lower == 4 and upper == 8:
        overlapped_store[4](ptr, value, count)
    elif lower == 8 and upper == 16:
        overlapped_store[8](ptr, value, count)
    elif lower == 16 and upper == 32:
        overlapped_store[16](ptr, value, count)
    elif lower == 32 and upper == 100:
        memset_system(ptr, value, count)

```

```
else:  
    constrained[False]()
```

Let's now add an implementation for the other case, where we need to generate a size check.

```
@adaptive  
@always_inline  
fn memset_impl_layer[  
    lower: Int, upper: Int  
](ptr: BufferPtrType, value: ValueType, count: Int):  
    alias cur: Int  
    autotune_fork[Int, 0, 4, 8, 16, 32 -> cur]()  
  
    constrained[cur > lower]()  
    constrained[cur < upper]()  
  
    if count > cur:  
        memset_impl_layer[max(cur, lower), upper](ptr, value, count)  
    else:  
        memset_impl_layer[lower, min(cur, upper)](ptr, value, count)
```

Here we use autotune_fork to generate all possible at that point checks.

We will discard values beyond the current interval, and for the values within we will recursively call this function on the interval splits.

This is sufficient to generate multiple correct versions of memset, but to achieve the best performance we need to take into account one more factor: when we're dealing with such small sizes, even the code location matters a lot. E.g. if we swap Then and Else branches and invert the condition, we might get a different performance of the final function.

To account for that, let's add one more implementation of our function, but now with branches swapped:

```
@adaptive  
@always_inline  
fn memset_impl_layer[  
    lower: Int, upper: Int  
](ptr: BufferPtrType, value: ValueType, count: Int):  
    alias cur: Int  
    autotune_fork[Int, 0, 4, 8, 16, 32 -> cur]()  
  
    constrained[cur > lower]()  
    constrained[cur < upper]()  
  
    if count <= cur:  
        memset_impl_layer[lower, min(cur, upper)](ptr, value, count)  
    else:  
        memset_impl_layer[max(cur, lower), upper](ptr, value, count)
```

We defined building blocks for our implementation, now we need to add a top level entry-point that will kick off the recursion we've just defined.

We will simply call our function with [-100,100] interval - -100 and 100 simply designate that no checks have been performed yet. This interval will be refined as we generate more and more check until we have enough to emit actual stores.

```
@adaptive  
fn memset_autotune_impl(ptr: BufferPtrType, value: ValueType, count: Int, /):  
    memset_impl_layer[-100, 100](ptr, value, count)
```

Ok, we're done with our memset implementation, now we just need to plug it to autotuning infrastructure to let the Mojo compiler do the search and pick the best implementation.

To do that, we need to define an evaluator - this is a function that will take an array of function pointers to all implementations of our function and will need to return an index of the best candidate.

There are no limitations in how this function can be implemented - it can return the first or a random candidate, or it can actually benchmark all of them and pick the fastest - this is what we're going to do for this example.

```
fn memset_evaluator(funcs: Pointer[memset_fn_type], size: Int) -> Int:  
    # This size is picked at random, in real code we could use a real size  
    # distribution here.  
    let size_to_optimize_for = 17  
    print(size_to_optimize_for)  
  
    var best_idx: Int = -1  
    var best_time: Int = -1  
  
    alias eval_iterations = MULT  
    alias eval_samples = 500  
  
    # Find the function that's the fastest on the size we're optimizing for  
    for f_idx in range(size):  
        let func = funcs.load(f_idx)  
        let cur_time = measure_time(  
            func, size_to_optimize_for, eval_iterations, eval_samples  
        )  
        if best_idx < 0:  
            best_idx = f_idx  
            best_time = cur_time  
        if best_time > cur_time:  
            best_idx = f_idx  
            best_time = cur_time  
  
    return best_idx
```

The evaluator is ready, the last brush stroke is to add a function that will call the best candidate.

The search will be performed at compile time, and at runtime we will go directly to the best implementation.

```
fn memset_autotune(ptr: BufferPtrType, value: ValueType, count: Int):  
    # Get the set of all candidates  
    alias candidates = memset_autotune_impl.__adaptive_set  
  
    # Use the evaluator to select the best candidate.  
    alias best_impl: memset_fn_type  
    search[memset_fn_type, VariadicList(candidates), memset_evaluator -> best_impl]()  
  
    # Run the best candidate  
    return best_impl(ptr, value, count)
```

We are now ready to benchmark our function, let's see how its performance looks!

```
benchmark(memset_manual, "Mojo manual memset")  
benchmark(memset_manual_2, "Mojo manual memset v2")  
benchmark(memset_system, "Mojo system memset")  
benchmark(memset_autotune, "Mojo autotune memset")
```

=====

Mojo manual memset

Size: 0	*****
Size: 1	*****
Size: 2	*****
Size: 3	*****
Size: 4	*****
Size: 5	*****
Size: 6	*****
Size: 7	*****
Size: 8	*****
Size: 9	*****
Size: 10	*****
Size: 11	*****
Size: 12	*****

Size: 13	*****
Size: 14	*****
Size: 15	*****
Size: 16	*****
Size: 17	*****
Size: 18	*****
Size: 19	*****
Size: 20	*****
Size: 21	*****
Size: 22	*****
Size: 23	*****
Size: 24	*****
Size: 25	*****
Size: 26	*****
Size: 27	*****
Size: 28	*****
Size: 29	*****
Size: 30	*****
Size: 31	*****
Size: 32	*****
Size: 33	*****
Size: 34	*****

=====
Mojo manual memset v2

```
Size: 0 | ****
Size: 1 | ****
Size: 2 | ****
Size: 3 | ****
Size: 4 | ****
Size: 5 | ****
Size: 6 | ****
Size: 7 | ****
Size: 8 | ****
Size: 9 | ****
Size: 10 | ****
Size: 11 | ****
Size: 12 | ****
Size: 13 | ****
Size: 14 | ****
Size: 15 | ****
Size: 16 | ****
Size: 17 | ****
Size: 18 | ****
Size: 19 | ****
Size: 20 | ****
Size: 21 | ****
Size: 22 | ****
Size: 23 | ****
Size: 24 | ****
Size: 25 | ****
Size: 26 | ****
Size: 27 | ****
Size: 28 | ****
Size: 29 | ****
Size: 30 | ****
Size: 31 | ****
Size: 32 | ****
Size: 33 | ****
Size: 34 | ****
```

=====
Mojo system memset

Size: 0	*****
Size: 1	*****
Size: 2	*****
Size: 3	*****
Size: 4	*****
Size: 5	*****
Size: 6	*****

Size: 7 | ****
Size: 8 | *****
Size: 9 | *****
Size: 10 | *****
Size: 11 | *****
Size: 12 | *****
Size: 13 | *****
Size: 14 | *****
Size: 15 | *****
Size: 16 | *****
Size: 17 | *****
Size: 18 | *****
Size: 19 | *****
Size: 20 | *****
Size: 21 | *****
Size: 22 | *****
Size: 23 | *****
Size: 24 | *****
Size: 25 | *****
Size: 26 | *****
Size: 27 | *****
Size: 28 | *****
Size: 29 | *****
Size: 30 | *****
Size: 31 | *****
Size: 32 | *****
Size: 33 | *****
Size: 34 | *****

=====

Mojo autotune memset

Size: 0 | *****
Size: 1 | *****
Size: 2 | *****
Size: 3 | *****
Size: 4 | *****
Size: 5 | *****
Size: 6 | *****
Size: 7 | *****
Size: 8 | *****
Size: 9 | *****
Size: 10 | *****
Size: 11 | *****
Size: 12 | *****
Size: 13 | *****
Size: 14 | *****
Size: 15 | *****
Size: 16 | *****
Size: 17 | *****
Size: 18 | *****
Size: 19 | *****
Size: 20 | *****
Size: 21 | *****
Size: 22 | *****
Size: 23 | *****
Size: 24 | *****
Size: 25 | *****
Size: 26 | *****
Size: 27 | *****
Size: 28 | *****
Size: 29 | *****
Size: 30 | *****
Size: 31 | *****
Size: 32 | *****
Size: 33 | *****
Size: 34 | *****

Ray tracing in Mojo

Learn how to draw 3D graphics with ray-traced lighting using Mojo.

This tutorial about [ray tracing](#) is based on the popular tutorial [Understandable RayTracing in C++](#). The mathematical explanations are well described in that tutorial, so we'll just point you to the appropriate sections for reference as we implement a basic ray tracer in Mojo.

Step 1: Basic definitions

We'll start by defining a `Vec3f` struct, which will use to represent a vector in 3D space as well as RGB pixels. We'll use a SIMD representation for our vector to enable vectorized operations. Note that since the SIMD type only allows a power of 2, we always pad the underlying storage with a 0.

```
from math import rsqrt

@register_passable("trivial")
struct Vec3f:
    var data: SIMD[DType.float32, 4]

    @always_inline
    fn __init__(x: Float32, y: Float32, z: Float32) -> Self:
        return Vec3f {data: SIMD[DType.float32, 4](x, y, z, 0)}

    @always_inline
    fn __init__(data: SIMD[DType.float32, 4]) -> Self:
        return Vec3f {data: data}

    @always_inline
    @staticmethod
    fn zero() -> Vec3f:
        return Vec3f(0, 0, 0)

    @always_inline
    fn __sub__(self, other: Vec3f) -> Vec3f:
        return self.data - other.data

    @always_inline
    fn __add__(self, other: Vec3f) -> Vec3f:
        return self.data + other.data

    @always_inline
    fn __matmul__(self, other: Vec3f) -> Float32:
        return (self.data * other.data).reduce_add()

    @always_inline
    fn __mul__(self, k: Float32) -> Vec3f:
        return self.data * k

    @always_inline
    fn __neg__(self) -> Vec3f:
        return self.data * -1.0

    @always_inline
    fn __getitem__(self, idx: Int) -> SIMD[DType.float32, 1]:
        return self.data[idx]

    @always_inline
    fn cross(self, other: Vec3f) -> Vec3f:
        let self_zxy = self.data.shuffle[2, 0, 1, 3]()
        let other_zxy = other.data.shuffle[2, 0, 1, 3]()
        return (self_zxy * other.data - self.data * other_zxy).shuffle[
            2, 0, 1, 3
        ]()

    @always_inline
    fn normalize(self) -> Vec3f:
        return self.data * rsqrt(self @ self)
```

We now define our `Image` struct, which will store the RGB pixels of our images. It also contains a method to conver this Mojo struct into a numpy image, which will be used for implementing a straightforward displaying mechanism. We will also implement a function for loading PNG files from disk.

First install the required libraries:

```
%%python
from importlib.util import find_spec
import sys
import subprocess

if not find_spec("matplotlib"):
    print("Matplotlib not found, installing...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "matplotlib"])

if not find_spec("numpy"):
    print("Numpy not found, installing...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "numpy"])

from python import Python
from python.object import PythonObject

struct Image:
    # reference count used to make the object efficiently copyable
    var rc: Pointer[Int]
    # the two dimensional image is represented as a flat array
    var pixels: Pointer[Vec3f]
    var height: Int
    var width: Int

    fn __init__(inout self, height: Int, width: Int):
        self.height = height
        self.width = width
        self.pixels = Pointer[Vec3f].alloc(self.height * self.width)
        self.rc = Pointer[Int].alloc(1)
        self.rc.store(1)
```

```

fn __copyinit__(inout self, other: Self):
    other._inc_rc()
    self.pixels = other.pixels
    self.rc = other.rc
    self.height = other.height
    self.width = other.width

fn __del__(owned self):
    self._dec_rc()

fn _get_rc(self) -> Int:
    return self.rc.load()

fn _dec_rc(self):
    let rc = self._get_rc()
    if rc > 1:
        self.rc.store(rc - 1)
        return
    self._free()

fn _inc_rc(self):
    let rc = self._get_rc()
    self.rc.store(rc + 1)

fn _free(self):
    self.rc.free()
    self.pixels.free()

@always_inline
fn set(self, row: Int, col: Int, value: Vec3f) -> None:
    self.pixels.store(self._pos_to_index(row, col), value)

@always_inline
fn _pos_to_index(self, row: Int, col: Int) -> Int:
    # Convert a (row, col) position into an index in the underlying linear storage
    return row * self.width + col

def to_numpy_image(self) -> PythonObject:
    let np = Python.import_module("numpy")
    let plt = Python.import_module("matplotlib.pyplot")

    let np_image = np.zeros((self.height, self.width, 3), np.float32)

    # We use raw pointers to efficiently copy the pixels to the numpy array
    let out_pointer = Pointer(
        __mlir_op.pop.index_to_pointer`[
            _type=__mlir_type['!kgen.pointer<scalar<f32>>']
        ](
            SIMD[DType.index, 1](
                np_image.__array_interface__["data"][0].__index__()
            ).value
        )
    )
    let in_pointer = Pointer(
        __mlir_op.pop.index_to_pointer`[
            _type=__mlir_type['!kgen.pointer<scalar<f32>>']
        ](SIMD[DType.index, 1](self.pixels._as_index()).value)
    )

    for row in range(self.height):
        for col in range(self.width):
            let index = self._pos_to_index(row, col)
            for dim in range(3):
                out_pointer.store(
                    index * 3 + dim, in_pointer[index * 4 + dim]
                )

    return np_image

def load_image(fname: String) -> Image:
    let np = Python.import_module("numpy")
    let plt = Python.import_module("matplotlib.pyplot")

    let np_image = plt.imread(fname)
    let rows = np_image.shape[0].__index__()
    let cols = np_image.shape[1].__index__()
    let image = Image(rows, cols)

    let in_pointer = Pointer(
        __mlir_op.pop.index_to_pointer`[
            _type=__mlir_type['!kgen.pointer<scalar<f32>>']
        ](
            SIMD[DType.index, 1](
                np_image.__array_interface__["data"][0].__index__()
            ).value
        )
    )
    let out_pointer = Pointer(
        __mlir_op.pop.index_to_pointer`[
            _type=__mlir_type['!kgen.pointer<scalar<f32>>']
        ](SIMD[DType.index, 1](image.pixels._as_index()).value)
    )
    for row in range(rows):
        for col in range(cols):
            let index = image._pos_to_index(row, col)
            for dim in range(3):
                out_pointer.store(
                    index * 4 + dim, in_pointer[index * 3 + dim]
                )
    return image

```

We then add a function for quickly displaying an `Image` into the notebook. Our Python interop comes in quite handy.

```

def render(image: Image):
    np = Python.import_module("numpy")
    plt = Python.import_module("matplotlib.pyplot")
    colors = Python.import_module("matplotlib.colors")
    dpi = 32
    fig = plt.figure(1, [image.height // 10, image.width // 10], dpi)

    plt.imshow(image.to_numpy_image())

```

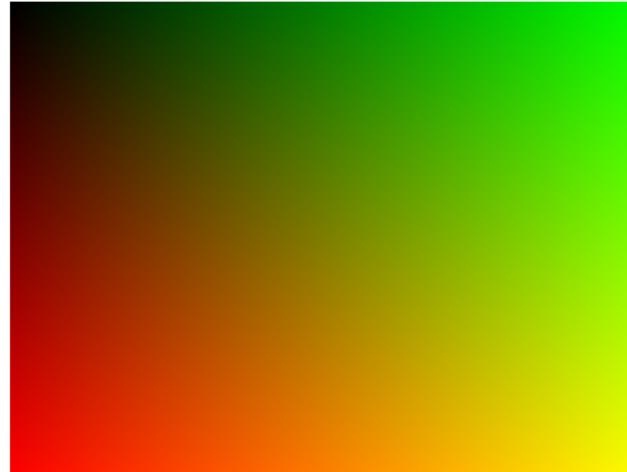
```
plt.axis("off")
plt.show()
```

Finally, we test all our code so far with a simple image, which is the one rendered in the [Step 1 of the C++ tutorial](#).

```
let image = Image(192, 256)

for row in range(image.height):
    for col in range(image.width):
        image.set(
            row,
            col,
            Vec3f(Float32(row) / image.height, Float32(col) / image.width, 0),
        )

render(image)
```



Step 2: Ray tracing

Now we'll perform ray tracing from a camera into a scene with a sphere. Before reading the code below, we suggest you read more about how this works conceptually from [Step 2 of the C++ tutorial](#).

We first define the `Material` and `Sphere` structs, which are the new data structures we'll need.

```
from math import sqrt

@register_passable("trivial")
struct Material:
    var color: Vec3f
    var albedo: Vec3f
    var specular_component: Float32

    fn __init__(color: Vec3f) -> Material:
        return Material {
            color: color, albedo: Vec3f(0, 0, 0), specular_component: 0
        }

    fn __init__(
        color: Vec3f, albedo: Vec3f, specular_component: Float32
    ) -> Material:
        return Material {
            color: color, albedo: albedo, specular_component: specular_component
        }

alias W = 1024
alias H = 768
alias bg_color = Vec3f(0.02, 0.02, 0.02)
let shiny_yellow = Material(Vec3f(0.95, 0.95, 0.4), Vec3f(0.7, 0.6, 0), 30.0)
let green_rubber = Material(Vec3f(0.3, 0.7, 0.3), Vec3f(0.9, 0.1, 0), 1.0)

@register_passable("trivial")
struct Sphere:
    var center: Vec3f
    var radius: Float32
    var material: Material

    fn __init__(c: Vec3f, r: Float32, material: Material) -> Self:
        return Sphere {center: c, radius: r, material: material}

    @always_inline
    fn intersects(self, orig: Vec3f, dir: Vec3f, inout dist: Float32) -> Bool:
        """This method returns True if a given ray intersects this sphere.
        And if it does, it writes in the `dist` parameter the distance to the
        origin of the ray.
        """
        let L = orig - self.center
        let a = dir @ dir
        let b = 2 * (dir @ L)
        let c = L @ L - self.radius * self.radius
        let discriminant = b * b - 4 * a * c
        if discriminant < 0:
            return False
        if discriminant == 0:
            dist = -b / 2 * a
            return True
        let q = -0.5 * (b + sqrt(discriminant)) if b > 0 else -0.5 * (
            b - sqrt(discriminant)
        )
        var t0 = q / a
        let t1 = c / q
        if t0 > t1:
```

```

t0 = t1
if t0 < 0:
    t0 = t1
    if t0 < 0:
        return False

dist = t0
return True

```

We then define a `cast_ray` method, which will be used to figure out the color of a particular pixel in the image we'll produce. It basically works by identifying whether this ray intersects the sphere or not.

```

fn cast_ray(orig: Vec3f, dir: Vec3f, sphere: Sphere) -> Vec3f:
    var dist: Float32 = 0
    if not sphere.intersects(orig, dir, dist):
        return bg_color

    return sphere.material.color

```

Lastly, we parallelize the ray tracing for every pixel row-wise.

```

from math import tan, acos
from algorithm import parallelize

fn create_image_with_sphere(sphere: Sphere, height: Int, width: Int) -> Image:
    let image = Image(height, width)

    @parameter
    fn _process_row(row: Int):
        let y = -((2.0 * row + 1) / height - 1)
        for col in range(width):
            let x = ((2.0 * col + 1) / width - 1) * width / height
            let dir = Vec3f(x, y, -1).normalize()
            image.set(row, col, cast_ray(Vec3f.zero(), dir, sphere))

    parallelize[_process_row](height)
    return image

render(
    create_image_with_sphere(Sphere(Vec3f(-3, 0, -16), 2, shiny_yellow), H, W)
)

```



Step 3: More spheres

This section corresponds to the [Step 3 of the C++ tutorial](#).

We include here all the necessary changes:

- We add 3 more spheres to the scene, 2 of them being of ivory material.
- When we intersect the ray with the sphere, we render the color of the closest sphere.

```
from algorithm import parallelize
from utils.vector import DynamicVector
from math.limit import inf
```

```

fn scene_intersect(
    orig: Vec3f,
    dir: Vec3f,
    spheres: DynamicVector[Sphere],
    background: Material,
) -> Material:
    var spheres_dist = inf[DTType.float32]()
    var material = background

    for i in range(0, spheres.size()):
        var dist = inf[DTType.float32]()
        if spheres[i].intersects(orig, dir, dist) and dist < spheres_dist:
            spheres_dist = dist
            material = spheres[i].material

    return material

fn cast_ray(
    orig: Vec3f, dir: Vec3f, spheres: DynamicVector[Sphere]
) -> Material:
    let background = Material(Vec3f(0.02, 0.02, 0.02))
    return scene_intersect(orig, dir, spheres, background)

fn create_image_with_spheres(
    spheres: DynamicVector[Sphere], height: Int, width: Int
) -> Image:
    let image = Image(height, width)

    @parameter
    fn _process_row(row: Int):
        let y = -((2.0 * row + 1) / height - 1)
        for col in range(width):
            let x = ((2.0 * col + 1) / width - 1) * width / height
            let dir = Vec3f(x, y, -1).normalize()
            image.set(row, col, cast_ray(Vec3f.zero(), dir, spheres).color)

    parallelize[_process_row](height)

    return image

let spheres = DynamicVector[Sphere]()
spheres.push_back(Sphere(Vec3f(-3, 0, -16), 2, shiny_yellow))
spheres.push_back(Sphere(Vec3f(-1.0, -1.5, -12), 1.8, green_rubber))
spheres.push_back(Sphere(Vec3f( 1.5, -0.5, -18), 3, green_rubber))
spheres.push_back(Sphere(Vec3f( 7, 5, -18), 4, shiny_yellow))

render(create_image_with_spheres(spheres, H, W))
```



Step 4: Add lighting

This section corresponds to the [Step 4 of the C++ tutorial](#). Please read that section for an explanation of the trick used to estimate the light intensity of pixel based on the angle of intersection between each ray and the spheres. The changes are minimal and are primarily about handling this intersection angle.

```
@register_passable("trivial")
struct Light:
    var position: Vec3f
    var intensity: Float32

fn __init__(p: Vec3f, i: Float32) -> Self:
    return Light {position: p, intensity: i}__
```

```

from math import max

fn scene_intersect(
    orig: Vec3f,
    dir: Vec3f,
    spheres: DynamicVector[Sphere],
    inout material: Material,
    inout hit: Vec3f,
    inout N: Vec3f,
) -> Bool:
    var spheres_dist = inf[DType.float32]()

    for i in range(0, spheres.size):
        var dist: Float32 = 0
        if spheres[i].intersects(orig, dir, dist) and dist < spheres_dist:
            spheres_dist = dist
            hit = orig + dir * dist
            N = (hit - spheres[i].center).normalize()
            material = spheres[i].material

    return (spheres_dist != inf[DType.float32()]).__bool__()

fn cast_ray(
    orig: Vec3f,
    dir: Vec3f,
    spheres: DynamicVector[Sphere],
    lights: DynamicVector[Light],
) -> Material:
    var point = Vec3f.zero()
    var material = Material(Vec3f.zero())
    var N = Vec3f.zero()
    if not scene_intersect(orig, dir, spheres, material, point, N):
        return bg_color

    var diffuse_light_intensity: Float32 = 0
    for i in range(lights.size):
        let light_dir = (lights[i].position - point).normalize()
        diffuse_light_intensity += lights[i].intensity * max(0, light_dir @ N)

    return material.color * diffuse_light_intensity

fn create_image_with_spheres_and_lights(
    spheres: DynamicVector[Sphere],
    lights: DynamicVector[Light],
    height: Int,
    width: Int,
) -> Image:
    let image = Image(height, width)

    @parameter
    fn _process_row(row: Int):
        let y = -((2.0 * row + 1) / height - 1)
        for col in range(width):
            let x = ((2.0 * col + 1) / width - 1) * width / height
            let dir = Vec3f(x, y, -1).normalize()
            image.set(
                row, col, cast_ray(Vec3f.zero(), dir, spheres, lights).color
            )
    parallelize[_process_row](height)

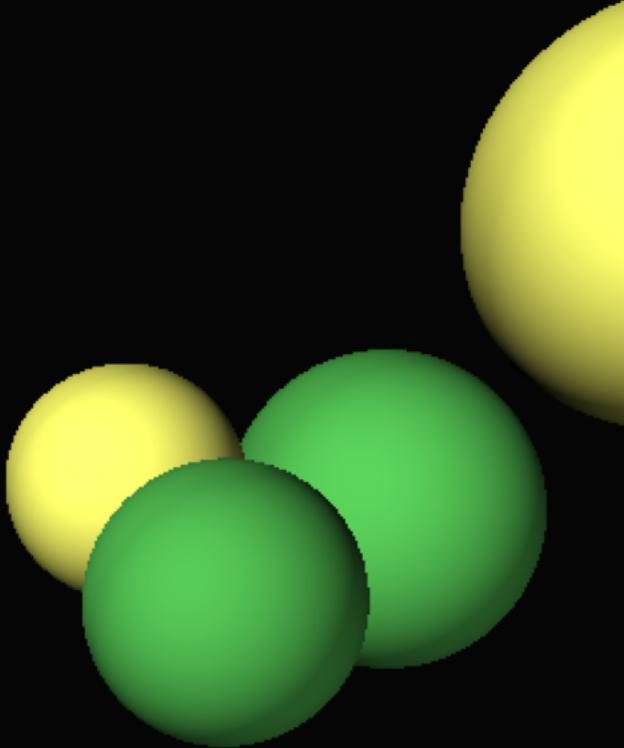
    return image

let lights = DynamicVector[Light]()
lights.push_back(Light(Vec3f(-20, 20, 20), 1.0))
lights.push_back(Light(Vec3f(20, -20, 20), 0.5))

render(create_image_with_spheres_and_lights(spheres, lights, H, W))_

```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



Step 5: Add specular lighting

This section corresponds to the [Step 5 of the C++ tutorial](#). The changes to the code are quite minimal, but the rendered picture looks much more realistic!

```
from math import pow

fn reflect(I: Vec3f, N: Vec3f) -> Vec3f:
    return I - N * (I @ N) * 2.0

fn cast_ray(
    orig: Vec3f,
```

```

dir: Vec3f,
spheres: DynamicVector[Sphere],
lights: DynamicVector[Light],
) -> Material:
    var point = Vec3f.zero()
    var material = Material(Vec3f.zero())
    var N = Vec3f.zero()
    if not scene_intersect(orig, dir, spheres, material, point, N):
        return bg_color

    var diffuse_light_intensity: Float32 = 0
    var specular_light_intensity: Float32 = 0
    for i in range(lights.size()):
        let light_dir = (lights[i].position - point).normalize()
        diffuse_light_intensity += lights[i].intensity * max(0, light_dir @ N)
        specular_light_intensity += (
            pow(
                max(0.0, -reflect(-light_dir, N) @ dir),
                material.specular_component,
            )
            * lights[i].intensity
        )

    let result = material.color * diffuse_light_intensity * material.albedo.data[
        0
    ] + Vec3f(
        1.0, 1.0, 1.0
    ) * specular_light_intensity * material.albedo.data[
        1
    ]
    let result_max = max(result[0], max(result[1], result[2]))
    # Cap the resulting vector
    if result_max > 1:
        return result * (1.0 / result_max)
    return result

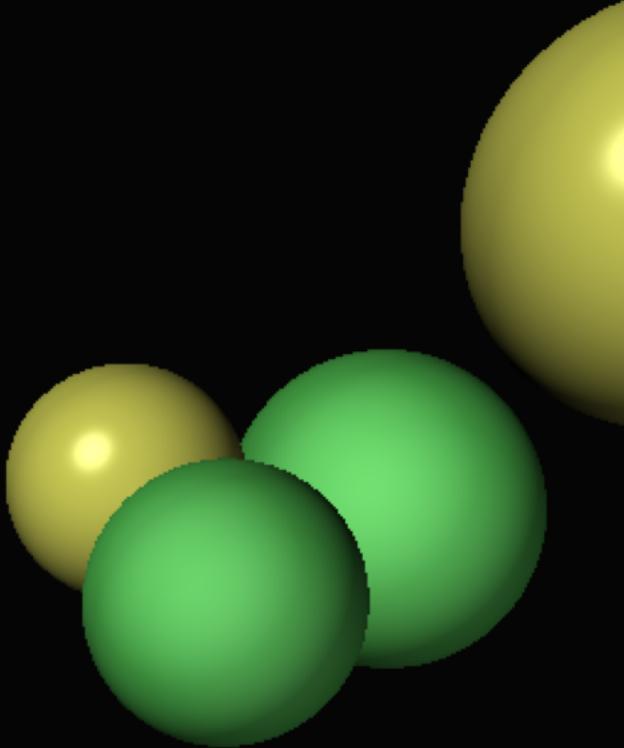
fn create_image_with_spheres_and_specular_lights(
    spheres: DynamicVector[Sphere],
    lights: DynamicVector[Light],
    height: Int,
    width: Int,
) -> Image:
    let image = Image(height, width)

    @parameter
    fn _process_row(row: Int):
        let y = -((2.0 * row + 1) / height - 1)
        for col in range(width):
            let x = ((2.0 * col + 1) / width - 1) * width / height
            let dir = Vec3f(x, y, -1).normalize()
            image.set(
                row, col, cast_ray(Vec3f.zero(), dir, spheres, lights).color
            )

    parallelize[_process_row](height)
    return image

render(create_image_with_spheres_and_specular_lights(spheres, lights, H, W)) ⊞

```



Step 6: Add background

As a last step, let's use an image for the background instead of a uniform fill. The only code that we need to change is the code where we used to return `bg_color`. Now we will determine a point in the background image to which the ray is directed and draw that.

```
from math import abs
```

```
fn cast_ray(  
    orig: Vec3f,  
    dir: Vec3f,  
    spheres: DynamicVector[Sphere],  
    lights: DynamicVector[Light],  
    bg: Image,
```

```

) -> Material:
var point = Vec3f.zero()
var material = Material(Vec3f.zero())
var N = Vec3f.zero()
if not scene.intersect(orig, dir, spheres, material, point, N):
    # Background
    # Given a direction vector `dir` we need to find a pixel in the image
    let x = dir[0]
    let y = dir[1]

    # Now map x from [-1,1] to [0,w-1] and do the same for y.
    let w = bg.width
    let h = bg.height
    let col = ((1.0 + x) * 0.5 * (w - 1)).to_int()
    let row = ((1.0 + y) * 0.5 * (h - 1)).to_int()
    return Material(bg.pixels[bg._pos_to_index(row, col)])

var diffuse_light_intensity: Float32 = 0
var specular_light_intensity: Float32 = 0
for i in range(lights.size()):
    let light_dir = (lights[i].position - point).normalize()
    diffuse_light_intensity += lights[i].intensity * max(0, light_dir @ N)
    specular_light_intensity += (
        pow(
            max(0.0, -reflect(-light_dir, N) @ dir),
            material.specular_component,
        )
        * lights[i].intensity
    )

let result = material.color * diffuse_light_intensity * material.albedo.data[
    0
] + Vec3f(
    1.0, 1.0, 1.0
) * specular_light_intensity * material.albedo.data[
    1
]
let result_max = max(result[0], max(result[1], result[2]))
# Cap the resulting vector
if result_max > 1:
    return result * (1.0 / result_max)
return result

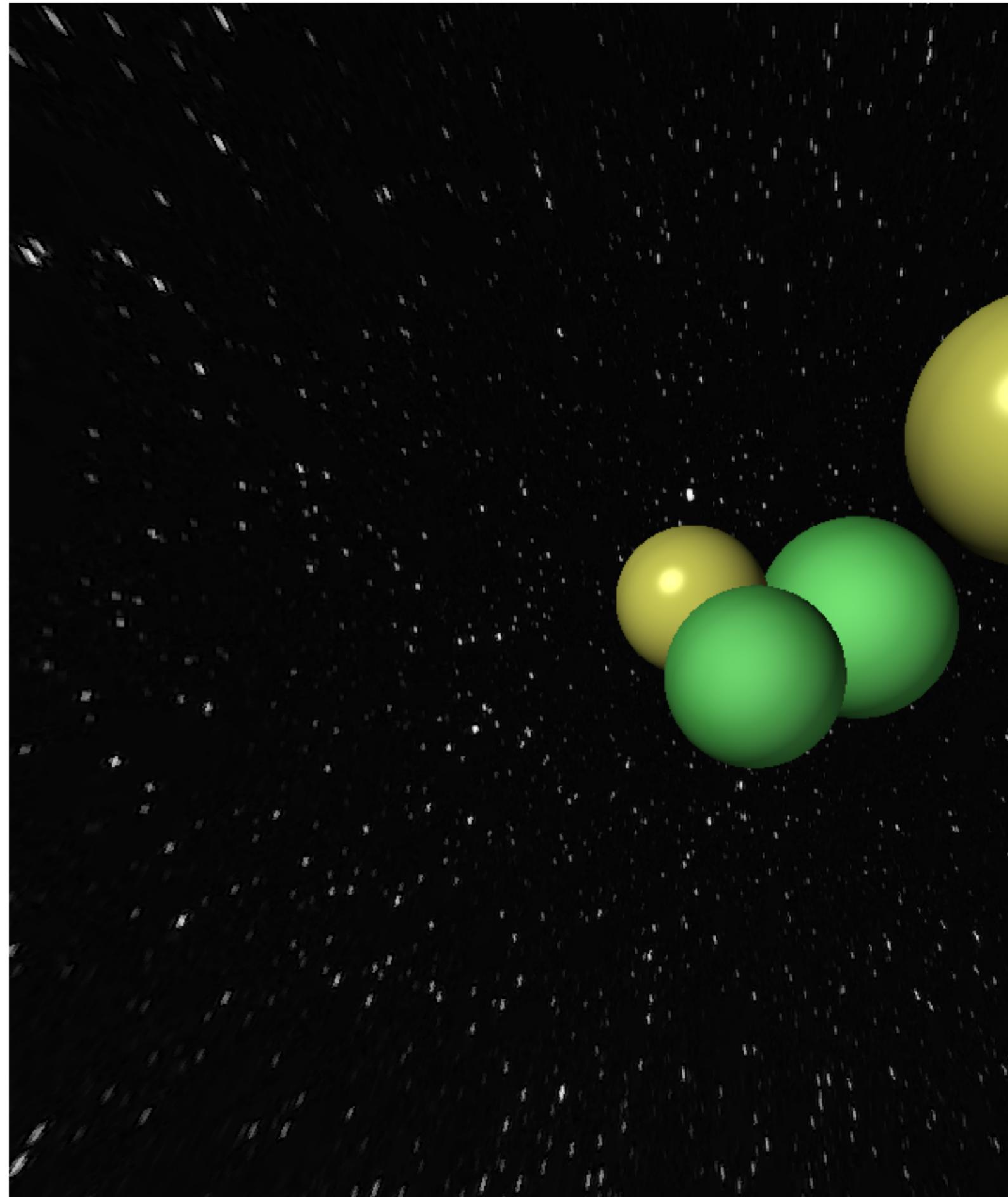
fn create_image_with_spheres_and_specular_lights(
    spheres: DynamicVector[Sphere],
    lights: DynamicVector[Light],
    height: Int,
    width: Int,
    bg: Image,
) -> Image:
    let image = Image(height, width)

    @parameter
    fn _process_row(row: Int):
        let y = -((2.0 * row + 1) / height - 1)
        for col in range(width):
            let x = ((2.0 * col + 1) / width - 1) * width / height
            let dir = Vec3f(x, y, -1).normalize()
            image.set(
                row, col, cast_ray(Vec3f.zero(), dir, spheres, lights, bg).color
            )

    parallelize[_process_row](height)
    return image

let bg = load_image("images/background.png")
render(
    create_image_with_spheres_and_specular_lights(spheres, lights, H, W, bg)
) ->

```



Next steps

We've only explored the basics of ray tracing here, but you can add shadows, reflections and so much more! Fortunately these are explained in [the C++ tutorial](#), and we leave the corresponding Mojo implementations as an exercise for you.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo[] modules

A list of all modules in the current standard library.

Mojo is a very young language, so these are the only modules we're making available at this time. There is much more to come!

Standard library modules

arg

[Implements functions and variables for interacting with execution and system environment.](#)

atomic

[Implements the Atomic class.](#)

autotuning

[Implements the autotune functionality.](#)

base64

[Provides functions for base64 encoding strings.](#)

benchmark

[Implements the Benchmark class for runtime benchmarking.](#)

bit

[Provides functions for bit manipulation.](#)

bool

[Implements the Bool class.](#)

buffer

[Implements the Buffer class.](#)

builtin_list

[Implements the ListLiteral class.](#)

builtin_slice

[Implements slice.](#)

complex

[Implements the Complex type.](#)

constrained

[Implements compile time constraints.](#)

coroutine

[Implements classes and methods for coroutines.](#)

debug_assert

[Implements a debug assert.](#)

dtype

[Implements the DTyPe class.](#)

env

[Implements basic routines for working with the OS.](#)

error

[Implements the Error class.](#)

file

[Implements the file based methods.](#)

float literal

[Implements the FloatLiteral class.](#)

functional

[Implements higher-order functions.](#)

index

[Implements StaticIntTuple which is commonly used to represent N-D indices.](#)

info

[Implements methods for querying the host target info.](#)

int

[Implements the Int class.](#)

intrinsics

[Defines intrinsics.](#)

io

[Provides utilities for working with input/output.](#)

len

[Provides the len function.](#)

limit

[Provides interfaces to query numeric various numeric properties of types.](#)

list

[Provides utilities for working with static and variadic lists.](#)

math

[Defines math utilities.](#)

memory

[Defines functions for memory manipulations.](#)

numerics

[Defines utilities to work with numeric types.](#)

object

[Defines the object type, which is used to represent untyped values.](#)

object

[Implements PyObject.](#)

param_env

[Implements functions for retrieving compile-time defines.](#)

path

Aliases:

polynomial

[Provides two implementations for evaluating polynomials.](#)

python

[Implements Python interoperability.](#)

random

[Provides functions for random numbers.](#)

range

[Implements a 'range' call.](#)

rebind

[Implements type rebind.](#)

reduction

[Implements SIMD reductions.](#)

simd

[Implements SIMD struct.](#)

sort

[Implements sorting functions.](#)

static_tuple

[Implements StaticTuple, a statically-sized uniform container.](#)

string

[Implements basic object methods for working with strings.](#)

string_literal

[Implements the StringLiteral class.](#)

StringRef

[Implements the StringRef class.](#)

tensor

[Implements the Tensor type.](#)

tensor_shape

[Implements the TensorShape type.](#)

tensor_spec

[Implements the TensorSpec type.](#)

testing

[Implements various testing utils.](#)

time

[Implements basic utils for working with time.](#)

tuple

[Implements the Tuple type.](#)

type_aliases

[Defines some type aliases.](#)

unsafe

[Implements classes for working with unsafe pointers.](#)

[vector](#)

[Defines several vector-like classes.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

functional

Module

Implements higher-order functions.

You can import these APIs from the `algorithm` package. For example:

```
from algorithm import map
```

Aliases:

- `Static1DTileUnitFunc = fnInt capturing -> None`: Signature of a 1d tiled function that performs some work with a static tile size and an offset. i.e. `func<tile_size: Int>(offset: Int)`
- `Dynamic1DTileUnitFunc = fn(Int, Int) capturing -> None`: Signature of a 1d tiled function that performs some work with a dynamic tile size and an offset. i.e. `func(offset: Int, tile_size: Int)`
- `BinaryTile1DTileUnitFunc = fn[Int](Int, Int) capturing -> None`: Signature of a tiled function that performs some work with a dynamic tile size and a secondary static tile size.
- `Static2DTileUnitFunc = fnInt, Int capturing -> None`: Signature of a 2d tiled function that performs some work with a static tile size and an offset. i.e. `func<tile_size_x: Int, tile_size_y: Int>(offset_x: Int, offset_y: Int)`
- `SwitchedFunction = fn[Bool]() capturing -> None`
- `SwitchedFunction2 = fn[Bool, Bool]() capturing -> None`
- `Static1DTileUnswitchUnitFunc = fn[Int, Bool](Int, Int) capturing -> None`: Signature of a tiled function that performs some work with a static tile size and an offset. i.e. `func<tile_size: Int>(offset: Int)`
- `Dynamic1DTileUnswitchUnitFunc = fn[Bool](Int, Int, Int) capturing -> None`

map

```
map[func: fn(Int) capturing -> None](size: Int)
```

Maps a function over a range from 0 to size.

Parameters:

- **func** (`fn(Int) capturing -> None`): Function to map.

Args:

- **size** (`Int`): The number of elements.

unroll

```
unroll[count: Int, func: fn[Int]() capturing -> None]()
```

Repeatedly evaluates a function `count` times.

Parameters:

- **count** (`Int`): A number of repetitions.
- **func** (`fn[Int]() capturing -> None`): The function to evaluate. The function should take a single `Int` argument.

```
unroll[dim0: Int, dim1: Int, func: fn[Int, Int]() capturing -> None]()
```

Repeatedly evaluates a 2D nested loop.

Parameters:

- **dim0** (Int): The first dimension size.
- **dim1** (Int): The second dimension size.
- **func** (fn[Int, Int]() capturing -> None): The function to evaluate. The function should take two Int arguments.

```
unroll[dim0: Int, dim1: Int, dim2: Int, func: fn[Int, Int, Int]() capturing -> None]()
```

Repeatedly evaluates a 3D nested loop.

Parameters:

- **dim0** (Int): The first dimension size.
- **dim1** (Int): The second dimension size.
- **dim2** (Int): The third dimension size.
- **func** (fn[Int, Int, Int]() capturing -> None): The function to evaluate. The function should take three Int arguments.

vectorize

```
vectorize[simd_width: Int, func: fn[Int](Int) capturing -> None](size: Int)
```

Maps a function which is parametrized over a simd_width over a range from 0 to size in SIMD fashion.

Parameters:

- **simd_width** (Int): The SIMD vector width.
- **func** (fnInt capturing -> None): The function for the loop body.

Args:

- **size** (Int): The total loop count.

vectorize_unroll

```
vectorize_unroll[simd_width: Int, unroll_factor: Int, func: fn[Int](Int) capturing -> None](size: Int)
```

Maps a function which is parametrized over a simd_width over a range from 0 to size in SIMD fashion and unroll the loop by unroll_factor.

Parameters:

- **simd_width** (Int): The SIMD vector width.
- **unroll_factor** (Int): The unroll factor for the main loop.
- **func** (fnInt capturing -> None): The function for the loop body.

Args:

- **size** (Int): The total loop count.

async_parallelize

```
async_parallelize[func: fn(Int) capturing -> None](out_chain: OutputChainPtr, num_work_items: Int)
```

Executes func(0) ... func(num_work_items-1) as sub-tasks in parallel and returns immediately. The out_chain will be marked as ready only when all sub-tasks have completed.

Execute func(0) ... func(num_work_items-1) as sub-tasks in parallel and mark out_chain as ready when all functions have returned. This function will return when the sub-tasks have been scheduled but not necessarily completed. The runtime may execute the sub-tasks in any order and with any degree of concurrency.

All free variables in func must be “async safe”. Currently this means: - The variable must be bound by a by-val function argument (ie no &), or let binding. - The variable’s type must be “async safe”, ie is marked as @register_passable and any internal pointers are to memory with lifetime at least until out_chain is ready. In practice, this means only pointers to buffers held alive by the runtime. Consider using sync_parallelize if this requirement is too onerous.

If num_work_items is 0 then the out_chain is marked as ready before async_parallelize returns. If num_work_items is 1 then func(0) may still be executed as a sub-task.

Parameters:

- **func** (fn(Int) capturing -> None): The function to invoke.

Args:

- **out_chain** (OutputChainPtr): Out chain onto which to signal completion.
- **num_work_items** (Int): Number of parallel tasks.

sync_parallelize

```
sync_parallelize[func: fn(Int) capturing -> None](out_chain: OutputChainPtr, num_work_items: Int)
```

Executes func(0) ... func(num_work_items-1) as sub-tasks in parallel. Marks out_chain as ready and returns only when all sub-tasks have completed.

Execute func(0) ... func(num_work_items-1) as sub-tasks in parallel, and return only when they have all functions have returned. The runtime may execute the sub-tasks in any order and with any degree of concurrency. The out_chain will be marked as ready before returning.

Parameters:

- **func** (fn(Int) capturing -> None): The function to invoke.

Args:

- **out_chain** (OutputChainPtr): Out chain onto which to signal completion.
- **num_work_items** (Int): Number of parallel tasks.

parallelize

```
parallelize[func: fn(Int) capturing -> None]()
```

Executes func(0) ... func(N-1) as sub-tasks in parallel and returns when all are complete. N is chosen to be the number of processors on the system.

Execute func(0) ... func(N-1) as sub-tasks in parallel. This function will return only after all the sub-tasks have completed.

CAUTION: Creates and destroys a local runtime! Do not use from kernels!

Parameters:

- **func** (fn(Int) capturing -> None): The function to invoke.

parallelize[func: fn(Int) capturing -> None](num_work_items: Int)

Executes func(0) ... func(num_work_items-1) as sub-tasks in parallel and returns when all are complete.

Execute func(0) ... func(num_work_items-1) as sub-tasks in parallel. This function will return only after all the sub-tasks have completed.

CAUTION: Creates and destroys a local runtime! Do not use from kernels!

Parameters:

- **func** (fn(Int) capturing -> None): The function to invoke.

Args:

- **num_work_items** (Int): Number of parallel tasks.

parallelize[func: fn(Int) capturing -> None](num_work_items: Int, num_workers: Int)

Executes func(0) ... func(num_work_items-1) as sub-tasks in parallel and returns when all are complete.

Execute func(0) ... func(num_work_items-1) as sub-tasks in parallel. This function will return only after all the sub-tasks have completed.

Parameters:

- **func** (fn(Int) capturing -> None): The function to invoke.

Args:

- **num_work_items** (Int): Number of parallel tasks.
- **num_workers** (Int): The number of works to use for execution.

tile

tile[workgroup_function: fnInt capturing -> None, tile_size_list: VariadicList[Int]](offset: Int, upperbound: Int)

A generator that launches work groups in specified list of tile sizes.

A workgroup function is a function that can process a configurable consecutive “tile” of workload. E.g. `work_on[3](5)` should launch computation on item 5,6,7, and should be semantically equivalent to `work_on[1](5), work_on[1](6), work_on[1](7)`.

This generator will try to proceed with the given list of tile sizes on the listed order. E.g. `tile[func, (3,2,1)](offset, upperbound)` will try to call `func[3]` starting from offset until remaining work is less than 3 from upperbound and then try `func[2]`, and then `func[1]`, etc.

Parameters:

- **workgroup_function** (fnInt capturing -> None): Workgroup function that processes one tile of workload.
- **tile_size_list** (VariadicList[Int]): List of tile sizes to launch work.

Args:

- **offset** (Int): The initial index to start the work from.
- **upperbound** (Int): The runtime upperbound that the work function should not exceed.

tile[workgroup_function: fn(Int, Int) capturing -> None](offset: Int, upperbound: Int, tile_size_list: VariadicList[Int])

A generator that launches work groups in specified list of tile sizes.

This is the version of tile generator for the case where work_group function can take the tile size as a runtime value.

Parameters:

- **workgroup_function** (fn(Int, Int) capturing -> None): Workgroup function that processes one tile of workload.

Args:

- **offset** (Int): The initial index to start the work from.
- **upperbound** (Int): The runtime upperbound that the work function should not exceed.
- **tile_size_list** (VariadicList[Int]): List of tile sizes to launch work.

```
tile[secondary_tile_size_list: VariadicList[Int], secondary_cleanup_tile: Int, workgroup_function: fn[Int](Int, Int) capturing -> None](offset: Int, upperbound: Int, primary_tile_size_list: VariadicList[Int], primary_cleanup_tile: Int)
```

A generator that launches work groups in specified list of tile sizes until the sum of primary_tile_sizes has exceeded the upperbound.

Parameters:

- **secondary_tile_size_list** (VariadicList[Int]): List of static tile sizes to launch work.
- **secondary_cleanup_tile** (Int): Last static tile to use when primary tile sizes don't fit exactly within the upperbound.
- **workgroup_function** (fn[Int](Int, Int) capturing -> None): Workgroup function that processes one tile of workload.

Args:

- **offset** (Int): The initial index to start the work from.
- **upperbound** (Int): The runtime upperbound that the work function should not exceed.
- **primary_tile_size_list** (VariadicList[Int]): List of dynamic tile sizes to launch work.
- **primary_cleanup_tile** (Int): Last dynamic tile to use when primary tile sizes don't fit exactly within the upperbound.

```
tile[workgroup_function: fn[Int, Int](Int, Int) capturing -> None, tile_sizes_x: VariadicList[Int], tile_sizes_y: VariadicList[Int]](offset_x: Int, offset_y: Int, upperbound_x: Int, upperbound_y: Int)
```

Launches workgroup_function using the largest tile sizes possible in each dimension, starting from the x and y offset, until the x and y upperbounds are reached.

Parameters:

- **workgroup_function** (fnInt, Int capturing -> None): Function that is invoked for each tile and offset.
- **tile_sizes_x** (VariadicList[Int]): List of tile sizes to use for the first parameter of workgroup_function.
- **tile_sizes_y** (VariadicList[Int]): List of tile sizes to use for the second parameter of workgroup_function.

Args:

- **offset_x** (Int): Initial x offset passed to workgroup_function.
- **offset_y** (Int): Initial y offset passed to workgroup_function.
- **upperbound_x** (Int): Max offset in x dimension passed to workgroup function.
- **upperbound_y** (Int): Max offset in y dimension passed to workgroup function.

unswitch

```
unswitch[switched_func: fn[Bool]() capturing -> None](dynamic_switch: Bool)
```

Performs a functional unswitch transformation.

Unswitch is a simple pattern that is similar idea to loop unswitching pass but extended to functional patterns. The pattern facilitates the following code transformation that reduces the number of branches in the generated code

Before:

```
for i in range(...)  
    if i < xxx:  
        ...
```

After:

```
if i < ...  
    for i in range(...)  
        ...  
else  
    for i in range(...)  
        if i < xxx:  
            ...
```

This unswitch function generalizes that pattern with the help of meta parameters and can be used to perform both loop unswitching and other tile predicate lifting like in simd and amx.

TODO: Generalize to support multiple predicates. TODO: Once nested lambdas compose well should make unswitch compose with tile in an easy way.

Parameters:

- **switched_func** (fn[Bool]() capturing -> None): The function containing the inner loop logic that can be unswitched.

Args:

- **dynamic_switch** (Bool): The dynamic condition that enables the unswitched code path.

```
unswitch[switched_func: fn[Bool, Bool]() capturing -> None](dynamic_switch_a: Bool,  
dynamic_switch_b: Bool)
```

Performs a functional 2-predicates unswitch transformation.

Parameters:

- **switched_func** (fn[Bool, Bool]() capturing -> None): The function containing the inner loop logic that has 2 predicates which can be unswitched.

Args:

- **dynamic_switch_a** (Bool): The first dynamic condition that enables the outer unswitched code path.
- **dynamic_switch_b** (Bool): The second dynamic condition that enables the inner unswitched code path.

tile_and_unswitch

```
tile_and_unswitch[workgroup_function: fn[Int, Bool](Int, Int) capturing -> None, tile_size_list:  
VariadicList[Int]](offset: Int, upperbound: Int)
```

Performs time and unswitch functional transformation.

A variant of static tile given a workgroup function that can be unswitched. This generator is a fused version of tile and unswitch, where the static unswitch is true throughout the “inner” portion of the workload and is false only on the residue tile.

Parameters:

- **workgroup_function** (fn[Int, Bool](Int, Int) capturing -> None): Workgroup function that processes one tile of workload.
- **tile_size_list** (VariadicList[Int]): List of tile sizes to launch work.

Args:

- **offset** (Int): The initial index to start the work from.
- **upperbound** (Int): The runtime upperbound that the work function should not exceed.

```
tile_and_unswitch[workgroup_function: fn[Bool](Int, Int, Int) capturing -> None](offset: Int,  
upperbound: Int, tile_size_list: VariadicList[Int])
```

Performs time and unswitch functional transformation.

A variant of dynamic tile given a workgroup function that can be unswitched. This generator is a fused version of tile and unswitch, where the static unswitch is true throughout the “inner” portion of the workload and is false only on the residue tile.

Parameters:

- **workgroup_function** (fn[Bool](Int, Int, Int) capturing -> None): Workgroup function that processes one tile of workload.

Args:

- **offset** (Int): The initial index to start the work from.
- **upperbound** (Int): The runtime upperbound that the work function should not exceed.
- **tile_size_list** (VariadicList[Int]): List of tile sizes to launch work.

elementwise

```
elementwise[rank: Int, simd_width: Int, func: fn[Int, Int](StaticIntTuple[*(0,1)]) capturing ->  
None](shape: StaticIntTuple[rank], out_chain: OutputChainPtr)
```

Executes func[width, rank](indices) as sub-tasks for a suitable combination of width and indices so as to cover shape.

Parameters:

- **rank** (Int): The rank of the buffer.
- **simd_width** (Int): The SIMD vector width to use.
- **func** (fn[Int, Int](StaticIntTuple[*(0,1)]) capturing -> None): The body function.

Args:

- **shape** (StaticIntTuple[rank]): The shape of the buffer.
- **out_chain** (OutputChainPtr): The our chain to attach results to.

parallelize_over_rows

```
parallelize_over_rows[rank: Int, func: fn(Int, Int) capturing -> None](shape: StaticIntTuple[rank],  
axis: Int, out_chain: OutputChainPtr, grain_size: Int)
```

Parallelize func over non-axis dims of shape.

Parameters:

- **rank** (Int): Rank of shape.
- **func** (fn(Int, Int) capturing -> None): Function to call on range of rows.

Args:

- **shape** (StaticIntTuple[rank]): Shape to parallelize over.
- **axis** (Int): Rows are slices along the axis dimension of shape.
- **out_chain** (OutputChainPtr): The out chain to attach results to.
- **grain_size** (Int): The minimum number of elements to warrant using an additional thread.

reduction

Module

Implements SIMD reductions.

You can import these APIs from the `algorithm` package. For example:

```
from algorithm import map_reduce
```

map_reduce

```
map_reduce[simd_width: Int, size: Dim, type: DType, acc_type: DType, input_gen_fn: fn[DType, Int](Int) capturing -> SIMD[*(0,0), *(0,1)], reduce_vec_to_vec_fn: fn[DType, DType, Int](SIMD[*(0,0), *(0,2)], SIMD[*(0,1), *(0,2)]) capturing -> SIMD[*(0,0), *(0,2)], reduce_vec_to_scalar_fn: fn[DType, Int](SIMD[*(0,0), *(0,1)]) -> SIMD[*(0,0), 1]](dst: Buffer[size, type], init: SIMD[acc_type, 1]) -> SIMD[acc_type, 1]
```

Stores the result of calling `input_gen_fn` in `dst` and simultaneously reduce the result using a custom reduction function.

Parameters:

- **simd_width** (Int): The vector width for the computation.
- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.
- **acc_type** (DType): The dtype of the reduction accumulator.
- **input_gen_fn** (fn[DType, Int](Int) capturing -> SIMD[*(0,0), *(0,1)]): A function that generates inputs to reduce.
- **reduce_vec_to_vec_fn** (fn[DType, DType, Int](SIMD[*(0,0), *(0,2)], SIMD[*(0,1), *(0,2)]) capturing -> SIMD[*(0,0), *(0,2)]): A mapping function. This function is used to combine (accumulate) two chunks of input data: e.g. we load two 8xfloat32 vectors of elements and need to reduce them into a single 8xfloat32 vector.
- **reduce_vec_to_scalar_fn** (fn[DType, Int](SIMD[*(0,0), *(0,1)]) -> SIMD[*(0,0), 1]): A reduction function. This function is used to reduce a vector to a scalar. E.g. when we got 8xfloat32 vector and want to reduce it to an float32 scalar.

Args:

- **dst** (Buffer[size, type]): The output buffer.
- **init** (SIMD[acc_type, 1]): The initial value to use in accumulator.

Returns:

The computed reduction value.

reduce

```
reduce[simd_width: Int, size: Dim, type: DType, acc_type: DType, map_fn: fn[DType, DType, Int](SIMD[*(0,0), *(0,2)], SIMD[*(0,1), *(0,2)]) capturing -> SIMD[*(0,0), *(0,2)], reduce_fn: fn[DType, Int](SIMD[*(0,0), *(0,1)]) -> SIMD[*(0,0), 1]](src: Buffer[size, type], init: SIMD[acc_type, 1]) -> SIMD[acc_type, 1]
```

Computes a custom reduction of buffer elements.

Parameters:

- **simd_width** (Int): The vector width for the computation.
- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

- **acc_type** (DType): The dtype of the reduction accumulator.
- **map_fn** (fn[DType, DType, Int](SIMD[*(0,0), *(0,2)], SIMD[*(0,1), *(0,2)]) capturing -> SIMD[*(0,0), *(0,2)]): A mapping function. This function is used when to combine (accumulate) two chunks of input data: e.g. we load two 8xfloat32 vectors of elements and need to reduce them to a single 8xfloat32 vector.
- **reduce_fn** (fn[DType, Int](SIMD[*(0,0), *(0,1)]) -> SIMD[*(0,0), 1]): A reduction function. This function is used to reduce a vector to a scalar. E.g. when we got 8xfloat32 vector and want to reduce it to 1xfloat32.

Args:

- **src** (Buffer[size, type]): The input buffer.
- **init** (SIMD[acc_type, 1]): The initial value to use in accumulator.

Returns:

The computed reduction value.

```
reduce[simd_width: Int, rank: Int, input_shape: DimList, output_shape: DimList, type: DType,
acc_type: DType, map_fn: fn[DType, DType, Int](SIMD[*(0,0), *(0,2)], SIMD[*(0,1), *(0,2)])
capturing -> SIMD[*(0,0), *(0,2)], reduce_fn: fn[DType, Int](SIMD[*(0,0), *(0,1)]) -> SIMD[*(0,0),
1], reduce_axis: Int](src: NDBuffer[rank, input_shape, type], dst: NDBuffer[rank, output_shape,
acc_type], init: SIMD[acc_type, 1])
```

Performs a reduction across `reduce_axis` of an `NDBuffer` (`src`) and stores the result in an `NDBuffer` (`dst`).

First `src` is reshaped into a 3D tensor. Without loss of generality, the three axes will be referred to as [H,W,C], where the axis to reduce across is W, the axes before the reduce axis are packed into H, and the axes after the reduce axis are packed into C. i.e. a tensor with dims [D1, D2, ..., Di, ..., Dn] reducing across axis i gets packed into a 3D tensor with dims [H, W, C], where H=prod(D1,...,Di-1), W = Di, and C = prod(Di+1,...,Dn).

Parameters:

- **simd_width** (Int): The vector width for the computation.
- **rank** (Int): The rank of the input/output buffers.
- **input_shape** (DimList): The input buffer shape.
- **output_shape** (DimList): The output buffer shape.
- **type** (DType): The buffer elements dtype.
- **acc_type** (DType): The dtype of the reduction accumulator.
- **map_fn** (fn[DType, DType, Int](SIMD[*(0,0), *(0,2)], SIMD[*(0,1), *(0,2)]) capturing -> SIMD[*(0,0), *(0,2)]): A mapping function. This function is used when to combine (accumulate) two chunks of input data: e.g. we load two 8xfloat32 vectors of elements and need to reduce them to a single 8xfloat32 vector.
- **reduce_fn** (fn[DType, Int](SIMD[*(0,0), *(0,1)]) -> SIMD[*(0,0), 1]): A reduction function. This function is used to reduce a vector to a scalar. E.g. when we got 8xfloat32 vector and want to reduce it to 1xfloat32.
- **reduce_axis** (Int): The axis to reduce across.

Args:

- **src** (NDBuffer[rank, input_shape, type]): The input buffer.
- **dst** (NDBuffer[rank, output_shape, acc_type]): The output buffer.
- **init** (SIMD[acc_type, 1]): The initial value to use in accumulator.

reduce_boolean

```
reduce_boolean[simd_width: Int, size: Dim, type: DType, reduce_fn: fn[DType, Int](SIMD[*(0,0), *
(0,1)]) capturing -> Bool, continue_fn: fn(Bool) capturing -> Bool](src: Buffer[size, type], init:
Bool) -> Bool
```

Computes a bool reduction of buffer elements. The reduction will early exit if the `continue_fn` returns False.

Parameters:

- **simd_width** (Int): The vector width for the computation.
- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.
- **reduce_fn** (fn[DType, Int](SIMD[*(0,0), *(0,1)]) capturing -> Bool): A boolean reduction function. This function is used to reduce a vector to a scalar. E.g. when we got 8xfloat32 vector and want to reduce it to a bool.
- **continue_fn** (fn(Bool) capturing -> Bool): A function to indicate whether we want to continue processing the rest of the iterations. This takes the result of the `reduce_fn` and returns True to continue processing and False to early exit.

Args:

- **src** (Buffer[size, type]): The input buffer.
- **init** (Bool): The initial value to use.

Returns:

The computed reduction value.

max

```
max[size: Dim, type: DType](src: Buffer[size, type]) -> SIMD[type, 1]
```

Computes the max element in a buffer.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

The maximum of the buffer elements.

```
max[rank: Int, input_shape: DimList, output_shape: DimList, type: DType, reduce_axis: Int](src: NDBuffer[rank, input_shape, type], dst: NDBuffer[rank, output_shape, type])
```

Computes the max across `reduce_axis` of an NDBuffer.

Parameters:

- **rank** (Int): The rank of the input/output buffers.
- **input_shape** (DimList): The input buffer shape.
- **output_shape** (DimList): The output buffer shape.
- **type** (DType): The buffer elements dtype.
- **reduce_axis** (Int): The axis to reduce across.

Args:

- **src** (NDBuffer[rank, input_shape, type]): The input buffer.
- **dst** (NDBuffer[rank, output_shape, type]): The output buffer.

min

```
min[size: Dim, type: DType](src: Buffer[size, type]) -> SIMD[type, 1]
```

Computes the min element in a buffer.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

The minimum of the buffer elements.

```
min[rank: Int, input_shape: DimList, output_shape: DimList, type: DType, reduce_axis: Int](src: NDBuffer[rank, input_shape, type], dst: NDBuffer[rank, output_shape, type])
```

Computes the min across reduce_axis of an NDBuffer.

Parameters:

- **rank** (Int): The rank of the input/output buffers.
- **input_shape** (DimList): The input buffer shape.
- **output_shape** (DimList): The output buffer shape.
- **type** (DType): The buffer elements dtype.
- **reduce_axis** (Int): The axis to reduce across.

Args:

- **src** (NDBuffer[rank, input_shape, type]): The input buffer.
- **dst** (NDBuffer[rank, output_shape, type]): The output buffer.

sum

```
sum[size: Dim, type: DType](src: Buffer[size, type]) -> SIMD[type, 1]
```

Computes the sum of buffer elements.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

The sum of the buffer elements.

```
sum[rank: Int, input_shape: DimList, output_shape: DimList, type: DType, reduce_axis: Int](src: NDBuffer[rank, input_shape, type], dst: NDBuffer[rank, output_shape, type])
```

Computes the sum across reduce_axis of an NDBuffer.

Parameters:

- **rank** (Int): The rank of the input/output buffers.
- **input_shape** (DimList): The input buffer shape.

- **output_shape** (DimList): The output buffer shape.
- **type** (DType): The buffer elements dtype.
- **reduce_axis** (Int): The axis to reduce across.

Args:

- **src** (NDBuffer[rank, input_shape, type]): The input buffer.
- **dst** (NDBuffer[rank, output_shape, type]): The output buffer.

product

product[size: Dim, type: DType](src: Buffer[size, type]) -> SIMD[type, 1]

Computes the product of the buffer elements.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

The product of the buffer elements.

product[rank: Int, input_shape: DimList, output_shape: DimList, type: DType, reduce_axis: Int](src: NDBuffer[rank, input_shape, type], dst: NDBuffer[rank, output_shape, type])

Computes the product across reduce_axis of an NDBuffer.

Parameters:

- **rank** (Int): The rank of the input/output buffers.
- **input_shape** (DimList): The input buffer shape.
- **output_shape** (DimList): The output buffer shape.
- **type** (DType): The buffer elements dtype.
- **reduce_axis** (Int): The axis to reduce across.

Args:

- **src** (NDBuffer[rank, input_shape, type]): The input buffer.
- **dst** (NDBuffer[rank, output_shape, type]): The output buffer.

mean

mean[size: Dim, type: DType](src: Buffer[size, type]) -> SIMD[type, 1]

Computes the mean value of the elements in a buffer.

Parameters:

- **size** (Dim): The size of the input buffer..
- **type** (DType): The type of the elements of the input buffer and output SIMD vector.

Args:

- **src** (Buffer[size, type]): The buffer of elements for which the mean is computed.

Returns:

The mean value of the elements in the given buffer.

```
mean[rank: Int, input_shape: DimList, output_shape: DimList, type: DType, reduce_axis: Int](src: NDBuffer[rank, input_shape, type], dst: NDBuffer[rank, output_shape, type])
```

Computes the mean across reduce_axis of an NDBuffer.

Parameters:

- **rank** (Int): The rank of the input/output buffers.
- **input_shape** (DimList): The input buffer shape.
- **output_shape** (DimList): The output buffer shape.
- **type** (DType): The buffer elements dtype.
- **reduce_axis** (Int): The axis to reduce across.

Args:

- **src** (NDBuffer[rank, input_shape, type]): The input buffer.
- **dst** (NDBuffer[rank, output_shape, type]): The output buffer.

variance

```
variance[size: Dim, type: DType](src: Buffer[size, type], mean_value: SIMD[type, 1], correction: Int) -> SIMD[type, 1]
```

Given a mean, computes the variance of elements in a buffer.

The mean value is used to avoid a second pass over the data:

```
variance = sum((x - E(x))^2) / (size - correction)
```

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.
- **mean_value** (SIMD[type, 1]): The mean value of the buffer.
- **correction** (Int): Normalize variance by size - correction.

Returns:

The variance value of the elements in a buffer.

```
variance[size: Dim, type: DType](src: Buffer[size, type], correction: Int) -> SIMD[type, 1]
```

Computes the variance value of the elements in a buffer.

```
variance(src) = sum((x - E(x))^2) / (size - correction)
```

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.
- **correction** (Int): Normalize variance by size - correction (Default=1).

Returns:

The variance value of the elements in a buffer.

all_true

```
all_true[size: Dim, type: DType](src: Buffer[size, type]) -> Bool
```

Returns True if all the elements in a buffer are True and False otherwise.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

True if all of the elements of the buffer are True and False otherwise.

any_true

```
any_true[size: Dim, type: DType](src: Buffer[size, type]) -> Bool
```

Returns True if any the elements in a buffer are True and False otherwise.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

True if any of the elements of the buffer are True and False otherwise.

none_true

```
none_true[size: Dim, type: DType](src: Buffer[size, type]) -> Bool
```

Returns True if none of the elements in a buffer are True and False otherwise.

Parameters:

- **size** (Dim): The buffer size.
- **type** (DType): The buffer elements dtype.

Args:

- **src** (Buffer[size, type]): The buffer.

Returns:

True if none of the elements of the buffer are True and False otherwise.

argmax

```
argmax[type: DType, out_type: DType, rank: Int](input: NDBuffer[rank,  
create_unknown[$builtin::$int::Int][rank](), type], axis: Int, output: NDBuffer[rank,  
create_unknown[$builtin::$int::Int][rank](), out_type], out_chain: OutputChainPtr)
```

Finds the indices of the maximum element along the specified axis.

Parameters:

- **type** (DType): Type of the input tensor.
- **out_type** (DType): Type of the output tensor.
- **rank** (Int): The rank of the input / output.

Args:

- **input** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), type]): The input tensor.
- **axis** (Int): The axis.
- **output** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), out_type]): The output tensor.
- **out_chain** (OutputChainPtr): The chain to attach results to.

```
argmax[type: DType, out_type: DType, axis_type: DType, rank: Int](input: NDBuffer[rank,  
create_unknown[$builtin::$int::Int][rank](), type], axis_buf: NDBuffer[1,  
create_unknown[$builtin::$int::Int][1](), axis_type], output: NDBuffer[rank,  
create_unknown[$builtin::$int::Int][rank](), out_type], out_chain: OutputChainPtr)
```

Finds the indices of the maximum element along the specified axis.

Parameters:

- **type** (DType): Type of the input tensor.
- **out_type** (DType): Type of the output tensor.
- **axis_type** (DType): Type of the axis tensor.
- **rank** (Int): The rank of the input / output.

Args:

- **input** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), type]): The input tensor.
- **axis_buf** (NDBuffer[1, create_unknown[\$builtin::\$int::Int][1](), axis_type]): The axis tensor.
- **output** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), out_type]): The axis tensor.
- **out_chain** (OutputChainPtr): The chain to attach results to.

argmin

```
argmin[type: DType, out_type: DType, rank: Int](input: NDBuffer[rank,  
create_unknown[$builtin::$int::Int][rank](), type], axis: Int, output: NDBuffer[rank,  
create_unknown[$builtin::$int::Int][rank](), out_type], out_chain: OutputChainPtr)
```

Finds the indices of the minimum element along the specified axis.

Parameters:

- **type** (DType): Type of the input tensor.
- **out_type** (DType): Type of the output tensor.
- **rank** (Int): The rank of the input / output.

Args:

- **input** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), type]): The input tensor.
- **axis** (Int): The axis.
- **output** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), out_type]): The output tensor.
- **out_chain** (OutputChainPtr): The chain to attach results to.

```
argmin[type: DType, out_type: DType, axis_type: DType, rank: Int](input: NDBuffer[rank,
create_unknown[$builtin::$int::Int][rank](), type], axis_buf: NDBuffer[1,
create_unknown[$builtin::$int::Int][1](), axis_type], output: NDBuffer[rank,
create_unknown[$builtin::$int::Int][rank](), out_type], out_chain: OutputChainPtr)
```

Finds the indices of the minimum element along the specified axis.

Parameters:

- **type** (DType): Type of the input tensor.
- **out_type** (DType): Type of the output tensor.
- **axis_type** (DType): Type of the axis tensor.
- **rank** (Int): The rank of the input / output.

Args:

- **input** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), type]): The input tensor.
- **axis_buf** (NDBuffer[1, create_unknown[\$builtin::\$int::Int][1](), axis_type]): The axis tensor.
- **output** (NDBuffer[rank, create_unknown[\$builtin::\$int::Int][rank](), out_type]): The axis tensor.
- **out_chain** (OutputChainPtr): The chain to attach results to.

reduce_shape

```
reduce_shape[input_rank: Int, input_type: DType, axis_type: DType, single_thread_blocking_override:
Bool](input_buf: NDBuffer[input_rank, create_unknown[$builtin::$int::Int][input_rank](),
input_type], axis_buf: NDBuffer[1, create_unknown[$builtin::$int::Int][1](), axis_type]) ->
StaticIntTuple[input_rank]
```

Compute the output shape of a pad operation, and assert the inputs are compatible.

Parameters:

- **input_rank** (Int): Input_rank of the input tensor.
- **input_type** (DType): Type of the input tensor.
- **axis_type** (DType): Type of the axis tensor.
- **single_thread_blocking_override** (Bool): Whether this function can block.

Args:

- **input_buf** (NDBuffer[input_rank, create_unknown[\$builtin::\$int::Int][input_rank](),
input_type]): The input tensor.
- **axis_buf** (NDBuffer[1, create_unknown[\$builtin::\$int::Int][1](), axis_type]): The axis tensor.

Returns:

The output shape.

cumsum

```
cumsum[size: Int, type: DType](dst: Buffer[_init_(size), type], src: Buffer[_init_(size),
type])
```

Computes the cumulative sum of all elements in a buffer. $dst[i] = src[i] + src[i-1] + \dots + src[0]$.

Parameters:

- **size** (Int): The size of the input and output buffers.
- **type** (DType): The type of the elements of the input and output buffers.

Args:

- **dst** (Buffer`[__init__(size), type]`): The buffer that stores the result of cumulative sum operation.
- **src** (Buffer`[__init__(size), type]`): The buffer of elements for which the cumulative sum is computed.

sort

Module

Implements sorting functions.

You can import these APIs from the `algorithm` package. For example:

```
from algorithm.sort import sort
```

partition

```
partition[type: AnyType, cmp_fn: fn[AnyType]($0, $0) capturing -> Bool](buff: Pointer[*"type"], k: Int, size: Int)
```

Partition the input vector inplace such that first `k` elements are the largest (or smallest if `cmp_fn` is `<=` operator) elements. The ordering of the first `k` elements is undefined.

Parameters:

- **type** (AnyType): DType of the underlying data.
- **cmp_fn** (fn[AnyType](\$0, \$0) capturing -> Bool): Comparison functor of type, type) capturing -> Bool type.

Args:

- **buff** (Pointer[*"type"]): Input buffer.
- **k** (Int): Index of the partition element.
- **size** (Int): The length of the buffer.

sort

```
sort(inout buff: Pointer[Int], len: Int)
```

Sort the vector inplace.

The function doesn't return anything, the vector is updated inplace.

Args:

- **buff** (Pointer[Int]): Input buffer.
- **len** (Int): The length of the buffer.

```
sort[type: DType](inout buff: Pointer[ SIMD[type, 1]], len: Int)
```

Sort the vector inplace.

The function doesn't return anything, the vector is updated inplace.

Parameters:

- **type** (DType): DType of the underlying data.

Args:

- **buff** (Pointer[SIMD[type, 1]]): Input buffer.
- **len** (Int): The length of the buffer.

```
sort(inout v: DynamicVector[Int])
```

Sort the vector inplace.

The function doesn't return anything, the vector is updated inplace.

Args:

- **v** (DynamicVector[Int]): Input integer vector to sort.

sort[type: DTType](inout v: DynamicVector[SIMD[type, 1]])

Sort the vector inplace.

The function doesn't return anything, the vector is updated inplace.

Parameters:

- **type** (DTType): DTType of the underlying data.

Args:

- **v** (DynamicVector[SIMD[type, 1]]): Input vector to sort.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

autotuning

Module

Implements the autotune functionality.

You can import these APIs from the `autotune` package. For example:

```
from autotune import search
```

autotune

```
autotune[T: AnyType, *Ts: AnyType](value: T, values: !pop.pack<Ts>) -> T
```

Forks compilation to evaluate each of the provided values.

Parameters:

- **T** (AnyType): The first value type.
- **Ts** (*AnyType): The types of the rest of the parameters.

Args:

- **value** (T): The first value in the pack.
- **values** (!pop.pack<Ts>): The tail values.

Returns:

The value being used in the current compilation fork.

autotune_fork

```
autotune_fork[type: AnyType, *values: **"type"]()
```

Forks compilation to evaluate each of the provided values.

Return parameters: **out**: The value being used in the current compilation fork.

Parameters:

- **type** (AnyType): The type of the parameters to be evaluated.
- **values** (**"type"): A list of parameters to be evaluated.

search

```
search[fn_type: AnyType, candidates: VariadicList[fn_type], evaluator: fn(Pointer[fn_type], Int) -> Int]()
```

Finds the best implementation among all candidates.

The function runs the search among the list of candidates using the provided evaluator. The evaluator function needs to take two inputs: a pointer to array of all candidates and the size of that array - and return an index of the best candidate.

Return parameters: The best found candidate.

Parameters:

- **fn_type** (AnyType): The signature type of the function.
- **candidates** (VariadicList[fn_type]): A list of candidates to search from.

- **evaluator** (fn(Pointer[fn_type], Int) -> Int): The evaluator function.

cost_of

cost_of[fn_type: AnyType, func: fn_type]() -> Int

Count the number of operations in a function.

This function takes a function reference and estimates the “cost” of invoking the function by counting the number of MLIR operations in the function after elaboration.

FIXME: This function should be marked @consteval or equivalent to prevent dynamic instantiation of kgen.cost_of.

Parameters:

- **fn_type** (AnyType): The signature type of the function.
- **func** (fn_type): The function to evaluate.

Returns:

The number of post-elaboration operations in the function.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

base64

Module

Provides functions for base64 encoding strings.

You can import these APIs from the `base64` package. For example:

```
from base64 import b64encode
```

b64encode

`b64encode(str: String) -> String`

Performs base64 encoding on the input string.

Args:

- **str** (`String`): The input string.

Returns:

Base64 encoding of the input string.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

benchmark

Module

Implements the Benchmark class for runtime benchmarking.

You can import these APIs from the `benchmark` package. For example:

```
from benchmark import Benchmark
```

Benchmark

A benchmark harness.

The class allows to benchmark a given function (passed as a parameter) and configure various benchmarking parameters, such as number of warmup iterations, maximum number of iterations, minimum and maximum elapsed time.

Fields:

- **num_warmup** (Int): The number of warmup iterations to perform before the main benchmark loop.
- **max_iters** (Int): The maximum number of iterations to perform during the main benchmark loop.
- **min_time_ns** (Int): The minimum time (in ns) to spend within the main benchmark loop.
- **max_time_ns** (Int): The maximum time (in ns) to spend within the main benchmark loop.

Functions:

`__init__`

```
__init__(inout self: Self, num_warmup: Int, max_iters: Int, min_time_ns: Int, max_time_ns: Int)
```

Constructs a new benchmark object.

Given a function the benchmark object will benchmark it until `min_time_ns` has elapsed and either `max_time_ns` OR `max_iters` is hit.

Args:

- **num_warmup** (Int): Number of warmup iterations to run before starting benchmarking (default 2).
- **max_iters** (Int): Max number of iterations to run (default 100_000).
- **min_time_ns** (Int): Upper bound on benchmarking time in ns (default 500ms).
- **max_time_ns** (Int): Lower bound on benchmarking time in ns (default 1s).

`__copyinit__`

```
__copyinit__(inout self: Self, existing: Self)
```

`__moveinit__`

```
__moveinit__(inout self: Self, owned existing: Self)
```

`run`

```
run(func: fn() capturing -> None)(self: Self) -> Int
```

Benchmarks the given function.

Benchmarking continues until `min_time_ns` has elapsed and either `max_time_ns` or `max_iters` is achieved.

Parameters:

- **func** (`fn() capturing -> None`): The function to benchmark.

Returns:

Average execution time of `func` in ns.

clobber_memory

```
clobber_memory()
```

Forces all pending memory writes to be flushed to memory.

This ensures that the compiler does not optimize away memory writes if it deems them to be not necessary. In effect, this operation acts a barrier to memory reads and writes.

keep

```
keep(val: Bool)
```

Provides a hint to the compiler to not optimize the variable use away.

This is useful in benchmarking to avoid the compiler not deleting the code to be benchmarked because the variable is not used in a side-effecting manner.

Args:

- **val** (`Bool`): The value to not optimize away.

```
keep(val: Int)
```

Provides a hint to the compiler to not optimize the variable use away.

This is useful in benchmarking to avoid the compiler not deleting the code to be benchmarked because the variable is not used in a side-effecting manner.

Args:

- **val** (`Int`): The value to not optimize away.

```
keep[type: DType, simd_width: Int](val: SIMD[type, simd_width])
```

Provides a hint to the compiler to not optimize the variable use away.

This is useful in benchmarking to avoid the compiler not deleting the code to be benchmarked because the variable is not used in a side-effecting manner.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **val** (`SIMD[type, simd_width]`): The value to not optimize away.

```
keep[type: DType](val: DTypePointer[type])
```

Provides a hint to the compiler to not optimize the variable use away.

This is useful in benchmarking to avoid the compiler not deleting the code to be benchmarked because the variable is not used in a side-effecting manner.

Parameters:

- **type** (DType): The type of the input.

Args:

- **val** (DTypePointer[type]): The value to not optimize away.

```
keep[type: AnyType](val: Pointer[*"type"])
```

Provides a hint to the compiler to not optimize the variable use away.

This is useful in benchmarking to avoid the compiler not deleting the code to be benchmarked because the variable is not used in a side-effecting manner.

Parameters:

- **type** (AnyType): The type of the input.

Args:

- **val** (Pointer[*"type"]): The value to not optimize away.

```
keep[type: AnyType](inout *val: "type")
```

Provides a hint to the compiler to not optimize the variable use away.

This is useful in benchmarking to avoid the compiler not deleting the code to be benchmarked because the variable is not used in a side-effecting manner.

Parameters:

- **type** (AnyType): The type of the input.

Args:

- **val** (*"type"): The value to not optimize away.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

bool

Module

Implements the Bool class.

These are Mojo built-ins, so you don't need to import them.

Bool

The primitive Bool scalar value used in Mojo.

Fields:

- **value** (scalar<bool>): The underlying storage of the boolean value.

Functions:

__init__

```
__init__(value: i1) -> Self
```

Construct a Bool value given a __mlir_type.i1 value.

Args:

- **value** (i1): The initial __mlir_type.i1 value.

Returns:

The constructed Bool value.

```
__init__[width: Int](value: SIMD[bool, width]) -> Self
```

Construct a Bool value given a SIMD value.

If there is more than a single element in the SIMD value, then value is reduced using the and operator.

Args:

- **value** (SIMD[bool, width]): The initial SIMD value.

Returns:

The constructed Bool value.

```
__init__(value: scalar<bool>) -> Self
```

__bool__

```
__bool__(self: Self) -> Self
```

Convert to Bool.

Returns:

This value.

__mlir_i1__

`__mlir_i1__(self: Self) -> i1`

Convert this Bool to `__mlir_type.i1`.

This method is a special hook used by the compiler to test boolean objects in control flow conditions. It should be implemented by `Bool` but not other general boolean convertible types (they should implement `__bool__` instead).

Returns:

The underlying value for the `Bool`.

`__invert__`

`__invert__(self: Self) -> Self`

Inverts the `Bool` value.

Returns:

True if the object is `false` and `False` otherwise.

`__eq__`

`__eq__(self: Self, rhs: Self) -> Self`

Compare this `Bool` to RHS.

Performs an equality comparison between the `Bool` value and the argument. This method gets invoked when a user uses the `==` infix operator.

Args:

- **rhs** (`Self`): The `rhs` value of the equality statement.

Returns:

True if the two values match and `False` otherwise.

`__ne__`

`__ne__(self: Self, rhs: Self) -> Self`

Compare this `Bool` to RHS.

Performs a non-equality comparison between the `Bool` value and the argument. This method gets invoked when a user uses the `!=` infix operator.

Args:

- **rhs** (`Self`): The `rhs` value of the non-equality statement.

Returns:

`False` if the two values do match and `True` otherwise.

`__and__`

`__and__(self: Self, rhs: Self) -> Self`

Compute `self & rhs`.

Bitwise and's the Bool value with the argument. This method gets invoked when a user uses the and infix operator.

Args:

- **rhs** (Self): The rhs value of the and statement.

Returns:

self & rhs.

__or__

__or__(self: Self, rhs: Self) -> Self

Compute self | rhs.

Bitwise or's the Bool value with the argument. This method gets invoked when a user uses the or infix operator.

Args:

- **rhs** (Self): The rhs value of the or statement.

Returns:

self | rhs.

__xor__

__xor__(self: Self, rhs: Self) -> Self

Compute self ^ rhs.

Bitwise Xor's the Bool value with the argument. This method gets invoked when a user uses the ^ infix operator.

Args:

- **rhs** (Self): The rhs value of the xor statement.

Returns:

self ^ rhs.

__rand__

__rand__(self: Self, value: Self) -> Self

Return value & self.

Args:

- **value** (Self): The other value.

Returns:

value & self.

__ror__

__ror__(self: Self, value: Self) -> Self

Return value | self.

Args:

- **value** (Self): The other value.

Returns:

value | self.

__rxor__

__rxor__(self: Self, value: Self) -> Self

Return value ^ self.

Args:

- **value** (Self): The other value.

Returns:

value ^ self.

builtin_list

Module

Implements the ListLiteral class.

These are Mojo built-ins, so you don't need to import them.

ListLiteral

The type of a literal heterogenous list expression.

A list consists of zero or more values, separated by commas.

Parameters:

- **Ts** (*AnyType): The type of the elements.

Fields:

- **storage** (!pop.pack<Ts>): The underlying storage for the list.

Functions:

__init__

```
__init__(args: !pop.pack<Ts>) -> Self
```

Construct the list literal from the given values.

Args:

- **args** (!pop.pack<Ts>): The init values.

Returns:

The constructed ListLiteral.

__len__

```
__len__(self: Self) -> Int
```

Get the list length.

Returns:

The length of this ListLiteral.

get

```
get[i: Int, T: AnyType](self: Self) -> T
```

Get a list element at the given index.

Parameters:

- **i** (Int): The element index.
- **T** (AnyType): The element type.

Returns:

The element at the given index.

builtin_slice

Module

Implements slice.

These are Mojo built-ins, so you don't need to import them.

slice

Represents a slice expression.

Objects of this type are generated when slice syntax [a:b:c] is used.

Fields:

- **start** (Int): The starting index of the slice.
- **end** (Int): The end index of the slice.
- **step** (Int): The step increment value of the slice.

Functions:

__init__

__init__(end: Int) -> Self

Construct slice given the end value.

Args:

- **end** (Int): The end value.

Returns:

The constructed slice.

__init__(start: Int, end: Int) -> Self

Construct slice given the start and end values.

Args:

- **start** (Int): The start value.
- **end** (Int): The end value.

Returns:

The constructed slice.

__init__[T0: AnyType, T1: AnyType, T2: AnyType](start: T0, end: T1, step: T2) -> Self

Construct slice given the start, end and step values.

Parameters:

- **T0** (AnyType): Type of the start value.
- **T1** (AnyType): Type of the end value.
- **T2** (AnyType): Type of the step value.

Args:

- **start** (T0): The start value.
- **end** (T1): The end value.
- **step** (T2): The step value.

Returns:

The constructed slice.

`__getitem__`

`__getitem__(self: Self, idx: Int) -> Int`

Get the slice index.

Args:

- **idx** (Int): The index.

Returns:

The slice index.

`__eq__`

`__eq__(self: Self, other: Self) -> Bool`

Compare this slice to the other.

Args:

- **other** (Self): The slice to compare to.

Returns:

True if start, end, and step values of this slice match the corresponding values of the other slice and False otherwise.

`__ne__`

`__ne__(self: Self, other: Self) -> Bool`

Compare this slice to the other.

Args:

- **other** (Self): The slice to compare to.

Returns:

False if start, end, and step values of this slice match the corresponding values of the other slice and True otherwise.

`__len__`

`__len__(self: Self) -> Int`

Return the length of the slice.

Returns:

The length of the slice.

constrained

Module

Implements compile time constraints.

These are Mojo built-ins, so you don't need to import them.

constrained

```
constrained[cond: Bool, msg: StringLiteral]()
```

Compile time checks that the condition is true.

The `constrained` is similar to `static_assert` in C++ and is used to introduce constraints on the enclosing function. In Mojo, the assert places a constraint on the function. The message is displayed when the assertion fails.

Parameters:

- **cond** (Bool): The bool value to assert.
- **msg** (StringLiteral): The message to display on failure.

```
constrained[cond: Bool]()
```

Compile time checks that the condition is true.

The `constrained` is similar to `static_assert` in C++ and is used to introduce constraints on the enclosing function. In Mojo, the assert places a constraint on the function.

Parameters:

- **cond** (Bool): The bool value to assert.

coroutine

Module

Implements classes and methods for coroutines.

These are Mojo built-ins, so you don't need to import them.

CoroutineContext

Represents a callback closure (fn_ptr + captures).

Coroutine

Represents a coroutine.

Coroutines can pause execution saving the state of the program (including values of local variables and the location of the next instruction to be executed). When the coroutine is resumed, execution continues from where it left off, with the saved state restored.

Parameters:

- **type** (AnyType): Type of value returned upon completion of the coroutine.

Functions:

`__init__`

```
__init__(handle: !pop.coroutine<() -> !kgen.paramref<*"type">>) -> Self
```

Construct a coroutine object from a handle.

Args:

- **handle** (!pop.coroutine<() -> !kgen.paramref<*"type">>): The init handle.

Returns:

The constructed coroutine object.

`__del__`

```
__del__(owned self: Self)
```

Destroy the coroutine object.

`__await__`

```
__await__(self: Self) -> *"type"
```

Suspends the current coroutine until the coroutine is complete.

Returns:

The coroutine promise.

`get_promise`

```
get_promise(self: Self) -> Pointer[*"type"]
```

Return the pointer to the beginning of the memory where the async function results are stored.

Returns:

The coroutine promise.

get

```
get(self: Self) -> *"type"
```

Get the value of the fulfilled coroutine promise.

Returns:

The value of the fulfilled promise.

get_ctx

```
get_ctx[ctx_type: AnyType](self: Self) -> Pointer[ctx_type]
```

Returns the pointer to the coroutine context.

Parameters:

- **ctx_type** (AnyType): The type of the coroutine context.

Returns:

The coroutine context.

__call__

```
__call__(self: Self) -> *"type"
```

Execute the coroutine synchronously.

Returns:

The coroutine promise.

RaisingCoroutine

Represents a coroutine that can raise.

Coroutines can pause execution saving the state of the program (including values of local variables and the location of the next instruction to be executed). When the coroutine is resumed, execution continues from where it left off, with the saved state restored.

Parameters:

- **type** (AnyType): Type of value returned upon completion of the coroutine.

Functions:

__init__

```
__init__(handle: !pop.coroutine<() throws -> !pop.variant<!kgen.declref<"$builtin":_"$error"::_Error>, *"type">>) -> Self
```

Construct a coroutine object from a handle.

Args:

- **handle** (!pop.coroutine<() throws ->
!pop.variant<!kgen.declref<_"\$builtin":_"\$error"::_Error>, *"type">>): The init handle.

Returns:

The constructed coroutine object.

__del__

__del__(owned self: Self)

Destroy the coroutine object.

__await__

__await__(self: Self) -> *"type"

Suspends the current coroutine until the coroutine is complete.

Returns:

The coroutine promise.

get_promise

get_promise(self: Self) -> Pointer[Variant[Error, *"type"]]

Return the pointer to the beginning of the memory where the async function results are stored.

Returns:

The coroutine promise.

get

get(self: Self) -> *"type"

Get the value of the fulfilled coroutine promise.

Returns:

The value of the fulfilled promise.

get_ctx

get_ctx[ctx_type: AnyType](self: Self) -> Pointer[ctx_type]

Returns the pointer to the coroutine context.

Parameters:

- **ctx_type** (AnyType): The type of the coroutine context.

Returns:

The coroutine context.

__call__

__call__(self: Self) -> *"type"

Execute the coroutine synchronously.

Returns:

The coroutine promise.

debug_assert

Module

Implements a debug assert.

These are Mojo built-ins, so you don't need to import them.

debug_assert

```
debug_assert(cond: Bool, msg: StringLiteral)
```

Asserts that the condition is true.

The `debug_assert` is similar to `assert` in C++. It is a no-op in release builds.

Args:

- **cond** (Bool): The bool value to assert.
- **msg** (StringLiteral): The message to display on failure.

dtype

Module

Implements the DType class.

These are Mojo built-ins, so you don't need to import them.

DType

Represents DType and provides methods for working with it.

Aliases:

- `type = dtype`
- `invalid = invalid`: Represents an invalid or unknown data type.
- `bool = bool`: Represents a boolean data type.
- `int8 = si8`: Represents a signed integer type whose bitwidth is 8.
- `uint8 = ui8`: Represents an unsigned integer type whose bitwidth is 8.
- `int16 = si16`: Represents a signed integer type whose bitwidth is 16.
- `uint16 = ui16`: Represents an unsigned integer type whose bitwidth is 16.
- `int32 = si32`: Represents a signed integer type whose bitwidth is 32.
- `uint32 = ui32`: Represents an unsigned integer type whose bitwidth is 32.
- `int64 = si64`: Represents a signed integer type whose bitwidth is 64.
- `uint64 = ui64`: Represents an unsigned integer type whose bitwidth is 64.
- `bfloat16 = bf16`: Represents a brain floating point value whose bitwidth is 16.
- `float16 = f16`: Represents an IEEE754-2008 binary16 floating point value.
- `float32 = f32`: Represents an IEEE754-2008 binary32 floating point value.
- `float64 = f64`: Represents an IEEE754-2008 binary64 floating point value.
- `index = index`: Represents an integral type whose bitwidth is the maximum integral value on the system.
- `address = address`: Represents a pointer type whose bitwidth is the same as the bitwidth of the hardware's pointer type (32-bit on 32-bit machines and 64-bit on 64-bit machines).

Fields:

- **value** (`dtype`): The underlying storage for the DType value.

Functions:

__init__

`__init__(value: dtype) -> Self`

__eq__

`__eq__(self: Self, rhs: Self) -> Bool`

Compares one DType to another for equality.

Args:

- **rhs** (Self): The DType to compare against.

Returns:

True if the DTYPES are the same and False otherwise.

`__ne__`

`__ne__(self: Self, rhs: Self) -> Bool`

Compares one DType to another for non-equality.

Args:

- **rhs** (Self): The DType to compare against.

Returns:

False if the DTYPES are the same and True otherwise.

`__str__`

`__str__(self: Self) -> StringLiteral`

Gets the name of the DType.

Returns:

The name of the dtype.

`get_value`

`get_value(self: Self) -> dtype`

Gets the associated internal kgen.dtype value.

Returns:

The kgen.dtype value.

`isa`

`isa[other: Self](self: Self) -> Bool`

Checks if this DType matches the other one, specified as a parameter.

Parameters:

- **other** (Self): The DType to compare against.

Returns:

True if the DTYPES are the same and False otherwise.

`is_bool`

`is_bool(self: Self) -> Bool`

Checks if this DType is Bool.

Returns:

True if the DType is Bool and False otherwise.

is_uint8

```
is_uint8(self: Self) -> Bool
```

Checks if this DType is UInt8.

Returns:

True if the DType is UInt8 and False otherwise.

is_int8

```
is_int8(self: Self) -> Bool
```

Checks if this DType is Int8.

Returns:

True if the DType is Int8 and False otherwise.

is_uint16

```
is_uint16(self: Self) -> Bool
```

Checks if this DType is UInt16.

Returns:

True if the DType is UInt16 and False otherwise.

is_int16

```
is_int16(self: Self) -> Bool
```

Checks if this DType is Int16.

Returns:

True if the DType is Int16 and False otherwise.

is_uint32

```
is_uint32(self: Self) -> Bool
```

Checks if this DType is UInt32.

Returns:

True if the DType is UInt32 and False otherwise.

is_int32

```
is_int32(self: Self) -> Bool
```

Checks if this DType is Int32.

Returns:

True if the DType is Int32 and False otherwise.

is_uint64

```
is_uint64(self: Self) -> Bool
```

Checks if this DType is UInt64.

Returns:

True if the DType is UInt64 and False otherwise.

is_int64

```
is_int64(self: Self) -> Bool
```

Checks if this DType is Int64.

Returns:

True if the DType is Int64 and False otherwise.

is_bfloat16

```
is_bfloat16(self: Self) -> Bool
```

Checks if this DType is BFloat16.

Returns:

True if the DType is BFloat16 and False otherwise.

is_float16

```
is_float16(self: Self) -> Bool
```

Checks if this DType is Float16.

Returns:

True if the DType is Float16 and False otherwise.

is_float32

```
is_float32(self: Self) -> Bool
```

Checks if this DType is Float32.

Returns:

True if the DType is Float32 and False otherwise.

is_float64

```
is_float64(self: Self) -> Bool
```

Checks if this DType is Float64.

Returns:

True if the DType is Float64 and False otherwise.

is_index

`is_index(self: Self) -> Bool`

Checks if this DType is Index.

Returns:

True if the DType is Index and False otherwise.

is_address

`is_address(self: Self) -> Bool`

Checks if this DType is Address.

Returns:

True if the DType is Address and False otherwise.

is_unsigned

`is_unsigned(self: Self) -> Bool`

Returns True if the type parameter is unsigned and False otherwise.

Returns:

Returns True if the input type parameter is unsigned.

is_signed

`is_signed(self: Self) -> Bool`

Returns True if the type parameter is signed and False otherwise.

Returns:

Returns True if the input type parameter is signed.

is_integral

`is_integral(self: Self) -> Bool`

Returns True if the type parameter is an integer and False otherwise.

Returns:

Returns True if the input type parameter is an integer.

is_floating_point

`is_floating_point(self: Self) -> Bool`

Returns True if the type parameter is a floating-point and False otherwise.

Returns:

Returns True if the input type parameter is a floating-point.

sizeof

`sizeof(self: Self) -> Int`

Returns the size in bytes of the current DType.

Returns:

Returns the size in bytes of the current DType and -1 if the size is unknown.

bitwidth

```
bitwidth(self: Self) -> Int
```

Returns the size in bits of the current DType.

Returns:

Returns the size in bits of the current DType and -1 if the size is unknown.

dispatch_integral

```
dispatch_integral[func: fn[DType]() capturing -> None](self: Self)
```

Dispatches an integral function corresponding to the current DType.

Constraints:

DType must be integral.

Parameters:

- **func** (fn[DType]() capturing -> None): A parametrized on dtype function to dispatch.

```
dispatch_integral[func: fn[DType]() capturing -> None](self: Self, out_chain: OutputChainPtr)
```

Dispatches an integral function corresponding to the current DType.

Constraints:

DType must be integral.

Parameters:

- **func** (fn[DType]() capturing -> None): A parametrized on dtype function to dispatch.

Args:

- **out_chain** (OutputChainPtr): The output chain used to report errors.

dispatch_floating

```
dispatch_floating[func: fn[DType]() capturing -> None](self: Self)
```

Dispatches a floating-point function corresponding to the current DType.

Constraints:

DType must be floating-point.

Parameters:

- **func** (fn[DType]() capturing -> None): A parametrized on dtype function to dispatch.

```
dispatch_floating[func: fn[DType]() capturing -> None](self: Self, out_chain: OutputChainPtr)
```

Dispatches a floating-point function corresponding to the current DType.

Constraints:

DType must be floating-point or integral.

Parameters:

- **func** (fn[DType]() capturing -> None): A parametrized on dtype function to dispatch.

Args:

- **out_chain** (OutputChainPtr): The output chain used to report errors.

dispatch_arithmetic

```
dispatch_arithmetic[func: fn[DType]() capturing -> None](self: Self)
```

Dispatches a function corresponding to the current DType.

Parameters:

- **func** (fn[DType]() capturing -> None): A parametrized on dtype function to dispatch.

```
dispatch_arithmetic[func: fn[DType]() capturing -> None](self: Self, out_chain: OutputChainPtr)
```

Dispatches a function corresponding to the current DType.

Parameters:

- **func** (fn[DType]() capturing -> None): A parametrized on dtype function to dispatch.

Args:

- **out_chain** (OutputChainPtr): The output chain used to report errors.

error

Module

Implements the Error class.

These are Mojo built-ins, so you don't need to import them.

Error

This type represents an Error.

Fields:

- **data** (DTypePointer[si8]): A pointer to the beginning of the string data being referenced.
- **loaded_length** (Int): The length of the string being referenced. Error instances conditionally own their error message. To reduce the size of the error instance we use the sign bit of the length field to store the ownership value. When loaded_length is negative it indicates ownership and a free is executed in the destructor.

Functions:

__init__

__init__() -> Self

Default constructor.

Returns:

The constructed Error object.

__init__(value: StringLiteral) -> Self

Construct an Error object with a given string literal.

Args:

- **value** (StringLiteral): The error message.

Returns:

The constructed Error object.

__init__(src: String) -> Self

Construct an Error object with a given string.

Args:

- **src** (String): The error message.

Returns:

The constructed Error object.

__init__(src: StringRef) -> Self

Construct an Error object with a given string ref.

Args:

- **src** (StringRef): The error message.

Returns:

The constructed Error object.

`__copyinit__`

```
__copyinit__(existing: Self) -> Self
```

Creates a deep copy of an existing error.

Returns:

The copy of the original error.

`__del__`

```
__del__(owned self: Self)
```

Releases memory if allocated.

`__str__`

```
__str__(self: Self) -> String
```

Converts the Error to string representation.

Returns:

A String of the error message.

`__repr__`

```
__repr__(self: Self) -> String
```

Converts the Error to printable representation.

Returns:

A printable representation of the error message.

file

Module

Implements the file based methods.

These are Mojo built-ins, so you don't need to import them.

For example, here's how to read a file:

```
var f = open("my_file.txt", "r")
print(f.read())
f.close()
```

Or use a `with` statement to close the file automatically:

```
with open("my_file.txt", "r") as f:
    print(f.read())
```

FileHandle

File handle to an opened file.

Fields:

- **handle** (DTypePointer[invalid]): The underlying pointer to the file handle.

Functions:

__init__

```
__init__(inout self: Self)
```

Default constructor.

```
__init__(inout self: Self, path: StringLiteral, mode: StringLiteral)
```

Construct the FileHandle using the file path and mode.

Args:

- **path** (StringLiteral): The file path.
- **mode** (StringLiteral): The mode to open the file in (the mode can be “r” or “w”).

```
__init__(inout self: Self, path: String, mode: String)
```

Construct the FileHandle using the file path and mode.

Args:

- **path** (String): The file path.
- **mode** (String): The mode to open the file in (the mode can be “r” or “w”).

```
__init__(inout self: Self, path: StringRef, mode: StringRef)
```

Construct the FileHandle using the file path and string.

Args:

- **path** (StringRef): The file path.
- **mode** (StringRef): The mode to open the file in (the mode can be “r” or “w”).

__moveinit__

```
__moveinit__(inout self: Self, owned existing: Self)
```

Moves constructor for the file handle.

Args:

- **existing** (Self): The existing file handle.

__takeinit__

```
__takeinit__(inout self: Self, inout existing: Self)
```

Moves constructor for the file handle.

Args:

- **existing** (Self): The existing file handle.

__del__

```
__del__(owned self: Self)
```

Closes the file handle.

close

```
close(inout self: Self)
```

Closes the file handle.

read

```
read(self: Self) -> String
```

Reads the data from the file.

Returns:

The contents of the file.

write

```
write(self: Self, data: StringLiteral)
```

Write the data to the file.

Args:

- **data** (StringLiteral): The data to write to the file.

```
write(self: Self, data: String)
```

Write the data to the file.

Args:

- **data** (String): The data to write to the file.

```
write(self: Self, data: StringRef)
```

Write the data to the file.

Args:

- **data** (StringRef): The data to write to the file.

__enter__

```
__enter__(owned self: Self) -> Self
```

The function to call when entering the context.

open

```
open(path: StringLiteral, mode: StringLiteral) -> FileHandle
```

Opens the file specified by path using the mode provided, returning a FileHandle.

Args:

- **path** (StringLiteral): The path to the file to open.
- **mode** (StringLiteral): The mode to open the file in (the mode can be “r” or “w”).

Returns:

A file handle.

```
open(path: StringRef, mode: StringRef) -> FileHandle
```

Opens the file specified by path using the mode provided, returning a FileHandle.

Args:

- **path** (StringRef): The path to the file to open.
- **mode** (StringRef): The mode to open the file in (the mode can be “r” or “w”).

Returns:

A file handle.

```
open(path: String, mode: String) -> FileHandle
```

Opens the file specified by path using the mode provided, returning a FileHandle.

Args:

- **path** (String): The path to the file to open.
- **mode** (String): The mode to open the file in.

Returns:

A file handle.

```
open(path: Path, mode: String) -> FileHandle
```

Opens the file specified by path using the mode provided, returning a FileHandle.

Args:

- **path** (Path): The path to the file to open.
- **mode** (String): The mode to open the file in (the mode can be “r” or “w”).

Returns:

A file handle.

float_literal

Module

Implements the `FloatLiteral` class.

These are Mojo built-ins, so you don't need to import them.

FloatLiteral

Mojo floating point literal type.

Aliases:

- `fp_type = scalar<f64>`

Fields:

- **value** (`scalar<f64>`): The underlying storage for the floating point value.

Functions:

`__init__`

`__init__(value: Self) -> Self`

Forwarding constructor.

Args:

- **value** (`Self`): The double value.

Returns:

The value.

`__init__(value: f64) -> Self`

Create a double value from a builtin MLIR `f64` value.

Args:

- **value** (`f64`): The underlying MLIR value.

Returns:

A double value.

`__init__(value: Int) -> Self`

Convert an integer to a double value.

Args:

- **value** (`Int`): The integer value.

Returns:

The integer value as a double.

`__init__(value: IntLiteral) -> Self`

Convert an IntLiteral to a double value.

Args:

- **value** (IntLiteral): The IntLiteral value.

Returns:

The integer value as a double.

`__init__(value: scalar<f64>) -> Self`

`__bool__`

`__bool__(self: Self) -> Bool`

A double value is true if it is non-zero.

Returns:

True if non-zero.

`__neg__`

`__neg__(self: Self) -> Self`

Return the negation of the double value.

Returns:

The negated double value.

`__lt__`

`__lt__(self: Self, rhs: Self) -> Bool`

Less than comparison.

Args:

- **rhs** (Self): The value to compare.

Returns:

True if this value is less than rhs.

`__le__`

`__le__(self: Self, rhs: Self) -> Bool`

Less than or equal to comparison.

Args:

- **rhs** (Self): The value to compare.

Returns:

True if this value is less than or equal to rhs.

`__eq__`

`__eq__(self: Self, rhs: Self) -> Bool`

Compare for equality.

Args:

- **rhs** (Self): The value to compare.

Returns:

True if they are equal.

__ne__

__ne__(self: Self, rhs: Self) -> Bool

Compare for inequality.

Args:

- **rhs** (Self): The value to compare.

Returns:

True if they are not equal.

__gt__

__gt__(self: Self, rhs: Self) -> Bool

Greater than comparison.

Args:

- **rhs** (Self): The value to compare.

Returns:

True if this value is greater than rhs.

__ge__

__ge__(self: Self, rhs: Self) -> Bool

Greater than or equal to comparison.

Args:

- **rhs** (Self): The value to compare.

Returns:

True if this value is greater than or equal to rhs.

__add__

__add__(self: Self, rhs: Self) -> Self

Add two doubles.

Args:

- **rhs** (Self): The value to add.

Returns:

The sum of the two values.

__sub__

`__sub__(self: Self, rhs: Self) -> Self`

Subtract two doubles.

Args:

- **rhs** (`Self`): The value to subtract.

Returns:

The difference of the two values.

__mul__

`__mul__(self: Self, rhs: Self) -> Self`

Multiply two doubles.

Args:

- **rhs** (`Self`): The value to multiply.

Returns:

The product of the two values.

__truediv__

`__truediv__(self: Self, rhs: Self) -> Self`

Divide two doubles.

Args:

- **rhs** (`Self`): The value to divide.

Returns:

The quotient of the two values.

__floordiv__

`__floordiv__(self: Self, rhs: Self) -> Self`

Divide two doubles and round towards negative infinity.

Args:

- **rhs** (`Self`): The value to divide.

Returns:

The quotient of the two values rounded towards negative infinity.

__mod__

`__mod__(self: Self, rhs: Self) -> Self`

Compute the remainder of dividing by a value.

Args:

- **rhs** (Self): The divisor.

Returns:

The remainder of the division operation.

__pow__

`__pow__(self: Self, rhs: Self) -> Self`

Compute the power.

Args:

- **rhs** (Self): The exponent.

Returns:

The current value raised to the exponent.

__radd__

`__radd__(self: Self, rhs: Self) -> Self`

Reversed addition operator.

Args:

- **rhs** (Self): The value to add.

Returns:

The sum of this and the given value.

__rsub__

`__rsub__(self: Self, rhs: Self) -> Self`

Reversed subtraction operator.

Args:

- **rhs** (Self): The value to subtract from.

Returns:

The result of subtracting this from the given value.

__rmul__

`__rmul__(self: Self, rhs: Self) -> Self`

Reversed multiplication operator.

Args:

- **rhs** (Self): The value to multiply.

Returns:

The product of the given number and this.

__rtruediv__

`__rtruediv__(self: Self, rhs: Self) -> Self`

Reversed division.

Args:

- **rhs** (`Self`): The value to be divided by this.

Returns:

The result of dividing the given value by this.

__rfloordiv__

`__rfloordiv__(self: Self, rhs: Self) -> Self`

Reversed floor division.

Args:

- **rhs** (`Self`): The value to be floor-divided by this.

Returns:

The result of dividing the given value by this, modulo any remainder.

__rmod__

`__rmod__(self: Self, rhs: Self) -> Self`

Reversed remainder.

Args:

- **rhs** (`Self`): The divisor.

Returns:

The remainder after dividing the given value by this.

__rpow__

`__rpow__(self: Self, rhs: Self) -> Self`

Reversed power.

Args:

- **rhs** (`Self`): The number to be raised to the power of this.

Returns:

The result of raising the given number by this value.

__iadd__

`__iadd__(inout self: Self, rhs: Self)`

In-place addition operator.

Args:

- **rhs** (Self): The value to add.

__isub__

__isub__(inout self: Self, rhs: Self)

In-place subtraction operator.

Args:

- **rhs** (Self): The value to subtract.

__imul__

__imul__(inout self: Self, rhs: Self)

In-place multiplication operator.

Args:

- **rhs** (Self): The value to multiply.

__itruediv__

__itruediv__(inout self: Self, rhs: Self)

In-place division.

Args:

- **rhs** (Self): The value to divide.

__ifloordiv__

__ifloordiv__(inout self: Self, rhs: Self)

In-place floor division.

Args:

- **rhs** (Self): The value to divide.

__imod__

__imod__(inout self: Self, rhs: Self)

In-place remainder.

Args:

- **rhs** (Self): The divisor.

__ipow__

__ipow__(inout self: Self, rhs: Self)

In-place power.

Args:

- **rhs** (Self): The exponent.

to_int

`to_int(self: Self) -> Int`

Casts to the floating point value to an Int. If there is a fractional component, then the value is truncated towards zero.

Returns:

The value as an integer.

`__int__`

`__int__(self: Self) -> Int`

Casts to the floating point value to an Int. If there is a fractional component, then the value is truncated towards zero.

Returns:

The value as an integer.

int

Module

Implements the Int class.

These are Mojo built-ins, so you don't need to import them.

Int

This type represents an integer value.

Fields:

- **value** (index): The underlying storage for the integer value.

Functions:

`__init__`

```
__init__() -> Self
```

Default constructor.

Returns:

The constructed Int object.

```
__init__(value: Self) -> Self
```

Construct Int from another Int value.

Args:

- **value** (Self): The init value.

Returns:

The constructed Int object.

```
__init__(value: index) -> Self
```

Construct Int from the given index value.

Args:

- **value** (index): The init value.

Returns:

The constructed Int object.

```
__init__(value: scalar<si16>) -> Self
```

Construct Int from the given Int16 value.

Args:

- **value** (scalar<si16>): The init value.

Returns:

The constructed Int object.

`__init__(value: scalar<si32>) -> Self`

Construct Int from the given Int32 value.

Args:

- **value** (`scalar<si32>`): The init value.

Returns:

The constructed Int object.

`__init__(value: scalar<si64>) -> Self`

Construct Int from the given Int64 value.

Args:

- **value** (`scalar<si64>`): The init value.

Returns:

The constructed Int object.

`__init__(value: scalar<index>) -> Self`

Construct Int from the given Index value.

Args:

- **value** (`scalar<index>`): The init value.

Returns:

The constructed Int object.

`__init__(value: IntLiteral) -> Self`

Construct Int from the given IntLiteral value.

Args:

- **value** (`IntLiteral`): The init value.

Returns:

The constructed Int object.

`__bool__`

`__bool__(self: Self) -> Bool`

Convert this Int to Bool.

Returns:

False Bool value if the value is equal to 0 and True otherwise.

`__neg__`

`__neg__(self: Self) -> Self`

Return -self.

Returns:

The -self value.

__pos__

`__pos__(self: Self) -> Self`

Return +self.

Returns:

The +self value.

__invert__

`__invert__(self: Self) -> Self`

Return ~self.

Returns:

The ~self value.

__lt__

`__lt__(self: Self, rhs: Self) -> Bool`

Compare this Int to the RHS using LT comparison.

Args:

- **rhs** (Self): The other Int to compare against.

Returns:

True if this Int is less-than the RHS Int and False otherwise.

__le__

`__le__(self: Self, rhs: Self) -> Bool`

Compare this Int to the RHS using LE comparison.

Args:

- **rhs** (Self): The other Int to compare against.

Returns:

True if this Int is less-or-equal than the RHS Int and False otherwise.

__eq__

`__eq__(self: Self, rhs: Self) -> Bool`

Compare this Int to the RHS using EQ comparison.

Args:

- **rhs** (Self): The other Int to compare against.

Returns:

True if this Int is equal to the RHS Int and False otherwise.

__ne__

`__ne__(self: Self, rhs: Self) -> Bool`

Compare this Int to the RHS using NE comparison.

Args:

- **rhs** (Self): The other Int to compare against.

Returns:

True if this Int is non-equal to the RHS Int and False otherwise.

__gt__

`__gt__(self: Self, rhs: Self) -> Bool`

Compare this Int to the RHS using GT comparison.

Args:

- **rhs** (Self): The other Int to compare against.

Returns:

True if this Int is greater-than the RHS Int and False otherwise.

__ge__

`__ge__(self: Self, rhs: Self) -> Bool`

Compare this Int to the RHS using GE comparison.

Args:

- **rhs** (Self): The other Int to compare against.

Returns:

True if this Int is greater-or-equal than the RHS Int and False otherwise.

__add__

`__add__(self: Self, rhs: Self) -> Self`

Return `self + rhs`.

Args:

- **rhs** (Self): The value to add.

Returns:

`self + rhs` value.

__sub__

`__sub__(self: Self, rhs: Self) -> Self`

Return `self - rhs`.

Args:

- **rhs** (`Self`): The value to subtract.

Returns:

`self - rhs` value.

__mul__

`__mul__(self: Self, rhs: Self) -> Self`

Return `self * rhs`.

Args:

- **rhs** (`Self`): The value to multiply with.

Returns:

`self * rhs` value.

__truediv__

`__truediv__(self: Self, rhs: Self) -> SIMD[f64, 1]`

Return the floating point division of `self` and `rhs`.

Args:

- **rhs** (`Self`): The value to divide on.

Returns:

`float(self)/float(rhs)` value.

__floordiv__

`__floordiv__(self: Self, rhs: Self) -> Self`

Return the division of `self` and `rhs` rounded down to the nearest integer.

Args:

- **rhs** (`Self`): The value to divide on.

Returns:

`floor(self/rhs)` value.

__mod__

`__mod__(self: Self, rhs: Self) -> Self`

Return the remainder of `self` divided by `rhs`.

Args:

- **rhs** (`Self`): The value to divide on.

Returns:

The remainder of dividing self by rhs.

__pow__

`__pow__(self: Self, rhs: Self) -> Self`

Return `pow(self, rhs)`.

Computes the power of an integer using the Russian Peasant Method.

Args:

- **rhs** (Self): The RHS value.

Returns:

The value of `pow(self, rhs)`.

__lshift__

`__lshift__(self: Self, rhs: Self) -> Self`

Return `self << rhs`.

Args:

- **rhs** (Self): The value to shift with.

Returns:

`self << rhs`.

__rshift__

`__rshift__(self: Self, rhs: Self) -> Self`

Return `self >> rhs`.

Args:

- **rhs** (Self): The value to shift with.

Returns:

`self >> rhs`.

__and__

`__and__(self: Self, rhs: Self) -> Self`

Return `self & rhs`.

Args:

- **rhs** (Self): The RHS value.

Returns:

`self & rhs`.

__or__

`__or__(self: Self, rhs: Self) -> Self`

Return self | rhs.

Args:

- **rhs** (Self): The RHS value.

Returns:

self | rhs.

__xor__

__xor__(self: Self, rhs: Self) -> Self

Return self ^ rhs.

Args:

- **rhs** (Self): The RHS value.

Returns:

self ^ rhs.

__radd__

__radd__(self: Self, value: Self) -> Self

Return value + self.

Args:

- **value** (Self): The other value.

Returns:

value + self.

__rsub__

__rsub__(self: Self, value: Self) -> Self

Return value - self.

Args:

- **value** (Self): The other value.

Returns:

value - self.

__rmul__

__rmul__(self: Self, value: Self) -> Self

Return value * self.

Args:

- **value** (Self): The other value.

Returns:

value * self.

__rfloordiv__

__rfloordiv__(self: Self, value: Self) -> Self

Return value // self.

Args:

- **value** (Self): The other value.

Returns:

value // self.

__rmod__

__rmod__(self: Self, value: Self) -> Self

Return value % self.

Args:

- **value** (Self): The other value.

Returns:

value % self.

__rpow__

__rpow__(self: Self, value: Self) -> Self

Return pow(value,self).

Args:

- **value** (Self): The other value.

Returns:

pow(value,self).

__rlshift__

__rlshift__(self: Self, value: Self) -> Self

Return value << self.

Args:

- **value** (Self): The other value.

Returns:

value << self.

__rrshift__

__rrshift__(self: Self, value: Self) -> Self

Return value >> self.

Args:

- **value** (Self): The other value.

Returns:

value >> self.

__rand__

__rand__(self: Self, value: Self) -> Self

Return value & self.

Args:

- **value** (Self): The other value.

Returns:

value & self.

__ror__

__ror__(self: Self, value: Self) -> Self

Return value | self.

Args:

- **value** (Self): The other value.

Returns:

value | self.

__rxor__

__rxor__(self: Self, value: Self) -> Self

Return value ^ self.

Args:

- **value** (Self): The other value.

Returns:

value ^ self.

__iadd__

__iadd__(inout self: Self, rhs: Self)

Compute self + rhs and save the result in self.

Args:

- **rhs** (Self): The RHS value.

__isub__

__isub__(inout self: Self, rhs: Self)

Compute `self - rhs` and save the result in `self`.

Args:

- **rhs** (`Self`): The RHS value.

__imul__

`__imul__`(`inout self: Self, rhs: Self`)

Compute `self*rhs` and save the result in `self`.

Args:

- **rhs** (`Self`): The RHS value.

__itruediv__

`__itruediv__`(`inout self: Self, rhs: Self`)

Compute `self / rhs`, convert to int, and save the result in `self`.

Since `floor(self / rhs)` is equivalent to `self // rhs`, this yields the same as `__ifloordiv__`.

Args:

- **rhs** (`Self`): The RHS value.

__ifloordiv__

`__ifloordiv__`(`inout self: Self, rhs: Self`)

Compute `self // rhs` and save the result in `self`.

Args:

- **rhs** (`Self`): The RHS value.

__imod__

`__imod__`(`inout self: Self, rhs: Self`)

Compute `self % rhs` and save the result in `self`.

Args:

- **rhs** (`Self`): The RHS value.

__ipow__

`__ipow__`(`inout self: Self, rhs: Self`)

Compute `pow(self, rhs)` and save the result in `self`.

Args:

- **rhs** (`Self`): The RHS value.

__ilshift__

`__ilshift__`(`inout self: Self, rhs: Self`)

Compute `self << rhs` and save the result in `self`.

Args:

- **rhs** (Self): The RHS value.

__irshift__

__irshift__(inout self: Self, rhs: Self)

Compute `self >> rhs` and save the result in `self`.

Args:

- **rhs** (Self): The RHS value.

__iand__

__iand__(inout self: Self, rhs: Self)

Compute `self & rhs` and save the result in `self`.

Args:

- **rhs** (Self): The RHS value.

__ixor__

__ixor__(inout self: Self, rhs: Self)

Compute `self ^ rhs` and save the result in `self`.

Args:

- **rhs** (Self): The RHS value.

__ior__

__ior__(inout self: Self, rhs: Self)

Compute `self|rhs` and save the result in `self`.

Args:

- **rhs** (Self): The RHS value.

__int__

__int__(self: Self) -> Self

Gets the integral value (this is an identity function for Int).

Returns:

The value as an integer.

__mlir_index__

__mlir_index__(self: Self) -> index

Convert to index.

Returns:

The corresponding `__mlir_type.index` value.

__index__

`__index__(self: Self) -> Self`

Return self converted to an integer, if self is suitable for use as an index into a list.

For Int type this is simply the value.

Returns:

The corresponding Int value.

io

Module

Provides utilities for working with input/output.

These are Mojo built-ins, so you don't need to import them.

put_new_line

```
put_new_line()
```

Prints a new line character.

print

```
print()
```

Prints a newline.

```
print(t: DType)
```

Prints a DType.

Args:

- **t** (DType): The DType to print.

```
print(x: String)
```

Prints a string.

Args:

- **x** (String): The string to print.

```
print(x: StringRef)
```

Prints a string.

Args:

- **x** (StringRef): The string to print.

```
print(x: StringLiteral)
```

Prints a string.

Args:

- **x** (StringLiteral): The string to print.

```
print(x: Bool)
```

Prints a boolean value.

Args:

- **x** (Bool): The value to print.

```
print(x: FloatLiteral)
```

Prints a float literal.

Args:

- **x** (FloatLiteral): The value to print.

```
print(x: Int)
```

Prints an integer value.

Args:

- **x** (Int): The value to print.

```
print[simd_width: Int, type: DType](vec: SIMD[type, simd_width])
```

Prints a SIMD value.

Parameters:

- **simd_width** (Int): The SIMD vector width.
- **type** (DType): The DType of the value.

Args:

- **vec** (SIMD[type, simd_width]): The SIMD value to print.

```
print[simd_width: Int, type: DType](vec: ComplexSIMD[type, simd_width])
```

Prints a SIMD value.

Parameters:

- **simd_width** (Int): The SIMD vector width.
- **type** (DType): The DType of the value.

Args:

- **vec** (ComplexSIMD[type, simd_width]): The complex value to print.

```
print[type: DType](x: Atomic[type])
```

Prints an atomic value.

Parameters:

- **type** (DType): The DType of the atomic value.

Args:

- **x** (Atomic[type]): The value to print.

```
print[length: Int](shape: DimList)
```

Prints a DimList object.

Parameters:

- **length** (Int): The length of the DimList.

Args:

- **shape** (DimList): The DimList object to print.

```
print(obj: object)
```

Prints an object type.

Args:

- **obj** (object): The object to print.

```
print(err: Error)
```

Prints an Error type.

Args:

- **err** (Error): The Error to print.

```
print(*elements: _Printable)
```

Prints a sequence of elements, joined by spaces, followed by a newline.

Args:

- **elements** (*_Printable): The elements to print.

print_no_newline

```
print_no_newline(*elements: _Printable)
```

Prints a sequence of elements, joined by spaces.

Args:

- **elements** (*_Printable): The elements to print.

len

Module

Provides the `len` function.

These are Mojo built-ins, so you don't need to import them.

len

```
len[type: DType, size: Int](value: SIMD[type, size]) -> Int
```

Returns the length of the given SIMD value.

Parameters:

- **type** (`DType`): The element type of the SIMD value.
- **size** (`Int`): The number of elements in the value.

Args:

- **value** (`SIMD[type, size]`): The SIMD value.

Returns:

The length of the SIMD value.

```
len[size: Int, type: AnyType](value: InlinedFixedVector[size, *"type"]) -> Int
```

Returns the length of the given fixed vector.

Parameters:

- **size** (`Int`): The number of elements in the value.
- **type** (`AnyType`): The element type of the value.

Args:

- **value** (`InlinedFixedVector[size, *"type"]`): The vector value.

Returns:

The length of the vector.

```
len[type: AnyType](value: UnsafeFixedVector[*"type"]) -> Int
```

Return the length of the given vector.

Parameters:

- **type** (`AnyType`): The element type of the value.

Args:

- **value** (`UnsafeFixedVector[*"type"]`): The vector value.

Returns:

The length of the vector.

```
len[type: AnyType](value: DynamicVector[*"type"]) -> Int
```

Return the length of the given vector.

Parameters:

- **type** (AnyType): The element type of the value.

Args:

- **value** (DynamicVector[*"type"]): The vector value.

Returns:

The length of the vector.

```
len(value: String) -> Int
```

Return the length of the given string.

Args:

- **value** (String): The string value.

Returns:

The length of the string.

```
len(value: StringRef) -> Int
```

Return the length of the given string reference.

Args:

- **value** (StringRef): The string value.

Returns:

The length of the string.

```
len(value: StringLiteral) -> Int
```

Return the length of the given string literal.

Args:

- **value** (StringLiteral): The string literal value.

Returns:

The length of the string literal.

```
len[size: Dim, type: DType](value: Buffer[size, type]) -> Int
```

Return the length of the given buffer.

Parameters:

- **size** (Dim): The number of elements in the value.
- **type** (DType): The element type of the value.

Args:

- **value** (Buffer[size, type]): The buffer value.

Returns:

The length of the buffer.

```
len[type: AnyType](value: VariadicList[*"type"]) -> Int
```

Return the length of the given variadic list.

Parameters:

- **type** (AnyType): The element type of the value.

Args:

- **value** (VariadicList[*"type"]): The variadic list value.

Returns:

The length of the variadic list.

```
len[type: AnyType](value: VariadicListMem[*"type"]) -> Int
```

Return the length of the given variadic list.

Parameters:

- **type** (AnyType): The element type of the value.

Args:

- **value** (VariadicListMem[*"type"]): The variadic list value.

Returns:

The length of the variadic list.

```
len[*types: AnyType](value: ListLiteral[types]) -> Int
```

Return the length of the given list literal.

Parameters:

- **types** (*AnyType): The element types of the value.

Args:

- **value** (ListLiteral[types]): The list literal value.

Returns:

The length of the list literal.

```
len[*types: AnyType](value: Tuple[types]) -> Int
```

Return the length of the given tuple literal.

Parameters:

- **types** (*AnyType): The element types of the value.

Args:

- **value** (Tuple[types]): The tuple literal value.

Returns:

The length of the tuple literal.

len[size: Int](value: StaticIntTuple[size]) -> Int

Return the length of the given static tuple.

Parameters:

- **size** (Int): The number of elements in the value.

Args:

- **value** (StaticIntTuple[size]): The static tuple value.

Returns:

The length of the static tuple.

object

Module

Defines the object type, which is used to represent untyped values.

These are Mojo built-ins, so you don't need to import them.

Attr

A generic object's attributes are set on construction, after which the attributes can be read and modified, but no attributes may be removed or added.

Fields:

- **key** (StringLiteral): The name of the attribute.
- **value** (object): The value of the attribute.

Functions:

__init__

```
__init__(inout self: Self, key: StringLiteral, owned value: object)
```

Initializes the attribute with a key and value.

Args:

- **key** (StringLiteral): The string literal key.
- **value** (object): The object value of the attribute.

__del__

```
__del__(owned self: Self)
```

object

Represents an object without a concrete type.

This is the type of arguments in def functions that do not have a type annotation, such as the type of x in def f(x): pass. A value of any type can be passed in as the x argument in this case, and so that value is used to construct this object type.

Aliases:

- nullary_function = fn() raises -> object: Nullary function type.
- unary_function = fn(object) raises -> object: Unary function type.
- binary_function = fn(object, object) raises -> object: Binary function type.
- ternary_function = fn(object, object, object) raises -> object: Ternary function type.

Functions:

__init__

```
__init__(inout self: Self)
```

Initializes the object with a None value.

```
__init__(inout self: Self, impl: _ObjectImpl)
```

Initializes the object with an implementation value. This is meant for internal use only.

Args:

- **impl** (_ObjectImpl): The object implementation.

```
__init__(inout self: Self, none: None)
```

Initializes a none value object from a None literal.

Args:

- **none** (None): None.

```
__init__(inout self: Self, value: Int)
```

Initializes the object with an integer value.

Args:

- **value** (Int): The integer value.

```
__init__(inout self: Self, value: FloatLiteral)
```

Initializes the object with an floating-point value.

Args:

- **value** (FloatLiteral): The float value.

```
__init__[dt: DType](inout self: Self, value: SIMD[dt, 1])
```

Initializes the object with a generic scalar value. If the scalar value type is bool, it is converted to a boolean. Otherwise, it is converted to the appropriate integer or floating point type.

Parameters:

- **dt** (DType): The scalar value type.

Args:

- **value** (SIMD[dt, 1]): The scalar value.

```
__init__(inout self: Self, value: Bool)
```

Initializes the object from a bool.

Args:

- **value** (Bool): The boolean value.

```
__init__(inout self: Self, value: StringLiteral)
```

Initializes the object from a string literal.

Args:

- **value** (StringLiteral): The string value.

```
__init__(inout self: Self, value: StringRef)
```

Initializes the object from a string reference.

Args:

- **value** (StringRef): The string value.

```
__init__(*Ts: AnyType)(inout self: Self, value: ListLiteral[Ts])
```

Initializes the object from a list literal.

Parameters:

- **Ts** (*AnyType): The list element types.

Args:

- **value** (ListLiteral[Ts]): The list value.

```
__init__(inout self: Self, func: fn() raises -> object)
```

Initializes an object from a function that takes no arguments.

Args:

- **func** (fn() raises -> object): The function.

```
__init__(inout self: Self, func: fn(object) raises -> object)
```

Initializes an object from a function that takes one argument.

Args:

- **func** (fn(object) raises -> object): The function.

```
__init__(inout self: Self, func: fn(object, object) raises -> object)
```

Initializes an object from a function that takes two arguments.

Args:

- **func** (fn(object, object) raises -> object): The function.

```
__init__(inout self: Self, func: fn(object, object, object) raises -> object)
```

Initializes an object from a function that takes three arguments.

Args:

- **func** (fn(object, object, object) raises -> object): The function.

```
__init__(inout self: Self, *attrs: Attr)
```

Initializes the object with a sequence of zero or more attributes.

Args:

- **attrs** (*Attr): Zero or more attributes.

```
__copyinit__
```

```
__copyinit__(inout self: Self, existing: Self)
```

Copies the object. This clones the underlying string value and increases the refcount of lists or dictionaries.

Args:

- **existing** (Self): The object to copy.

__moveinit__

```
__moveinit__(inout self: Self, owned existing: Self)
```

Move the value of an object.

Args:

- **existing** (Self): The object to move.

__del__

```
__del__(owned self: Self)
```

Delete the object and release any owned memory.

__bool__

```
__bool__(self: Self) -> Bool
```

Performs conversion to bool according to Python semantics. Integers and floats are true if they are non-zero, and strings and lists are true if they are non-empty.

Returns:

Whether the object is considered true.

__getitem__

```
__getitem__(self: Self, i: Self) -> Self
```

Gets the i-th item from the object. This is only valid for strings, lists, and dictionaries.

Args:

- **i** (Self): The string or list index, or dictionary key.

Returns:

The value at the index or key.

```
__getitem__(self: Self, *i: Self) -> Self
```

Gets the i-th item from the object, where i is a tuple of indices.

Args:

- **i (*Self)**: A compound index.

Returns:

The value at the index.

__setitem__

```
__setitem__(self: Self, i: Self, value: Self)
```

Sets the i-th item in the object. This is only valid for strings, lists, and dictionaries.

Args:

- **i** (`Self`): The string or list index, or dictionary key.
- **value** (`Self`): The value to set.

`__setitem__(self: Self, i: Self, j: Self, value: Self)`

Sets the (i, j) -th element in the object.

FIXME: We need this because `obj[i, j] = value` will attempt to invoke this method with 3 arguments, and we can only have variadics as the last argument.

Args:

- **i** (`Self`): The first index.
- **j** (`Self`): The second index.
- **value** (`Self`): The value to set.

`__neg__`

`__neg__(self: Self) -> Self`

Negation operator. Only valid for bool, int, and float types. Negation on any bool value converts it to an integer.

Returns:

The negative of the current value.

`__invert__`

`__invert__(self: Self) -> Self`

Invert value operator. This is only valid for bool and int values.

Returns:

The inverted value.

`__lt__`

`__lt__(self: Self, rhs: Self) -> Self`

Less-than comparator. This lexicographically compares strings and lists.

Args:

- **rhs** (`Self`): Right hand value.

Returns:

True if the object is less than the right hand argument.

`__le__`

`__le__(self: Self, rhs: Self) -> Self`

Less-than-or-equal to comparator. This lexicographically compares strings and lists.

Args:

- **rhs** (`Self`): Right hand value.

Returns:

True if the object is less than or equal to the right hand argument.

__eq__

__eq__(self: Self, rhs: Self) -> Self

Equality comparator. This compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the objects are equal.

__ne__

__ne__(self: Self, rhs: Self) -> Self

Inequality comparator. This compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the objects are not equal.

__gt__

__gt__(self: Self, rhs: Self) -> Self

Greater-than comparator. This lexicographically compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the left hand value is greater.

__ge__

__ge__(self: Self, rhs: Self) -> Self

Greater-than-or-equal-to comparator. This lexicographically compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the left hand value is greater than or equal to the right hand value.

__add__

`__add__(self: Self, rhs: Self) -> Self`

Addition and concatenation operator. For arithmetic types, this function will compute the sum of the left and right hand values. For strings and lists, this function will concat the objects.

Args:

- **rhs** (`Self`): Right hand value.

Returns:

The sum or concatenated values.

`__sub__`

`__sub__(self: Self, rhs: Self) -> Self`

Subtraction operator. Valid only for arithmetic types.

Args:

- **rhs** (`Self`): Right hand value.

Returns:

The difference.

`__mul__`

`__mul__(self: Self, rhs: Self) -> Self`

Multiplication operator. Valid only for arithmetic types.

Args:

- **rhs** (`Self`): Right hand value.

Returns:

The product.

`__and__`

`__and__(self: Self, rhs: Self) -> Self`

Bool AND operator. If the left hand value is False, return the left-hand value.

Args:

- **rhs** (`Self`): Right hand value.

Returns:

The current value if it is False.

`__or__`

`__or__(self: Self, rhs: Self) -> Self`

Bool OR operator. If the left hand value is True, return the left-hand value.

Args:

- **rhs** (Self): Right hand value.

Returns:

The current value if it is True.

__radd__

__radd__(self: Self, lhs: Self) -> Self

Reverse addition or concatenation operator.

Args:

- **Lhs** (Self): Left hand value.

Returns:

The sum or concatenated value.

__rsub__

__rsub__(self: Self, lhs: Self) -> Self

Reverse subtraction operator.

Args:

- **Lhs** (Self): Left hand value.

Returns:

The result of subtracting this from the left-hand-side value.

__rmul__

__rmul__(self: Self, lhs: Self) -> Self

Reverse multiplication operator.

Args:

- **Lhs** (Self): Left hand value.

Returns:

The product.

__rand__

__rand__(self: Self, lhs: Self) -> Self

Reverse AND operator.

Args:

- **Lhs** (Self): Left hand value.

Returns:

The bitwise AND of the left-hand-side value and this.

__ror__

`__ror__(self: Self, lhs: Self) -> Self`

Reverse OR operator.

Args:

- **lhs** (`Self`): Left hand value.

Returns:

The bitwise OR of the left-hand-side value and this.

`__iadd__`

`__iadd__(inout self: Self, rhs: Self)`

In-place addition or concatenation operator.

Args:

- **rhs** (`Self`): Right hand value.

`__isub__`

`__isub__(inout self: Self, rhs: Self)`

In-place subtraction operator.

Args:

- **rhs** (`Self`): Right hand value.

`__imul__`

`__imul__(inout self: Self, rhs: Self)`

In-place multiplication operator.

Args:

- **rhs** (`Self`): Right hand value.

`__iand__`

`__iand__(inout self: Self, rhs: Self)`

In-place AND operator.

Args:

- **rhs** (`Self`): Right hand value.

`__ior__`

`__ior__(inout self: Self, rhs: Self)`

In-place OR operator.

Args:

- **rhs** (`Self`): Right hand value.

`append`

```
append(self: Self, value: Self)
```

Appends a value to the list.

Args:

- **value** (Self): The value to append.

__len__

```
__len__(self: Self) -> Int
```

Returns the “length” of the object. Only strings, lists, and dictionaries have lengths.

>Returns:

The length of the string value or the number of elements in the list or dictionary value.

__getattr__

```
__getattr__(self: Self, key: StringLiteral) -> Self
```

__setattr__

```
__setattr__(inout self: Self, key: StringLiteral, value: Self)
```

__call__

```
__call__(self: Self) -> Self
```

```
__call__(self: Self, arg0: Self) -> Self
```

```
__call__(self: Self, arg0: Self, arg1: Self) -> Self
```

```
__call__(self: Self, arg0: Self, arg1: Self, arg2: Self) -> Self
```

print

```
print(self: Self)
```

Prints the value of the object.

range

Module

Implements a ‘range’ call.

These are Mojo built-ins, so you don’t need to import them.

range

```
range(end: Int) -> _ZeroStartingRange
```

Constructs a [0; end) Range.

Args:

- **end** (Int): The end of the range.

Returns:

The constructed range.

```
range(end: PythonObject) -> _ZeroStartingRange
```

Constructs a [0; end) Range.

Args:

- **end** (PythonObject): The end of the range.

Returns:

The constructed range.

```
range(length: object) -> _ZeroStartingRange
```

Constructs a [0; length) Range.

Args:

- **length** (object): The end of the range.

Returns:

The constructed range.

```
range(start: Int, end: Int) -> _SequentialRange
```

Constructs a [start; end) Range.

Args:

- **start** (Int): The start of the range.
- **end** (Int): The end of the range.

Returns:

The constructed range.

```
range(start: object, end: object) -> _SequentialRange
```

Constructs a [start; end) Range.

Args:

- **start** (object): The start of the range.
- **end** (object): The end of the range.

Returns:

The constructed range.

```
range(start: PythonObject, end: PythonObject) -> _SequentialRange
```

Constructs a [start; end) Range.

Args:

- **start** (PythonObject): The start of the range.
- **end** (PythonObject): The end of the range.

Returns:

The constructed range.

```
range(start: Int, end: Int, step: Int) -> _StridedRange
```

Constructs a [start; end) Range with a given step.

Args:

- **start** (Int): The start of the range.
- **end** (Int): The end of the range.
- **step** (Int): The step for the range.

Returns:

The constructed range.

```
range(start: object, end: object, step: object) -> _StridedRange
```

Constructs a [start; end) Range with a given step.

Args:

- **start** (object): The start of the range.
- **end** (object): The end of the range.
- **step** (object): The step for the range.

Returns:

The constructed range.

```
range(start: PythonObject, end: PythonObject, step: PythonObject) -> _StridedRange
```

Constructs a [start; end) Range with a given step.

Args:

- **start** (PythonObject): The start of the range.
- **end** (PythonObject): The end of the range.
- **step** (PythonObject): The step for the range.

Returns:

The constructed range.

rebind

Module

Implements type rebind.

These are Mojo built-ins, so you don't need to import them.

rebind

```
rebind[dest_type: AnyType, src_type: AnyType](val: src_type) -> dest_type
```

Statically assert that a parameter input type `src_type` resolves to the same type as a parameter result type `dest_type` after function instantiation and “rebind” the input to the result type.

This function is meant to be used in uncommon cases where a parametric type depends on the value of a constrained parameter in order to manually refine the type with the constrained parameter value.

Parameters:

- **dest_type** (AnyType): The type to rebind to.
- **src_type** (AnyType): The original type.

Args:

- **val** (src_type): The value to rebind.

Returns:

The rebound value of `dest_type`.

simd

Module

Implements SIMD struct.

These are Mojo built-ins, so you don't need to import them.

Aliases:

- `Int8 = SIMD[si8, 1]`: Represents an 8-bit signed scalar integer.
- `UInt8 = SIMD[ui8, 1]`: Represents an 8-bit unsigned scalar integer.
- `Int16 = SIMD[si16, 1]`: Represents a 16-bit signed scalar integer.
- `UInt16 = SIMD[ui16, 1]`: Represents a 16-bit unsigned scalar integer.
- `Int32 = SIMD[si32, 1]`: Represents a 32-bit signed scalar integer.
- `UInt32 = SIMD[ui32, 1]`: Represents a 32-bit unsigned scalar integer.
- `Int64 = SIMD[si64, 1]`: Represents a 64-bit signed scalar integer.
- `UInt64 = SIMD[ui64, 1]`: Represents a 64-bit unsigned scalar integer.
- `Float16 = SIMD[f16, 1]`: Represents a 16-bit floating point value.
- `Float32 = SIMD[f32, 1]`: Represents a 32-bit floating point value.
- `Float64 = SIMD[f64, 1]`: Represents a 64-bit floating point value.

SIMD

Represents a small vector that is backed by a hardware vector element.

SIMD allows a single instruction to be executed across the multiple data elements of the vector.

Constraints:

The size of the SIMD vector to be positive and a power of 2.

Parameters:

- **type** (`DTType`): The data type of SIMD vector elements.
- **size** (`Int`): The size of the SIMD vector.

Aliases:

- `element_type = _65x13_type`

Fields:

- **value** (`simd<#lit.struct.extract<:!kgen.declref<@"$builtin"::@"$int"::@Int> size, "value">, #lit.struct.extract<:!kgen.declref<@"$builtin"::@"$dtype"::@DTType> type, "value">>>`): The underlying storage for the vector.

Functions:

__init__

`__init__() -> Self`

Default initializer of the SIMD vector.

By default the SIMD vectors are initialized to all zeros.

Returns:

SIMD vector whose elements are 0.

`__init__(value: Int) -> Self`

Initializes the SIMD vector with an integer.

The integer value is splatted across all the elements of the SIMD vector.

Args:

- **value** (Int): The input value.

Returns:

SIMD vector whose elements have the specified value.

`__init__(value: Bool) -> Self`

Initializes the SIMD vector with a bool value.

The bool value is splatted across all elements of the SIMD vector.

Args:

- **value** (Bool): The bool value.

Returns:

SIMD vector whose elements have the specified value.

`__init__(value: SIMD<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$int":_Int> size, "value">, #lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType> type, "value">>) -> Self`

Initializes the SIMD vector with the underlying mlir value.

Args:

- **value** (SIMD<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$int":_Int> size, "value">, #lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype":_DType> type, "value">>): The input value.

Returns:

SIMD vector using the specified value.

`__init__(*elems: SIMD[type, 1]) -> Self`

Constructs a SIMD vector via a variadic list of elements.

If there is just one input value, then it is splatted to all elements of the SIMD vector. Otherwise, the input values are assigned to the corresponding elements of the SIMD vector.

Constraints:

The number of input values is 1 or equal to size of the SIMD vector.

Args:

- **elems** (*SIMD[type, 1]): The variadic list of elements from which the SIMD vector is constructed.

Returns:

The constructed SIMD vector.

```
__init__(value: FloatLiteral) -> Self
```

Initializes the SIMD vector with a FP64 value.

The input value is splatted (broadcast) across all the elements of the SIMD vector.

Args:

- **value** (FloatLiteral): The input value.

Returns:

A SIMD vector whose elements have the specified value.

__bool__

```
__bool__(self: Self) -> Bool
```

Converts the SIMD vector into a boolean scalar value.

Returns:

True if all the elements in the SIMD vector are non-zero and False otherwise.

__getitem__

```
__getitem__(self: Self, idx: Int) -> SIMD[type, 1]
```

Gets an element from the vector.

Args:

- **idx** (Int): The element index.

Returns:

The value at position `idx`.

__setitem__

```
__setitem__(inout self: Self, idx: Int, val: SIMD[type, 1])
```

Sets an element in the vector.

Args:

- **idx** (Int): The index to set.
- **val** (SIMD[type, 1]): The value to set.

```
__setitem__(inout self: Self, idx: Int, val:  
scalar<#lit.struct.extract<#!kgen.declref<_"$builtin":_"$dtype"::_DType> type, "value">>)
```

Sets an element in the vector.

Args:

- **idx** (Int): The index to set.

- **val** (scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype"::_DType> type, "value">>): The value to set.

__neg__

`__neg__(self: Self) -> Self`

Defines the unary - operation.

Returns:

The negation of this SIMD vector.

__pos__

`__pos__(self: Self) -> Self`

Defines the unary + operation.

Returns:

This SIMD vector.

__invert__

`__invert__(self: Self) -> Self`

Returns `~self`.

Constraints:

The element type of the SIMD vector must be boolean or integral.

Returns:

The `~self` value.

__lt__

`__lt__(self: Self, rhs: Self) -> SIMD[bool, size]`

Compares two SIMD vectors using less-than comparison.

Args:

- **rhs** (Self): The rhs of the operation.

Returns:

A new bool SIMD vector of the same size whose element at position `i` is True or False depending on the expression `self[i] < rhs[i]`.

__le__

`__le__(self: Self, rhs: Self) -> SIMD[bool, size]`

Compares two SIMD vectors using less-than-or-equal comparison.

Args:

- **rhs** (Self): The rhs of the operation.

Returns:

A new bool SIMD vector of the same size whose element at position i is True or False depending on the expression `self[i] <= rhs[i]`.

__eq__

`__eq__(self: Self, rhs: Self) -> SIMD[bool, size]`

Compares two SIMD vectors using equal-to comparison.

Args:

- **rhs** (`Self`): The rhs of the operation.

Returns:

A new bool SIMD vector of the same size whose element at position i is True or False depending on the expression `self[i] == rhs[i]`.

__ne__

`__ne__(self: Self, rhs: Self) -> SIMD[bool, size]`

Compares two SIMD vectors using not-equal comparison.

Args:

- **rhs** (`Self`): The rhs of the operation.

Returns:

A new bool SIMD vector of the same size whose element at position i is True or False depending on the expression `self[i] != rhs[i]`.

__gt__

`__gt__(self: Self, rhs: Self) -> SIMD[bool, size]`

Compares two SIMD vectors using greater-than comparison.

Args:

- **rhs** (`Self`): The rhs of the operation.

Returns:

A new bool SIMD vector of the same size whose element at position i is True or False depending on the expression `self[i] > rhs[i]`.

__ge__

`__ge__(self: Self, rhs: Self) -> SIMD[bool, size]`

Compares two SIMD vectors using greater-than-or-equal comparison.

Args:

- **rhs** (`Self`): The rhs of the operation.

Returns:

A new bool SIMD vector of the same size whose element at position i is True or False depending on the expression `self[i] >= rhs[i]`.

__add__

`__add__(self: Self, rhs: Self) -> Self`

Computes `self + rhs`.

Args:

- **rhs** (`Self`): The rhs value.

Returns:

A new vector whose element at position `i` is computed as `self[i] + rhs[i]`.

__sub__

`__sub__(self: Self, rhs: Self) -> Self`

Computes `self - rhs`.

Args:

- **rhs** (`Self`): The rhs value.

Returns:

A new vector whose element at position `i` is computed as `self[i] - rhs[i]`.

__mul__

`__mul__(self: Self, rhs: Self) -> Self`

Computes `self * rhs`.

Args:

- **rhs** (`Self`): The rhs value.

Returns:

A new vector whose element at position `i` is computed as `self[i] * rhs[i]`.

__truediv__

`__truediv__(self: Self, rhs: Self) -> Self`

Computes `self / rhs`.

Args:

- **rhs** (`Self`): The rhs value.

Returns:

A new vector whose element at position `i` is computed as `self[i] / rhs[i]`.

__floordiv__

`__floordiv__(self: Self, rhs: Self) -> Self`

Returns the division of `self` and `rhs` rounded down to the nearest integer.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **rhs** (Self): The value to divide on.

Returns:

`floor(self / rhs)` value.

__mod__

`__mod__(self: Self, rhs: Self) -> Self`

Returns the remainder of self divided by rhs.

Args:

- **rhs** (Self): The value to divide on.

Returns:

The remainder of dividing self by rhs.

__pow__

`__pow__(self: Self, rhs: Int) -> Self`

Computes the vector raised to the power of the input integer value.

Args:

- **rhs** (Int): The exponential value.

Returns:

A SIMD vector where each element is raised to the power of the specified exponential value.

`__pow__[rhs_type: DType](self: Self, rhs: SIMD[rhs_type, size]) -> Self`

Computes the vector raised to the power of the input integer value.

Parameters:

- **rhs_type** (DType): The dtype of the rhs SIMD vector.

Args:

- **rhs** (SIMD[rhs_type, size]): The exponential value.

Returns:

A SIMD vector where each element is raised to the power of the specified exponential value.

__lshift__

`__lshift__(self: Self, rhs: Self) -> Self`

Returns `self << rhs`.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **rhs** (Self): The RHS value.

Returns:

self << rhs.

__rshift__

__rshift__(self: Self, rhs: Self) -> Self

Returns self >> rhs.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **rhs** (Self): The RHS value.

Returns:

self >> rhs.

__and__

__and__(self: Self, rhs: Self) -> Self

Returns self & rhs.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **rhs** (Self): The RHS value.

Returns:

self & rhs.

__or__

__or__(self: Self, rhs: Self) -> Self

Returns self | rhs.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **rhs** (Self): The RHS value.

Returns:

self | rhs.

__xor__

`__xor__(self: Self, rhs: Self) -> Self`

Returns `self ^ rhs`.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **rhs** (`Self`): The RHS value.

Returns:

`self ^ rhs`.

`__radd__`

`__radd__(self: Self, value: Self) -> Self`

Returns `value + self`.

Args:

- **value** (`Self`): The other value.

Returns:

`value + self`.

`__rsub__`

`__rsub__(self: Self, value: Self) -> Self`

Returns `value - self`.

Args:

- **value** (`Self`): The other value.

Returns:

`value - self`.

`__rmul__`

`__rmul__(self: Self, value: Self) -> Self`

Returns `value * self`.

Args:

- **value** (`Self`): The other value.

Returns:

`value * self`.

`__rtruediv__`

`__rtruediv__(self: Self, value: Self) -> Self`

Returns `value / self`.

Args:

- **value** (Self): The other value.

Returns:

value / self.

__rfloordiv__

__rfloordiv__(self: Self, value: Int) -> Int

Returns value // self.

Args:

- **value** (Int): The other value.

Returns:

value // self.

__rmod__

__rmod__(self: Self, value: Int) -> Int

Returns value % self.

Args:

- **value** (Int): The other value.

Returns:

value % self.

__rlshift__

__rlshift__(self: Self, value: Self) -> Self

Returns value << self.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **value** (Self): The other value.

Returns:

value << self.

__rrshift__

__rrshift__(self: Self, value: Self) -> Self

Returns value >> self.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **value** (Self): The other value.

Returns:

value >> self.

__rand__

__rand__(self: Self, value: Self) -> Self

Returns value & self.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **value** (Self): The other value.

Returns:

value & self.

__ror__

__ror__(self: Self, value: Self) -> Self

Returns value | self.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **value** (Self): The other value.

Returns:

value | self.

__rxor__

__rxor__(self: Self, value: Self) -> Self

Returns value ^ self.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **value** (Self): The other value.

Returns:

value ^ self.

__iadd__

`__iadd__(inout self: Self, rhs: Self)`

Performs in-place addition.

The vector is mutated where each element at position `i` is computed as `self[i] + rhs[i]`.

Args:

- **rhs** (`Self`): The rhs of the addition operation.

`__isub__`

`__isub__(inout self: Self, rhs: Self)`

Performs in-place subtraction.

The vector is mutated where each element at position `i` is computed as `self[i] - rhs[i]`.

Args:

- **rhs** (`Self`): The rhs of the operation.

`__imul__`

`__imul__(inout self: Self, rhs: Self)`

Performs in-place multiplication.

The vector is mutated where each element at position `i` is computed as `self[i] * rhs[i]`.

Args:

- **rhs** (`Self`): The rhs of the operation.

`__itruediv__`

`__itruediv__(inout self: Self, rhs: Self)`

In-place true divide operator.

The vector is mutated where each element at position `i` is computed as `self[i] / rhs[i]`.

Args:

- **rhs** (`Self`): The rhs of the operation.

`__ifloordiv__`

`__ifloordiv__(inout self: Self, rhs: Self)`

In-place flood div operator.

The vector is mutated where each element at position `i` is computed as `self[i] // rhs[i]`.

Args:

- **rhs** (`Self`): The rhs of the operation.

`__imod__`

`__imod__(inout self: Self, rhs: Self)`

In-place mod operator.

The vector is mutated where each element at position *i* is computed as `self[i] % rhs[i]`.

Args:

- **rhs** (`Self`): The rhs of the operation.

__ipow__

`__ipow__`(`inout self: Self, rhs: Int`)

In-place pow operator.

The vector is mutated where each element at position *i* is computed as `pow(self[i], rhs)`.

Args:

- **rhs** (`Int`): The rhs of the operation.

__ilshift__

`__ilshift__`(`inout self: Self, rhs: Self`)

Computes `self << rhs` and save the result in `self`.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **rhs** (`Self`): The RHS value.

__irshift__

`__irshift__`(`inout self: Self, rhs: Self`)

Computes `self >> rhs` and save the result in `self`.

Constraints:

The element type of the SIMD vector must be integral.

Args:

- **rhs** (`Self`): The RHS value.

__iand__

`__iand__`(`inout self: Self, rhs: Self`)

Computes `self & rhs` and save the result in `self`.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **rhs** (`Self`): The RHS value.

__ixor__

`__ixor__`(`inout self: Self, rhs: Self`)

Computes `self ^ rhs` and save the result in `self`.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **rhs** (`Self`): The RHS value.

__ior__

```
__ior__(inout self: Self, rhs: Self)
```

Computes `self | rhs` and save the result in `self`.

Constraints:

The element type of the SIMD vector must be bool or integral.

Args:

- **rhs** (`Self`): The RHS value.

__len__

```
__len__(self: Self) -> Int
```

Gets the length of the SIMD vector.

Returns:

The length of the SIMD vector.

splat

```
splat(x: Bool) -> Self
```

Splats (broadcasts) the element onto the vector.

Args:

- **x** (`Bool`): The input value.

Returns:

A new SIMD vector whose elements are the same as the input value.

```
splat(x: SIMD[type, 1]) -> Self
```

Splats (broadcasts) the element onto the vector.

Args:

- **x** (`SIMD[type, 1]`): The input scalar value.

Returns:

A new SIMD vector whose elements are the same as the input value.

cast

```
cast[target: DType](self: Self) -> SIMD[target, size]
```

Casts the elements of the SIMD vector to the target element type.

Parameters:

- **target** (DType): The target DType.

Returns:

A new SIMD vector whose elements have been casted to the target element type.

`__int__`

```
__int__(self: Self) -> Int
```

Casts to the value to an Int. If there is a fractional component, then the value is truncated towards zero.

Constraints:

The size of the SIMD vector must be 1.

Returns:

The value as an integer.

`to_int`

```
to_int(self: Self) -> Int
```

Casts to the value to an Int. If there is a fractional component, then the value is truncated towards zero.

Constraints:

The size of the SIMD vector must be 1.

Returns:

The value of the single integer element in the SIMD vector.

`fma`

```
fma(self: Self, multiplier: Self, accumulator: Self) -> Self
```

Performs a fused multiply-add operation, i.e. `self*multiplier + accumulator`.

Args:

- **multiplier** (Self): The value to multiply.
- **accumulator** (Self): The value to accumulate.

Returns:

A new vector whose element at position `i` is computed as `self[i]*multiplier[i] + accumulator[i]`.

`shuffle`

```
shuffle[*mask: Int](self: Self) -> Self
```

Shuffles (also called blend) the values of the current vector with the other value using the specified mask (permutation).

Parameters:

- **mask** (*Int): The permutation to use in the shuffle.

Returns:

A new vector of length `len` where the value at position `i` is `(self)[permutation[i]]`.

```
shuffle[*mask: Int](self: Self, other: Self) -> Self
```

Shuffles (also called blend) the values of the current vector with the `other` value using the specified mask (permutation).

Parameters:

- **mask** (*Int): The permutation to use in the shuffle.

Args:

- **other** (Self): The other vector to shuffle with.

Returns:

A new vector of length `len` where the value at position `i` is `(self+other)[permutation[i]]`.

slice

```
slice[output_width: Int](self: Self, offset: Int) -> SIMD[type, output_width]
```

Returns a slice of the vector of the specified width with the given offset.

Constraints:

`output_width + offset` must not exceed the size of this SIMD vector.

Parameters:

- **output_width** (Int): The output SIMD vector size.

Args:

- **offset** (Int): The given offset for the slice.

Returns:

A new vector whose elements map to `self[offset:offset+output_width]`.

min

```
min(self: Self, other: Self) -> Self
```

Computes the elementwise minimum between the two vectors.

Args:

- **other** (Self): The other SIMD vector.

Returns:

A new SIMD vector where each element at position `i` is `min(self[i], other[i])`.

max

```
max(self: Self, other: Self) -> Self
```

Computes the elementwise maximum between the two vectors.

Args:

- **other** (Self): The other SIMD vector.

Returns:

A new SIMD vector where each element at position i is $\max(\text{self}[i], \text{other}[i])$.

reduce

```
reduce[func: fn[DType, Int](SIMD[*(0,0), *(0,1)], SIMD[*(0,0), *(0,1)]) capturing -> SIMD[*(0,0), *(0,1)]](self: Self) -> SIMD[type, 1]
```

Reduces the vector using a provided reduce operator.

Parameters:

- **func** (fn[DType, Int](SIMD[*(0,0), *(0,1)], SIMD[*(0,0), *(0,1)]) capturing -> SIMD[*(0,0), *(0,1)]): The reduce function to apply to elements in this SIMD.

Returns:

A new scalar which is the reduction of all vector elements.

reduce_max

```
reduce_max(self: Self) -> SIMD[type, 1]
```

Reduces the vector using the `max` operator.

Constraints:

The element type of the vector must be integer or FP.

Returns:

The maximum element of the vector.

reduce_min

```
reduce_min(self: Self) -> SIMD[type, 1]
```

Reduces the vector using the `min` operator.

Constraints:

The element type of the vector must be integer or FP.

Returns:

The minimum element of the vector.

reduce_add

```
reduce_add(self: Self) -> SIMD[type, 1]
```

Reduces the vector using the `add` operator.

Returns:

The sum of all vector elements.

reduce_mul

```
reduce_mul(self: Self) -> SIMD[type, 1]
```

Reduces the vector using the `mul` operator.

Constraints:

The element type of the vector must be integer or FP.

Returns:

The product of all vector elements.

reduce_and

```
reduce_and(self: Self) -> Bool
```

Reduces the boolean vector using the `and` operator.

Constraints:

The element type of the vector must be boolean.

Returns:

True if all element in the vector is True and False otherwise.

reduce_or

```
reduce_or(self: Self) -> Bool
```

Reduces the boolean vector using the `or` operator.

Constraints:

The element type of the vector must be boolean.

Returns:

True if any element in the vector is True and False otherwise.

select

```
select[result_type: DType](self: Self, true_case: SIMD[result_type, size], false_case: SIMD[result_type, size]) -> SIMD[result_type, size]
```

Selects the values of the `true_case` or the `false_case` based on the current boolean values of the SIMD vector.

Parameters:

- **result_type** (`DType`): The element type of the input and output SIMD vectors.

Args:

- **true_case** (`SIMD[result_type, size]`): The values selected if the positional value is True.
- **false_case** (`SIMD[result_type, size]`): The values selected if the positional value is False.

Returns:

A new vector of the form `[true_case[i] if elem else false_case[i] in enumerate(self)]`.

rotate_left

```
rotate_left[shift: Int](self: Self) -> Self
```

Shifts the elements of a SIMD vector to the left by `shift` elements (with wrap-around).

Constraints:

`-size <= shift < size`

Parameters:

- **shift** (Int): The number of positions by which to rotate the elements of SIMD vector to the left (with wrap-around).

Returns:

The SIMD vector rotated to the left by `shift` elements (with wrap-around).

`rotate_right`

`rotate_right[shift: Int](self: Self) -> Self`

Shifts the elements of a SIMD vector to the right by `shift` elements (with wrap-around).

Constraints:

`-size < shift <= size`

Parameters:

- **shift** (Int): The number of positions by which to rotate the elements of SIMD vector to the right (with wrap-around).

Returns:

The SIMD vector rotated to the right by `shift` elements (with wrap-around).

`shift_left`

`shift_left[shift: Int](self: Self) -> Self`

Shifts the elements of a SIMD vector to the left by `shift` elements (no wrap-around, fill with zero).

Constraints:

`0 <= shift <= size`

Parameters:

- **shift** (Int): The number of positions by which to rotate the elements of SIMD vector to the left (no wrap-around, fill with zero).

Returns:

The SIMD vector rotated to the left by `shift` elements (no wrap-around, fill with zero).

`shift_right`

`shift_right[shift: Int](self: Self) -> Self`

Shifts the elements of a SIMD vector to the right by `shift` elements (no wrap-around, fill with zero).

Constraints:

`0 <= shift <= size`

Parameters:

- **shift** (Int): The number of positions by which to rotate the elements of SIMD vector to the right (no wrap-around, fill with zero).

Returns:

The SIMD vector rotated to the right by `shift` elements (no wrap-around, fill with zero).

string

Module

Implements basic object methods for working with strings.

These are Mojo built-ins, so you don't need to import them.

String

Represents a mutable string.

Functions:

`__init__`

```
__init__(inout self: Self)
```

Construct an empty string.

```
__init__(inout self: Self, str: StringRef)
```

Construct a string from a StringRef object.

Args:

- **str** (StringRef): The StringRef from which to construct this string object.

```
__init__(inout self: Self, str: StringLiteral)
```

Constructs a String value given a constant string.

Args:

- **str** (StringLiteral): The input constant string.

```
__init__(inout self: Self, val: Bool)
```

Constructs a string representing an bool value.

Args:

- **val** (Bool): The boolean value.

```
__init__(inout self: Self, num: Int)
```

Constructs a string representing an integer value.

Args:

- **num** (Int): The integer value.

```
__init__(inout self: Self, num: FloatLiteral)
```

Constructs a string representing a float value.

Args:

- **num** (FloatLiteral): The float value.

```
__init__[type: DType, simd_width: Int](inout self: Self, vec: SIMD[type, simd_width])
```

Constructs a string for a given SIMD value.

Parameters:

- **type** (DType): The dtype of the SIMD value.
- **simd_width** (Int): The width of the SIMD value.

Args:

- **vec** (SIMD[type, simd_width]): The SIMD value.

`__init__(type: DType, simd_width: Int)(inout self: Self, vec: ComplexSIMD[type, simd_width])`

Constructs a string for a given complex value.

Parameters:

- **type** (DType): The dtype of the SIMD value.
- **simd_width** (Int): The width of the SIMD value.

Args:

- **vec** (ComplexSIMD[type, simd_width]): The complex value.

`__init__(size: Int)(inout self: Self, tuple: StaticIntTuple[size])`

Constructs a string from a given StaticIntTuple.

Parameters:

- **size** (Int): The size of the tuple.

Args:

- **tuple** (StaticIntTuple[size]): The input tuple.

`__init__(inout self: Self, ptr: Pointer[SIMD[si8, 1]], len: Int)`

Creates a string from the buffer. Note that the string now owns the buffer.

Args:

- **ptr** (Pointer[SIMD[si8, 1]]): The pointer to the buffer.
- **len** (Int): The length of the buffer.

copyinit

`copyinit(inout self: Self, existing: Self)`

Creates a deep copy of an existing string.

Args:

- **existing** (Self): The string to copy.

del

`del(owned self: Self)`

Deallocates the string.

bool

`bool(self: Self) -> Bool`

Checks if the string is empty.

Returns:

True if the string is empty and False otherwise.

__getitem__

`__getitem__(self: Self, idx: Int) -> Self`

Gets the character at the specified position.

Args:

- **idx** (Int): The index value.

Returns:

A new string containing the character at the specified position.

`__getitem__(self: Self, span: slice) -> Self`

Gets the sequence of characters at the specified positions.

Args:

- **span** (slice): A slice that specifies positions of the new substring.

Returns:

A new string containing the string at the specified positions.

__eq__

`__eq__(self: Self, other: Self) -> Bool`

Compares two Strings if they have the same values.

Args:

- **other** (Self): The rhs of the operation.

Returns:

True if the Strings are equal and False otherwise.

__ne__

`__ne__(self: Self, other: Self) -> Bool`

Compares two Strings if they do not have the same values.

Args:

- **other** (Self): The rhs of the operation.

Returns:

True if the Strings are not equal and False otherwise.

__add__

`__add__(self: Self, other: Self) -> Self`

Creates a string by appending another string at the end.

Args:

- **other** (Self): The string to append.

Returns:

The new constructed string.

__radd__

```
__radd__(self: Self, other: Self) -> Self
```

Creates a string by prepending another string to the start.

Args:

- **other** (Self): The string to prepend.

Returns:

The new constructed string.

```
__radd__(self: Self, other: StringLiteral) -> Self
```

Creates a string by prepending another string to the start.

Args:

- **other** (StringLiteral): The string to prepend.

Returns:

The new constructed string.

__iadd__

```
__iadd__(inout self: Self, other: Self)
```

Appends another string to this string.

Args:

- **other** (Self): The string to append.

__len__

```
__len__(self: Self) -> Int
```

Returns the string length.

Returns:

The string length.

join

```
join[rank: Int](self: Self, elems: StaticIntTuple[rank]) -> Self
```

Joins the elements from the tuple using the current string as a delimiter.

Parameters:

- **rank** (Int): The size of the tuple.

Args:

- **elems** (StaticIntTuple[rank]): The input tuple.

Returns:

The joined string.

```
join(self: Self, *lst: Int) -> Self
```

Joins integer elements using the current string as a delimiter.

Args:

- **lst** (*Int): The input values.

Returns:

The joined string.

```
join(self: Self, *strs: Self) -> Self
```

Joins string elements using the current string as a delimiter.

Args:

- **strs** (*Self): The input values.

Returns:

The joined string.

ord

```
ord(s: String) -> Int
```

Returns an integer that represents the given one-character string.

Given a string representing one ASCII character, return an integer representing the code point of that character. For example, `ord("a")` returns the integer 97. This is the inverse of the `chr()` function.

Args:

- **s** (String): The input string, which must contain only a single character.

Returns:

An integer representing the code point of the given character.

chr

```
chr(c: Int) -> String
```

Returns a string based on the given Unicode code point.

Returns the string representing a character whose code point (which must be a positive integer between 0 and 255) is the integer `i`. For example, `chr(97)` returns the string "a". This is the inverse of the `ord()` function.

Args:

- **c** (Int): An integer between 0 and 255 that represents a code point.

Returns:

A string containing a single character based on the given code point.

atol

atol(str: String) -> Int

Parses the given string as a base-10 integer and returns that value.

For example, `atol("19")` returns 19. If the given string cannot be parsed as an integer value, an error is raised. For example, `atol("hi")` raises an error.

Args:

- **str** (String): A string to be parsed as a base-10 integer.

Returns:

An integer value that represents the string, or otherwise raises.

isdigit

isdigit(c: SIMD[si8, 1]) -> Bool

Determines whether the given character is a digit [0-9].

Args:

- **c** (SIMD[si8, 1]): The character to check.

Returns:

True if the character is a digit.

string_literal

Module

Implements the `StringLiteral` class.

These are Mojo built-ins, so you don't need to import them.

StringLiteral

This type represents a string literal.

String literals are all null-terminated for compatibility with C APIs, but this is subject to change. String literals store their length as an integer, and this does not include the null terminator.

Aliases:

- `type = string`

Fields:

- **value** (string): The underlying storage for the string literal.

Functions:

__init__

`__init__(value: string) -> Self`

Create a string literal from a builtin string type.

Args:

- **value** (string): The string value.

Returns:

A string literal object.

__bool__

`__bool__(self: Self) -> Bool`

Convert the string to a bool value.

Returns:

True if the string is not empty.

__eq__

`__eq__(self: Self, rhs: Self) -> Bool`

Compare two string literals for equality.

Args:

- **rhs** (Self): The string to compare.

Returns:

True if they are equal.

__ne__

`__ne__(self: Self, rhs: Self) -> Bool`

Compare two string literals for inequality.

Args:

- **rhs** (`Self`): The string to compare.

Returns:

True if they are not equal.

__add__

`__add__(self: Self, rhs: Self) -> Self`

Concatenate two string literals.

Args:

- **rhs** (`Self`): The string to concat.

Returns:

The concatenated string.

__len__

`__len__(self: Self) -> Int`

Get the string length.

Returns:

The length of this StringLiteral.

data

`data(self: Self) -> DTypePointer[si8]`

Get raw pointer to the underlying data.

Returns:

The raw pointer to the data.

StringRef

Module

Implements the StringRef class.

These are Mojo built-ins, so you don't need to import them.

StringRef

Represent a constant reference to a string, i.e. a sequence of characters and a length, which need not be null terminated.

Fields:

- **data** (DTypePointer[si8]): A pointer to the beginning of the string data being referenced.
- **length** (Int): The length of the string being referenced.

Functions:

__init__

```
__init__(str: StringLiteral) -> Self
```

Construct a StringRef value given a constant string.

Args:

- **str** (StringLiteral): The input constant string.

Returns:

Constructed StringRef object.

```
__init__(ptr: Pointer[SIMD[si8, 1]], len: Int) -> Self
```

Construct a StringRef value given a (potentially non-0 terminated string).

The constructor takes a raw pointer and a length.

Args:

- **ptr** (Pointer[SIMD[si8, 1]]): Pointer to the string.
- **len** (Int): The length of the string.

Returns:

Constructed StringRef object.

```
__init__(ptr: DTypePointer[si8], len: Int) -> Self
```

Construct a StringRef value given a (potentially non-0 terminated string).

The constructor takes a raw pointer and a length.

Args:

- **ptr** (DTypePointer[si8]): Pointer to the string.
- **len** (Int): The length of the string.

Returns:

Constructed StringRef object.

`__init__(ptr: Pointer[SIMD[si8, 1]]) -> Self`

Construct a StringRef value given a (potentially non-0 terminated string).

The constructor takes a raw pointer and a length.

Args:

- **ptr** (Pointer[SIMD[si8, 1]]): Pointer to the string.

Returns:

Constructed StringRef object.

`__init__(ptr: DTypePointer[si8]) -> Self`

Construct a StringRef value given a (potentially non-0 terminated string).

The constructor takes a raw pointer and a length.

Args:

- **ptr** (DTypePointer[si8]): Pointer to the string.

Returns:

Constructed StringRef object.

`__getitem__`

`__getitem__(self: Self, idx: Int) -> Self`

Get the string value at the specified position.

Args:

- **idx** (Int): The index position.

Returns:

The character at the specified position.

`__eq__`

`__eq__(self: Self, rhs: Self) -> Bool`

Compares two strings are equal.

Args:

- **rhs** (Self): The other string.

Returns:

True if the strings match and False otherwise.

`__ne__`

`__ne__(self: Self, rhs: Self) -> Bool`

Compares two strings are not equal.

Args:

- **rhs** (Self): The other string.

Returns:

True if the strings do not match and False otherwise.

__len__

```
__len__(self: Self) -> Int
```

tuple

Module

Implements the Tuple type.

These are Mojo built-ins, so you don't need to import them.

Tuple

The type of a literal tuple expression.

A tuple consists of zero or more values, separated by commas.

Parameters:

- **Ts** (*AnyType): The elements type.

Fields:

- **storage** (!pop.pack<Ts>): The underlying storage for the tuple.

Functions:

__init__

__init__(args: !pop.pack<Ts>) -> Self

Construct the tuple.

Args:

- **args** (!pop.pack<Ts>): Initial values.

Returns:

Constructed tuple.

__copyinit__

__copyinit__(existing: Self) -> Self

Copy construct the tuple.

Returns:

Constructed tuple.

__len__

__len__(self: Self) -> Int

Get the number of elements in the tuple.

Returns:

The tuple length.

get

get[i: Int, T: AnyType](self: Self) -> T

Get a tuple element.

Parameters:

- **i** (Int): The element index.
- **T** (AnyType): The element type.

Returns:

The tuple element at the requested index.

type_aliases

Module

Defines some type aliases.

These are Mojo built-ins, so you don't need to import them.

Aliases:

- `AnyType = AnyType`: Represents any Mojo data type.
- `NoneType = None`: Represents the absence of a value.
- `Lifetime = lifetime`: Value lifetime specifier.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

complex

Module

Implements the Complex type.

You can import these APIs from the `complex` package. For example:

```
from complex import ComplexSIMD
```

Aliases:

- `ComplexFloat32` = `ComplexSIMD[f32, 1]`
- `ComplexFloat64` = `ComplexSIMD[f64, 1]`

ComplexSIMD

Represents a complex SIMD value.

The class provides basic methods for manipulating complex values.

Parameters:

- **type** (`DTType`): `DTType` of the value.
- **size** (`Int`): SIMD width of the value.

Fields:

- **re** (`SIMD[type, size]`): The real part of the complex SIMD value.
- **im** (`SIMD[type, size]`): The imaginary part of the complex SIMD value.

Functions:

__init__

```
__init__(re: SIMD[type, size], im: SIMD[type, size]) -> Self
```

__neg__

```
__neg__(self: Self) -> Self
```

Negates the complex value.

Returns:

The negative of the complex value.

__add__

```
__add__(self: Self, rhs: Self) -> Self
```

Adds two complex values.

Args:

- **rhs** (`Self`): Complex value to add.

Returns:

A sum of this and RHS complex values.

__mul__

`__mul__(self: Self, rhs: Self) -> Self`

Multiplies two complex values.

Args:

- **rhs** (`Self`): Complex value to multiply with.

Returns:

A product of this and RHS complex values.

norm

`norm(self: Self) -> SIMD[type, size]`

Returns the magnitude of the complex value.

Returns:

Value of $\sqrt{re*re + im*im}$.

squared_norm

`squared_norm(self: Self) -> SIMD[type, size]`

Returns the squared magnitude of the complex value.

Returns:

Value of $re*re + im*im$.

fma

`fma(self: Self, b: Self, c: Self) -> Self`

Computes FMA operation.

Compute fused multiple-add with two other complex values: `result = self * b + c`

Args:

- **b** (`Self`): Multiplier complex value.
- **c** (`Self`): Complex value to add.

Returns:

Computed `Self * B + C` complex value.

squared_add

`squared_add(self: Self, c: Self) -> Self`

Computes Square-Add operation.

Compute `Self * Self + C`.

Args:

- **c** (`Self`): Complex value to add.

Returns:

Computed `Self * Self + c` complex value.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

bit

Module

Provides functions for bit manipulation.

You can import these APIs from the `math` package. For example:

```
from math.bit import ctlz
```

ctlz

```
ctlz(val: Int) -> Int
```

Counts the number of leading zeros of an integer.

Args:

- **val** (Int): The input value.

Returns:

The number of leading zeros of the input.

```
ctlz[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Counts the per-element number of leading zeros in a SIMD vector.

Constraints:

DType must be integral.

Parameters:

- **type** (DType): dtype used for the computation.
- **simd_width** (Int): SIMD width used for the computation.

Args:

- **val** (SIMD[type, simd_width]): The input value.

Returns:

A SIMD value where the element at position *i* contains the number of leading zeros at position *i* of the input value.

cttz

```
cttz(val: Int) -> Int
```

Counts the number of trailing zeros for an integer.

Args:

- **val** (Int): The input value.

Returns:

The number of trailing zeros of the input.

```
cttz[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Counts the number of trailing zero for a SIMD vector.

Constraints:

DType must be integral.

Parameters:

- **type** (DType): dtype used for the computation.
- **simd_width** (Int): SIMD width used for the computation.

Args:

- **val** (SIMD[type, simd_width]): The input value.

Returns:

A SIMD value where the element at position i contains the number of trailing zeros at position i of the input value.

select

```
select[type: DType, simd_width: Int](cond: SIMD[bool, simd_width], true_case: SIMD[type, simd_width], false_case: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs an elementwise select based on the input condition value.

Parameters:

- **type** (DType): dtype used for the computation.
- **simd_width** (Int): SIMD width used for the computation.

Args:

- **cond** (SIMD[bool, simd_width]): The condition.
- **true_case** (SIMD[type, simd_width]): The value to pick if the condition is True.
- **false_case** (SIMD[type, simd_width]): The value to pick if the condition is False.

Returns:

A SIMD value where the element at position i contains $\text{true_case}[i]$ if $\text{cond}[i]$ is True and $\text{false_case}[i]$ otherwise.

bitreverse

```
bitreverse[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Reverses the bitpattern of an integral value.

Constraints:

DType must be integral.

Parameters:

- **type** (DType): dtype used for the computation.
- **simd_width** (Int): SIMD width used for the computation.

Args:

- **val** (SIMD[type, simd_width]): The input value.

Returns:

A SIMD value where the element at position *i* has a reversed bitpattern of an integer value of the element at position *i* of the input value.

bswap

```
bswap[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Byte-swaps a value.

Byte swap an integer value or vector of integer values with an even number of bytes (positive multiple of 16 bits). This is equivalent to `llvm.bswap` intrinsic that has the following semantics:

The `llvm.bswap.i16` intrinsic returns an `i16` value that has the high and low byte of the input `i16` swapped. Similarly, the `llvm.bswap.i32` intrinsic returns an `i32` value that has the four bytes of the input `i32` swapped, so that if the input bytes are numbered 0, 1, 2, 3 then the returned `i32` will have its bytes in 3, 2, 1, 0 order. The `llvm.bswap.i48`, `llvm.bswap.i64` and other intrinsics extend this concept to additional even-byte lengths (6 bytes, 8 bytes and more, respectively).

Constraints:

Number of bytes must be even (`Bitwidth % 16 == 0`). `DType` must be integral.

Parameters:

- **type** (`DType`): `dtype` used for the computation.
- **simd_width** (`Int`): SIMD width used for the computation.

Args:

- **val** (`SIMD[type, simd_width]`): The input value.

Returns:

A SIMD value where the element at position *i* is the value of the element at position *i* of the input value with its bytes swapped.

ctpop

```
ctpop[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Counts the number of bits set in a value.

Constraints:

`DType` must be integral.

Parameters:

- **type** (`DType`): `dtype` used for the computation.
- **simd_width** (`Int`): SIMD width used for the computation.

Args:

- **val** (`SIMD[type, simd_width]`): The input value.

Returns:

A SIMD value where the element at position *i* contains the number of bits set in the element at position *i* of the input value.

bit_not

`bit_not[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs a bitwise NOT operation on an integral.

Constraints:

DType must be integral.

Parameters:

- **type** (`DType`): dtype used for the computation.
- **simd_width** (`Int`): SIMD width used for the computation.

Args:

- **val** (`SIMD[type, simd_width]`): The input value.

Returns:

A SIMD value where the element at position `i` is computed as a bitwise NOT of the integer value at position `i` of the input value.

bit_and

`bit_and[type: DType, simd_width: Int](a: SIMD[type, simd_width], b: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs a bitwise AND operation.

Constraints:

DType must be integral.

Parameters:

- **type** (`DType`): dtype used for the computation.
- **simd_width** (`Int`): SIMD width used for the computation.

Args:

- **a** (`SIMD[type, simd_width]`): The first input value.
- **b** (`SIMD[type, simd_width]`): The second input value.

Returns:

A SIMD value where the element at position `i` is computed as a bitwise AND of the elements at position `i` of the input values.

bit_length

`bit_length[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the number of digits required to represent the integer.

Constraints:

DType must be integral. The function asserts on non-integral dtypes in debug builds and returns 0 in release builds.

Parameters:

- **type** (DType): dtype used for the computation.
- **simd_width** (Int): SIMD width used for the computation.

Args:

- **val** (SIMD[type, simd_width]): The input value.

Returns:

A SIMD value where the element at position *i* equals to the number of digits required to represent the integer at position *i* of the input value.

limit

Module

Provides interfaces to query numeric various numeric properties of types.

You can import these APIs from the `math` package. For example:

```
from math.limit import inf
```

inf

```
inf[type: DType]() -> SIMD[type, 1]
```

Gets a +inf value for the given dtype.

Constraints:

Can only be used for FP dtypes.

Parameters:

- **type** (DType): The value dtype.

Returns:

The +inf value of the given dtype.

neginf

```
neginf[type: DType]() -> SIMD[type, 1]
```

Gets a -inf value for the given dtype.

Constraints:

Can only be used for FP dtypes.

Parameters:

- **type** (DType): The value dtype.

Returns:

The -inf value of the given dtype.

isinf

```
isinf[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Checks if the value is infinite.

This is always False for non-FP data types.

Parameters:

- **type** (DType): The value dtype.
- **simd_width** (Int): The width of the SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The value to check.

Returns:

True if val is infinite and False otherwise.

isfinite

isfinite[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[bool, simd_width]

Checks if the value is not infinite.

This is always True for non-FP data types.

Parameters:

- **type** (DType): The value dtype.
- **simd_width** (Int): The width of the SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The value to check.

Returns:

True if val is finite and False otherwise.

max_finite

max_finite[type: DType]() -> SIMD[type, 1]

Returns the maximum finite value of type.

Parameters:

- **type** (DType): The value dtype.

Returns:

The maximum representable value of the type. Does not include infinity for floating-point types.

max_or_inf

max_or_inf[type: DType]() -> SIMD[type, 1]

Returns the maximum value of type.

Parameters:

- **type** (DType): The value dtype.

Returns:

The maximum value of the type or infinity for floating-point types.

min_finite

min_finite[type: DType]() -> SIMD[type, 1]

Returns the minimum (lowest) finite value of type.

Parameters:

- **type** (DType): The value dtype.

Returns:

The minimum representable value of the type. Does not include negative infinity for floating-point types.

min_or_neginf

```
min_or_neginf[type: DType]() -> SIMD[type, 1]
```

Returns the minimum value of type.

Parameters:

- **type** (DType): The value dtype.

Returns:

The minimum value of the type or infinity for floating-point types.

math

Module

Defines math utilities.

You can import these APIs from the `math` package. For example:

```
from math import mul
```

mod

```
mod[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise modulo operation of two SIMD vectors.

Parameters:

- **type** (`DType`): `DType` of the input SIMD vectors.
- **simd_width** (`Int`): Width of the input SIMD vectors.

Args:

- **x** (`SIMD[type, simd_width]`): The numerator of the operation.
- **y** (`SIMD[type, simd_width]`): The denominator of the operation.

Returns:

The remainder of `x` divided by `y`.

mul

```
mul[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise multiplication of two SIMD vectors.

Parameters:

- **type** (`DType`): `DType` of the input SIMD vectors.
- **simd_width** (`Int`): Width of the input SIMD vectors.

Args:

- **x** (`SIMD[type, simd_width]`): First SIMD vector to multiply.
- **y** (`SIMD[type, simd_width]`): Second SIMD vector to multiply.

Returns:

Elementwise multiplication of `x` and `y`.

sub

```
sub[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise subtraction of two SIMD vectors.

Parameters:

- **type** (DType): DType of the input SIMD vectors.
- **simd_width** (Int): Width of the input SIMD vectors.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector which y will be subtracted from.
- **y** (SIMD[type, simd_width]): SIMD vector to subtract from x.

Returns:

Elementwise subtraction of SIMD vector y x - y).

add

```
add[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise addition of two SIMD vectors.

Parameters:

- **type** (DType): DType of the input SIMD vectors.
- **simd_width** (Int): Width of the input SIMD vectors.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to add.
- **y** (SIMD[type, simd_width]): Second SIMD vector to add.

Returns:

Elementwise addition of x and y.

div

```
div[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise division of two SIMD vectors.

Parameters:

- **type** (DType): DType of the input SIMD vectors.
- **simd_width** (Int): Width of the input SIMD vectors.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector containing the dividends.
- **y** (SIMD[type, simd_width]): SIMD vector containing the quotients.

Returns:

Elementwise division of SIMD vector x by SIMD vector y (this is x / y).

clamp

```
clamp[type: DType, simd_width: Int](x: SIMD[type, simd_width], lower_bound: SIMD[type, simd_width], upper_bound: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Clamps the values in a SIMD vector to be in a certain range.

Clamp cuts values in the input SIMD vector off at the upper bound and lower bound values. For example, SIMD vector [0, 1, 2, 3] clamped to a lower bound of 1 and an upper bound of 2 would return [1, 1, 2, 2].

Parameters:

- **type** (DType): DType of the input SIMD vectors.
- **simd_width** (Int): Width of the input SIMD vectors.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform the clamp operation on.
- **lower_bound** (SIMD[type, simd_width]): Minimum of the range to clamp to.
- **upper_bound** (SIMD[type, simd_width]): Maximum of the range to clamp to.

Returns:

A new SIMD vector containing x clamped to be within lower_bound and upper_bound.

abs

abs(x: Int) -> Int

Gets the absolute value of an integer.

Args:

- **x** (Int): Value to take the absolute value of.

Returns:

The absolute value of x.

abs[type: DType, simd_width: Int](x: ComplexSIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise abs (norm) on each element of the complex value.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (ComplexSIMD[type, simd_width]): The complex vector to perform absolute value on.

Returns:

The elementwise abs of x.

abs[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise absolute value on the elements of a SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform absolute value on.

Returns:

The elementwise absolute value of `x`.

`rotate_bits_left`

```
rotate_bits_left[shift: Int](x: Int) -> Int
```

Shifts the bits of a input to the left by `shift` bits (with wrap-around).

Constraints:

`-size <= shift < size`

Parameters:

- **shift** (Int): The number of bit positions by which to rotate the bits of the integer to the left (with wrap-around).

Args:

- **x** (Int): The input value.

Returns:

The input rotated to the left by `shift` elements (with wrap-around).

```
rotate_bits_left[shift: Int, type: DType, width: Int](x: SIMD[type, width]) -> SIMD[type, width]
```

Shifts bits to the left by `shift` positions (with wrap-around) for each element of a SIMD vector.

Constraints:

`0 <= shift < size` Only unsigned types can be rotated.

Parameters:

- **shift** (Int): The number of positions by which to shift left the bits for each element of a SIMD vector to the left (with wrap-around).
- **type** (DType): The `dtype` of the input and output SIMD vector.
- **width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, width]): SIMD vector to perform the operation on.

Returns:

The SIMD vector with each element's bits shifted to the left by `shift` bits (with wrap-around).

`rotate_bits_right`

```
rotate_bits_right[shift: Int](x: Int) -> Int
```

Shifts the bits of a input to the left by `shift` bits (with wrap-around).

Constraints:

`-size <= shift < size`

Parameters:

- **shift** (Int): The number of bit positions by which to rotate the bits of the integer to the left (with wrap-around).

Args:

- **x** (Int): The input value.

Returns:

The input rotated to the left by `shift` elements (with wrap-around).

```
rotate_bits_left[shift: Int, type: DType, width: Int](x: SIMD[type, width]) -> SIMD[type, width]
```

Shifts bits to the left by `shift` positions (with wrap-around) for each element of a SIMD vector.

Constraints:

`0 <= shift < size` Only unsigned types can be rotated.

Parameters:

- **shift** (Int): The number of positions by which to shift right the bits for each element of a SIMD vector to the left (with wrap-around).
- **type** (DType): The `dtype` of the input and output SIMD vector.
- **width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, width]): SIMD vector to perform the operation on.

Returns:

The SIMD vector with each element's bits shifted to the right by `shift` bits (with wrap-around).

rotate_left

```
rotate_left[shift: Int](x: Int) -> Int
```

Shifts the bits of a input to the left by `shift` bits (with wrap-around).

Constraints:

`-size <= shift < size`

Parameters:

- **shift** (Int): The number of bit positions by which to rotate the bits of the integer to the left (with wrap-around).

Args:

- **x** (Int): The input value.

Returns:

The input rotated to the left by `shift` elements (with wrap-around).

```
rotate_left[shift: Int, type: DType, size: Int](x: SIMD[type, size]) -> SIMD[type, size]
```

Shifts the elements of a SIMD vector to the left by `shift` elements (with wrap-around).

Constraints:

`-size <= shift < size`

Parameters:

- **shift** (Int): The number of positions by which to rotate the elements of SIMD vector to the left (with wrap-around).
- **type** (DType): The DType of the input and output SIMD vector.
- **size** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, size]): The input value.

Returns:

The SIMD vector rotated to the left by `shift` elements (with wrap-around).

rotate_right

```
rotate_right[shift: Int](x: Int) -> Int
```

Shifts the bits of a input to the right by `shift` bits (with wrap-around).

Constraints:

`-size <= shift < size`

Parameters:

- **shift** (Int): The number of bit positions by which to rotate the bits of the integer to the right (with wrap-around).

Args:

- **x** (Int): The input value.

Returns:

The input rotated to the right by `shift` elements (with wrap-around).

```
rotate_right[shift: Int, type: DType, size: Int](x: SIMD[type, size]) -> SIMD[type, size]
```

Shifts the elements of a SIMD vector to the right by `shift` elements (with wrap-around).

Constraints:

`-size < shift <= size`

Parameters:

- **shift** (Int): The number of positions by which to rotate the elements of SIMD vector to the right (with wrap-around).
- **type** (DType): The DType of the input and output SIMD vector.
- **size** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, size]): The input value.

Returns:

The SIMD vector rotated to the right by `shift` elements (with wrap-around).

floor

`floor[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs elementwise floor on the elements of a SIMD vector.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **x** (`SIMD[type, simd_width]`): SIMD vector to perform floor on.

Returns:

The elementwise floor of `x`.

ceil

`ceil[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs elementwise ceiling on the elements of a SIMD vector.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **x** (`SIMD[type, simd_width]`): SIMD vector to perform ceiling on.

Returns:

The elementwise ceiling of `x`.

ceildiv

`ceildiv(x: Int, y: Int) -> Int`

Return the rounded-up result of dividing `x` by `y`.

Args:

- **x** (`Int`): The numerator.
- **y** (`Int`): The denominator.

Returns:

The ceiling of dividing `x` by `y`.

trunc

`trunc[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs elementwise truncation on the elements of a SIMD vector.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform trunc on.

Returns:

The elementwise truncation of x.

round

round[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise rounding on the elements of a SIMD vector.

This rounding goes to the nearest integer with ties away from zero.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform rounding on.

Returns:

The elementwise rounding of x.

roundeven

roundeven[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise banker's rounding on the elements of a SIMD vector.

This rounding goes to the nearest integer with ties toward the nearest even integer.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform rounding on.

Returns:

The elementwise banker's rounding of x.

round_half_down

round_half_down[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Rounds ties towards the smaller integer”.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform rounding on.

Returns:

The elementwise rounding of x evaluating ties towards the smaller integer.

round_half_up

round_half_up[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise rounding of x evaluating ties towards the smaller integer.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform rounding on.

Returns:

The elementwise rounding of x evaluating ties towards the larger integer.

sqrt

sqrt(x: Int) -> Int

Performs square root on an integer.

Args:

- **x** (Int): The integer value to perform square root on.

Returns:

The square root of x.

sqrt[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise square root on the elements of a SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform square root on.

Returns:

The elementwise square root of x.

rsqrt

rsqrt[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise reciprocal square root on the elements of a SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform reciprocal square root on.

Returns:

The elementwise reciprocal square root of x.

exp2

exp2[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes elementwise 2 raised to the power of n, where n is an element of the input SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector to perform exp2 on.

Returns:

Vector containing 2^n computed elementwise, where n is an element in the input SIMD vector.

ldexp

ldexp[type: DType, simd_width: Int](x: SIMD[type, simd_width], exp: SIMD[si32, simd_width]) -> SIMD[type, simd_width]

Computes elementwise ldexp function.

The ldexp function multiplies a floating point value x by the number 2 raised to the exp power. I.e. $ldexp(x, exp)$ calculate the value of $x * 2^{exp}$ and is used within the *erf* function.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): SIMD vector of floating point values.
- **exp** (SIMD[si32, simd_width]): SIMD vector containing the exponents.

Returns:

Vector containing elementwise result of ldexp on x and exp.

exp

exp[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Calculates elementwise e^{x_i} , where x_i is an element in the input SIMD vector at position i .

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): The input SIMD vector.

Returns:

A SIMD vector containing e raised to the power x_i where x_i is an element in the input SIMD vector.

frexp

```
frexp[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> StaticTuple[2, SIMD[type, simd_width]]
```

Breaks floating point values into a fractional part and an exponent part.

Constraints:

type must be a floating point value.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): The input values.

Returns:

A tuple of two SIMD vectors containing the fractional and exponent parts of the input floating point values.

log

```
log[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise natural log (base E) of a SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): Vector to perform logarithm operation on.

Returns:

Vector containing result of performing natural log base E on x.

log2

`log2[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs elementwise log (base 2) of a SIMD vector.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **x** (`SIMD[type, simd_width]`): Vector to perform logarithm operation on.

Returns:

Vector containing result of performing log base 2 on x.

copysign

`copysign[type: DType, simd_width: Int](magnitude: SIMD[type, simd_width], sign: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Returns a value with the magnitude of the first operand and the sign of the second operand.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **magnitude** (`SIMD[type, simd_width]`): The magnitude to use.
- **sign** (`SIMD[type, simd_width]`): The sign to copy.

Returns:

Copies the sign from sign to magnitude.

erf

`erf[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs the elementwise Erf on a SIMD vector.

Parameters:

- **type** (`DType`): The `dtype` of the input and output SIMD vector.
- **simd_width** (`Int`): The width of the input and output SIMD vector.

Args:

- **x** (`SIMD[type, simd_width]`): SIMD vector to perform elementwise Erf on.

Returns:

The result of the elementwise Erf operation.

tanh

`tanh[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Performs elementwise evaluation of the tanh function.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): The vector to perform the elementwise tanh on.

Returns:

The result of the elementwise tanh operation.

isclose

isclose[type: DType, simd_width: Int](a: SIMD[type, simd_width], b: SIMD[type, simd_width], absolute_tolerance: SIMD[type, 1], relative_tolerance: SIMD[type, 1]) -> SIMD[bool, simd_width]

Checks if the two input values are numerically within a tolerance.

When the type is integral, then equality is checked. When the type is floating point, then this checks if the two input values are numerically the close using the $abs(a - b) \leq max(rtol * max(abs(a), abs(b)), atol)$ formula.

Unlike Python's `math.isclose`, this implementation is symmetric. I.e. `isclose(a, b) == isclose(b, a)`.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **a** (SIMD[type, simd_width]): The first value to compare.
- **b** (SIMD[type, simd_width]): The second value to compare.
- **absolute_tolerance** (SIMD[type, 1]): The absolute tolerance.
- **relative_tolerance** (SIMD[type, 1]): The relative tolerance.

Returns:

A boolean vector where a and b are equal within the specified tolerance.

isclose[type: DType, simd_width: Int](a: SIMD[type, simd_width], b: SIMD[type, simd_width]) -> SIMD[bool, simd_width]

Checks if the two input values are numerically within tolerance, with `atol=1e-08` and `rtol=1e-05`.

When the type is integral, then equality is checked. When the type is floating point, then this checks if the two input values are numerically the close using the $abs(a - b) \leq max(rtol * max(abs(a), abs(b)), atol)$ formula. The default absolute and relative tolerances are picked from the numpy default values.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **a** (SIMD[type, simd_width]): The first value to compare.
- **b** (SIMD[type, simd_width]): The second value to compare.

Returns:

A boolean vector where a and b are equal within tolerance, with atol=1e-08 and rtol=1e-05.

all_true

```
all_true[simd_width: Int](val: SIMD[bool, simd_width]) -> Bool
```

Returns True if all elements in the SIMD vector are True and False otherwise.

Parameters:

- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[bool, simd_width]): The SIMD vector to reduce.

Returns:

True if all values in the SIMD vector are True and False otherwise.

any_true

```
any_true[simd_width: Int](val: SIMD[bool, simd_width]) -> Bool
```

Returns True if any elements in the SIMD vector is True and False otherwise.

Parameters:

- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[bool, simd_width]): The SIMD vector to reduce.

Returns:

True if any values in the SIMD vector is True and False otherwise.

none_true

```
none_true[simd_width: Int](val: SIMD[bool, simd_width]) -> Bool
```

Returns True if all element in the SIMD vector are False and False otherwise.

Parameters:

- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[bool, simd_width]): The SIMD vector to reduce.

Returns:

True if all values in the SIMD vector are False and False otherwise.

reduce_bit_count

```
reduce_bit_count[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> Int
```

Returns a scalar containing total number of bits set in given vector.

Constraints:

The input must be either integral or boolean type.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The SIMD vector to reduce.

Returns:

Count of set bits across all elements of the vector.

iota

```
iota[type: DType, simd_width: Int]() -> SIMD[type, simd_width]
```

Creates a SIMD vector containing an increasing sequence, starting from 0.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Returns:

An increasing sequence of values, starting from 0.

```
iota[type: DType, simd_width: Int](offset: SIMD[type, 1]) -> SIMD[type, simd_width]
```

Creates a SIMD vector containing an increasing sequence, starting from offset.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **offset** (SIMD[type, 1]): The value to start the sequence at. Default is zero.

Returns:

An increasing sequence of values, starting from offset.

```
iota[type: DType](inout buff: DTypePointer[type], len: Int, offset: Int)
```

Fill the buffer with numbers ranging from offset to offset + len - 1, spaced by 1.

The function doesn't return anything, the buffer is updated inplace.

Parameters:

- **type** (DType): DType of the underlying data.

Args:

- **buff** (DTypePointer[type]): The buffer to fill.
- **len** (Int): The length of the buffer to fill.

- **offset** (Int): The value to fill at index 0.

```
iota[type: DType](inout v: DynamicVector[SIMD[type, 1]], offset: Int)
```

Fill the vector with numbers ranging from offset to offset + len - 1, spaced by 1.

The function doesn't return anything, the vector is updated inplace.

Parameters:

- **type** (DType): DType of the underlying data.

Args:

- **v** (DynamicVector[SIMD[type, 1]]): The vector to fill.
- **offset** (Int): The value to fill at index 0.

```
iota(inout v: DynamicVector[Int], offset: Int)
```

Fill the vector with numbers ranging from offset to offset + len - 1, spaced by 1.

The function doesn't return anything, the vector is updated inplace.

Args:

- **v** (DynamicVector[Int]): The vector to fill.
- **offset** (Int): The value to fill at index 0.

is_power_of_2

```
is_power_of_2[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether SIMD vector contains integer powers of two.

An element of the result SIMD vector will be True if the value is an integer power of two, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The SIMD vector to perform is_power_of_2 on.

Returns:

A SIMD vector containing True if the corresponding element in val is a power of two, otherwise False.

```
is_power_of_2(val: Int) -> Bool
```

Checks whether an integer is a power of two.

Args:

- **val** (Int): The integer to check.

Returns:

True if val is a power of two, otherwise False.

is_odd

`is_odd(val: Int) -> Bool`

Performs elementwise check of whether an integer value is odd.

Args:

- **val** (Int): The int value to check.

Returns:

True if the input is odd and False otherwise.

`is_odd[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[bool, simd_width]`

Performs elementwise check of whether SIMD vector contains odd values.

An element of the result SIMD vector will be True if the value is odd, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The SIMD vector to check.

Returns:

A SIMD vector containing True if the corresponding element in val is odd, otherwise False.

is_even

`is_even(val: Int) -> Bool`

Performs elementwise check of whether an integer value is even.

Args:

- **val** (Int): The int value to check.

Returns:

True if the input is even and False otherwise.

`is_even[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[bool, simd_width]`

Performs elementwise check of whether SIMD vector contains even values.

An element of the result SIMD vector will be True if the value is even, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The SIMD vector to check.

Returns:

A SIMD vector containing True if the corresponding element in val is even, otherwise False.

fma

fma(a: Int, b: Int, c: Int) -> Int

Performs fma (fused multiply-add) on the inputs.

The result is $(a * b) + c$.

Args:

- **a** (Int): The first input.
- **b** (Int): The second input.
- **c** (Int): The third input.

Returns:

$(a * b) + c$.

fma[type: DType, simd_width: Int](a: SIMD[type, simd_width], b: SIMD[type, simd_width], c: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Performs elementwise fma (fused multiply-add) on the inputs.

Each element in the result SIMD vector is $(A_i * B_i) + C_i$, where A_i , B_i and C_i are elements at index i in a, b, and c respectively.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **a** (SIMD[type, simd_width]): The first vector of inputs.
- **b** (SIMD[type, simd_width]): The second vector of inputs.
- **c** (SIMD[type, simd_width]): The third vector of inputs.

Returns:

Elementwise fma of a, b and c.

reciprocal

reciprocal[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Takes the elementwise reciprocal of a SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): The SIMD vector to perform elementwise reciprocal on.

Returns:

A SIMD vector the elementwise reciprocal of x.

identity

```
identity[type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Gets the identity of a SIMD vector.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): The SIMD vector to take identity of.

Returns:

Identity of x, which is x.

greater

```
greater[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether values in x are greater than values in y.

An element of the result SIMD vector will be True if the corresponding element in x is greater than the corresponding element in y, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to compare.
- **y** (SIMD[type, simd_width]): Second SIMD vector to compare.

Returns:

A SIMD vector containing True if the corresponding element in x is greater than the corresponding element in y, otherwise False.

greater_equal

```
greater_equal[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether values in x are greater than or equal to values in y.

An element of the result SIMD vector will be True if the corresponding element in x is greater than or equal to the corresponding element in y, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to compare.

- **y** (SIMD[type, simd_width]): Second SIMD vector to compare.

Returns:

A SIMD vector containing True if the corresponding element in x is greater than or equal to the corresponding element in y, otherwise False.

less

```
less[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether values in x are less than values in y.

An element of the result SIMD vector will be True if the corresponding element in x is less than the corresponding element in y, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to compare.
- **y** (SIMD[type, simd_width]): Second SIMD vector to compare.

Returns:

A SIMD vector containing True if the corresponding element in x is less than the corresponding element in y, otherwise False.

less_equal

```
less_equal[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether values in x are less than or equal to values in y.

An element of the result SIMD vector will be True if the corresponding element in x is less than or equal to the corresponding element in y, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to compare.
- **y** (SIMD[type, simd_width]): Second SIMD vector to compare.

Returns:

A SIMD vector containing True if the corresponding element in x is less than or equal to the corresponding element in y, otherwise False.

equal

```
equal[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether values in x are equal to values in y.

An element of the result SIMD vector will be True if the corresponding element in x is equal to the corresponding element in y, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to compare.
- **y** (SIMD[type, simd_width]): Second SIMD vector to compare.

Returns:

A SIMD vector containing True if the corresponding element in x is equal to the corresponding element in y, otherwise False.

logical_and

```
logical_and[simd_width: Int](x: SIMD[bool, simd_width], y: SIMD[bool, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise logical And operation.

An element of the result SIMD vector will be True if the corresponding elements in x and y are both True, and False otherwise.

Parameters:

- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[bool, simd_width]): First SIMD vector to perform the And operation.
- **y** (SIMD[bool, simd_width]): Second SIMD vector to perform the And operation.

Returns:

A SIMD vector containing True if the corresponding elements in x and y are both True, otherwise False.

logical_not

```
logical_not[simd_width: Int](x: SIMD[bool, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise logical Not operation.

An element of the result SIMD vector will be True if the corresponding element in x is True, and False otherwise.

Parameters:

- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[bool, simd_width]): SIMD vector to perform the Not operation.

Returns:

A SIMD vector containing True if the corresponding element in x is True, otherwise False.

logical_xor

```
logical_xor[simd_width: Int](x: SIMD[bool, simd_width], y: SIMD[bool, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise logical Xor operation.

An element of the result SIMD vector will be True if only one of the corresponding elements in x and y is True, and False otherwise.

Parameters:

- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[bool, simd_width]): First SIMD vector to perform the Xor operation.
- **y** (SIMD[bool, simd_width]): Second SIMD vector to perform the Xor operation.

Returns:

A SIMD vector containing True if only one of the corresponding elements in x and y is True, otherwise False.

not_equal

```
not_equal[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[bool, simd_width]
```

Performs elementwise check of whether values in x are not equal to values in y.

An element of the result SIMD vector will be True if the corresponding element in x is not equal to the corresponding element in y, and False otherwise.

Parameters:

- **type** (DType): The dtype of the input SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector to compare.
- **y** (SIMD[type, simd_width]): Second SIMD vector to compare.

Returns:

A SIMD vector containing True if the corresponding element in x is not equal to the corresponding element in y, otherwise False.

select

```
select[type: DType, simd_width: Int](cond: SIMD[bool, simd_width], true_case: SIMD[type, simd_width], false_case: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Selects the values of the `true_case` or the `false_case` based on the input boolean values of the given SIMD vector.

Parameters:

- **type** (DType): The element type of the input and output SIMD vectors.

- **simd_width** (Int): Width of the SIMD vectors we are comparing.

Args:

- **cond** (SIMD[bool, simd_width]): The vector of bools to check.
- **true_case** (SIMD[type, simd_width]): The values selected if the positional value is True.
- **false_case** (SIMD[type, simd_width]): The values selected if the positional value is False.

Returns:

A new vector of the form [true_case[i] if cond[i] else false_case[i] in enumerate(self)].

max

```
max(x: Int, y: Int) -> Int
```

Gets the maximum of two integers.

Args:

- **x** (Int): Integer input to max.
- **y** (Int): Integer input to max.

Returns:

Maximum of x and y.

```
max[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Performs elementwise maximum of x and y.

An element of the result SIMD vector will be the maximum of the corresponding elements in x and y.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector.
- **y** (SIMD[type, simd_width]): Second SIMD vector.

Returns:

A SIMD vector containing the elementwise maximum of x and y.

min

```
min(x: Int, y: Int) -> Int
```

Gets the minimum of two integers.

Args:

- **x** (Int): Integer input to max.
- **y** (Int): Integer input to max.

Returns:

Minimum of x and y.

```
min[type: DType, simd_width: Int](x: SIMD[type, simd_width], y: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Gets the elementwise minimum of x and y.

An element of the result SIMD vector will be the minimum of the corresponding elements in x and y.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **x** (SIMD[type, simd_width]): First SIMD vector.
- **y** (SIMD[type, simd_width]): Second SIMD vector.

Returns:

A SIMD vector containing the elementwise minimum of x and y.

pow

```
pow[type: DType, simd_width: Int](lhs: SIMD[type, simd_width], rhs: Int) -> SIMD[type, simd_width]
```

Computes the pow of the inputs.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **lhs** (SIMD[type, simd_width]): The first input argument.
- **rhs** (Int): The second input argument.

Returns:

The pow of the inputs.

```
pow[lhs_type: DType, rhs_type: DType, simd_width: Int](lhs: SIMD[lhs_type, simd_width], rhs: SIMD[rhs_type, simd_width]) -> SIMD[lhs_type, simd_width]
```

Computes elementwise power of a floating point type raised to another floating point type.

An element of the result SIMD vector will be the result of raising the corresponding element of lhs to the corresponding element of rhs.

Constraints:

rhs_type and lhs_type must be the same, and must be floating point types.

Parameters:

- **lhs_type** (DType): The dtype of the lhs SIMD vector.
- **rhs_type** (DType): The dtype of the rhs SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vectors.

Args:

- **lhs** (SIMD[lhs_type, simd_width]): Base of the power operation.
- **rhs** (SIMD[rhs_type, simd_width]): Exponent of the power operation.

Returns:

A SIMD vector containing elementwise lhs raised to the power of rhs.

```
pow[n: Int, type: DType, simd_width: Int](x: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the elementwise power where the exponent is an integer known at compile time.

Constraints:

n must be a signed si32 type.

Parameters:

- **n** (Int): Exponent of the power operation.
- **type** (DType): The dtype of the x SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vectors.

Args:

- **x** (SIMD[type, simd_width]): Base of the power operation.

Returns:

A SIMD vector containing elementwise x raised to the power of n.

div_ceil

```
div_ceil(numerator: Int, denominator: Int) -> Int
```

Divides an integer by another integer, and round up to the nearest integer.

Constraints:

Will raise an exception if denominator is zero.

Args:

- **numerator** (Int): The numerator.
- **denominator** (Int): The denominator.

Returns:

The ceiling of numerator divided by denominator.

align_down

```
align_down(value: Int, alignment: Int) -> Int
```

Returns the closest multiple of alignment that is less than or equal to value.

Constraints:

Will raise an exception if the alignment is zero.

Args:

- **value** (Int): The value to align.
- **alignment** (Int): Value to align to.

Returns:

Closest multiple of the alignment that is less than or equal to the input value. In other words, $\text{floor}(\text{value} / \text{alignment}) * \text{alignment}$.

align_up

```
align_up(value: Int, alignment: Int) -> Int
```

Returns the closest multiple of alignment that is greater than or equal to value.

Constraints:

Will raise an exception if the alignment is zero.

Args:

- **value** (Int): The value to align.
- **alignment** (Int): Value to align to.

Returns:

Closest multiple of the alignment that is greater than or equal to the input value. In other words, $\text{ceiling}(\text{value} / \text{alignment}) * \text{alignment}$.

acos

```
acos[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the acos of the inputs.

Constraints:

The input must be a floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The acos of the input.

asin

```
asin[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the asin of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `asin` of the input.

atan

`atan[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the `atan` of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The `dtype` of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `atan` of the input.

atan2

`atan2[type: DType, simd_width: Int](arg0: SIMD[type, simd_width], arg1: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the `atan2` of the inputs.

Constraints:

The inputs must be of floating point type.

Parameters:

- **type** (DType): The `dtype` of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg0** (SIMD[type, simd_width]): The first input argument.
- **arg1** (SIMD[type, simd_width]): The second input argument.

Returns:

The `atan2` of the inputs.

cos

`cos[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the `cos` of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The cos of the input.

sin

`sin[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the sin of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The sin of the input.

tan

`tan[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the tan of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The tan of the input.

acosh

acosh[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the acosh of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The acosh of the input.

asinh

asinh[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the asinh of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The asinh of the input.

atanh

atanh[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the atanh of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The atanh of the input.

cosh

cosh[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the cosh of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The cosh of the input.

sinh

sinh[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the sinh of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The sinh of the input.

expm1

expm1[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the expm1 of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The \exp^{-1} of the input.

log10

`log10[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the \log_{10} of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The \log_{10} of the input.

log1p

`log1p[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the \log_{10} of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The \log_{10} of the input.

logb

`logb[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the `logb` of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The `dtype` of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `logb` of the input.

cbrt

`cbrt[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the `cbrt` of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The `dtype` of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `cbrt` of the input.

hypot

`hypot[type: DType, simd_width: Int](arg0: SIMD[type, simd_width], arg1: SIMD[type, simd_width]) -> SIMD[type, simd_width]`

Computes the `hypot` of the inputs.

Constraints:

The inputs must be of floating point type.

Parameters:

- **type** (DType): The `dtype` of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg0** (SIMD[type, simd_width]): The first input argument.
- **arg1** (SIMD[type, simd_width]): The second input argument.

Returns:

The hypot of the inputs.

erfc

erfc[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the erfc of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The erfc of the input.

lgamma

lgamma[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the lgamma of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The lgamma of the input.

tgamma

tgamma[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]

Computes the tgamma of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `tgamma` of the input.

nearbyint

```
nearbyint[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the `nearbyint` of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `nearbyint` of the input.

rint

```
rint[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the `rint` of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The `rint` of the input.

remainder

```
remainder[type: DType, simd_width: Int](arg0: SIMD[type, simd_width], arg1: SIMD[type, simd_width])  
-> SIMD[type, simd_width]
```

Computes the remainder of the inputs.

Constraints:

The inputs must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg0** (SIMD[type, simd_width]): The first input argument.
- **arg1** (SIMD[type, simd_width]): The second input argument.

Returns:

The remainder of the inputs.

nextafter

```
nextafter[type: DType, simd_width: Int](arg0: SIMD[type, simd_width], arg1: SIMD[type, simd_width])  
-> SIMD[type, simd_width]
```

Computes the nextafter of the inputs.

Constraints:

The inputs must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg0** (SIMD[type, simd_width]): The first input argument.
- **arg1** (SIMD[type, simd_width]): The second input argument.

Returns:

The nextafter of the inputs.

j0

```
j0[type: DType, simd_width: Int](arg: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the j0 of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The j_0 of the input.

j1

$j1[\text{type: DType, simd_width: Int}](\text{arg: SIMD[type, simd_width]}) \rightarrow \text{SIMD[type, simd_width]}$

Computes the j_1 of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The j_1 of the input.

y0

$y0[\text{type: DType, simd_width: Int}](\text{arg: SIMD[type, simd_width]}) \rightarrow \text{SIMD[type, simd_width]}$

Computes the y_0 of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The y_0 of the input.

y1

$y1[\text{type: DType, simd_width: Int}](\text{arg: SIMD[type, simd_width]}) \rightarrow \text{SIMD[type, simd_width]}$

Computes the y_1 of the inputs.

Constraints:

The input must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg** (SIMD[type, simd_width]): The input argument.

Returns:

The y1 of the input.

scalb

```
scalb[type: DType, simd_width: Int](arg0: SIMD[type, simd_width], arg1: SIMD[type, simd_width]) -> SIMD[type, simd_width]
```

Computes the scalb of the inputs.

Constraints:

The inputs must be of floating point type.

Parameters:

- **type** (DType): The dtype of the input and output SIMD vector.
- **simd_width** (Int): The width of the input and output SIMD vector.

Args:

- **arg0** (SIMD[type, simd_width]): The first input argument.
- **arg1** (SIMD[type, simd_width]): The second input argument.

Returns:

The scalb of the inputs.

gcd

```
gcd(a: Int, b: Int) -> Int
```

Computes the greatest common divisor of two integers.

Constraints:

The inputs must be non-negative integers.

Args:

- **a** (Int): The first input argument.
- **b** (Int): The second input argument.

Returns:

The gcd of the inputs.

lcm

```
lcm(a: Int, b: Int) -> Int
```

Computes the least common divisor of two integers.

Constraints:

The inputs must be non-negative integers.

Args:

- **a** (Int): The first input argument.
- **b** (Int): The second input argument.

Returns:

The lcm of the inputs.

factorial

`factorial(n: Int) -> Int`

Computes the factorial of the integer.

Args:

- **n** (Int): The input value.

Returns:

The factorial of the input.

nan

`nan[type: DType]() -> SIMD[type, 1]`

Gets a NaN value for the given dtype.

Constraints:

Can only be used for FP dtypes.

Parameters:

- **type** (DType): The value dtype.

Returns:

The NaN value of the given dtype.

isnan

`isnan[type: DType, simd_width: Int](val: SIMD[type, simd_width]) -> SIMD[bool, simd_width]`

Checks if the value is Not a Number (NaN).

Parameters:

- **type** (DType): The value dtype.
- **simd_width** (Int): The width of the SIMD vector.

Args:

- **val** (SIMD[type, simd_width]): The value to check.

Returns:

True if val is NaN and False otherwise.

numerics

Module

Defines utilities to work with numeric types.

You can import these APIs from the `math` package. For example:

```
from math.numeric import FPUtils
```

FPUtils

Collection of utility functions for working with FP values.

Constraints:

The type is floating point.

Parameters:

- **type** (`DTType`): The concrete FP dtype (FP32/FP64/etc).

Aliases:

- `integral_type = _integral_type_of[$builtin::$dtype::DTType][_137x16_type]()`: The equivalent integer type of the float type.

Functions:

`mantissa_width`

```
mantissa_width() -> Int
```

Returns the mantissa width of a floating point type.

Returns:

The mantissa width.

`max_exponent`

```
max_exponent() -> Int
```

Returns the max exponent of a floating point type.

Returns:

The max exponent.

`exponent_width`

```
exponent_width() -> Int
```

Returns the exponent width of a floating point type.

Returns:

The exponent width.

`mantissa_mask`

`mantissa_mask() -> Int`

Returns the mantissa mask of a floating point type.

Returns:

The mantissa mask.

exponent_bias

`exponent_bias() -> Int`

Returns the exponent bias of a floating point type.

Returns:

The exponent bias.

sign_mask

`sign_mask() -> Int`

Returns the sign mask of a floating point type. It is computed by `1 << (exponent_width + mantissa_mask)`.

Returns:

The sign mask.

exponent_mask

`exponent_mask() -> Int`

Returns the exponent mask of a floating point type. It is computed by `~(sign_mask | mantissa_mask)`.

Returns:

The exponent mask.

exponent_mantissa_mask

`exponent_mantissa_mask() -> Int`

Returns the exponent and mantissa mask of a floating point type. It is computed by `exponent_mask + mantissa_mask`.

Returns:

The exponent and mantissa mask.

quiet_nan_mask

`quiet_nan_mask() -> Int`

Returns the quiet NaN mask for a floating point type.

The mask is defined by evaluating:

`(1<<exponent_width-1)<<mantissa_width + 1<<(mantissa_width-1)`

Returns:

The quiet NaN mask.

bitcast_to_integer

```
bitcast_to_integer(value: SIMD[type, 1]) -> Int
```

Bitcasts the floating-point value to an integer.

Args:

- **value** (SIMD[type, 1]): The floating-point type.

Returns:

An integer representation of the floating-point value.

bitcast_from_integer

```
bitcast_from_integer(value: Int) -> SIMD[type, 1]
```

Bitcasts the floating-point value from an integer.

Args:

- **value** (Int): The int value.

Returns:

An floating-point representation of the Int.

get_sign

```
get_sign(value: SIMD[type, 1]) -> Bool
```

Returns the sign of the floating point value. True if the sign is set and False otherwise.

Args:

- **value** (SIMD[type, 1]): The floating-point type.

Returns:

Returns True if the sign is set and False otherwise.

set_sign

```
set_sign(value: SIMD[type, 1], sign: Bool) -> SIMD[type, 1]
```

Sets the sign of the floating point value.

Args:

- **value** (SIMD[type, 1]): The floating-point value.
- **sign** (Bool): True to set the sign and false otherwise.

Returns:

Returns the floating point value with the sign set.

get_exponent

```
get_exponent(value: SIMD[type, 1]) -> Int
```

Returns the exponent bits of the floating-point value.

Args:

- **value** (SIMD[type, 1]): The floating-point value.

Returns:

Returns the exponent bits.

get_exponent_without_bias

```
get_exponent_without_bias(value: SIMD[type, 1]) -> Int
```

Returns the exponent bits of the floating-point value.

Args:

- **value** (SIMD[type, 1]): The floating-point value.

Returns:

Returns the exponent bits.

set_exponent

```
set_exponent(value: SIMD[type, 1], exponent: Int) -> SIMD[type, 1]
```

Sets the exponent bits of the floating-point value.

Args:

- **value** (SIMD[type, 1]): The floating-point value.
- **exponent** (Int): The exponent bits.

Returns:

Returns the floating-point value with the exponent bits set.

get_mantissa

```
get_mantissa(value: SIMD[type, 1]) -> Int
```

Gets the mantissa bits of the floating-point value.

Args:

- **value** (SIMD[type, 1]): The floating-point value.

Returns:

The mantissa bits.

set_mantissa

```
set_mantissa(value: SIMD[type, 1], mantissa: Int) -> SIMD[type, 1]
```

Sets the mantissa bits of the floating-point value.

Args:

- **value** (SIMD[type, 1]): The floating-point value.
- **mantissa** (Int): The mantissa bits.

Returns:

Returns the floating-point value with the mantissa bits set.

pack

```
pack(sign: Bool, exponent: Int, mantissa: Int) -> SIMD[type, 1]
```

Construct a floating-point value from its constituent sign, exponent, and mantissa.

Args:

- **sign** (Bool): The sign of the floating-point value.
- **exponent** (Int): The exponent of the floating-point value.
- **mantissa** (Int): The mantissa of the floating-point value.

Returns:

Returns the floating-point value.

FlushDenormals

Flushes and denormals are set to zero within the context and the state is restored to the prior value on exit.

Fields:

- **state** (SIMD[si32, 1]): The current state.

Functions:

__init__

```
__init__(inout self: Self)
```

Initializes the FlushDenormals.

__enter__

```
__enter__(self: Self)
```

Enters the context. This will set denormals to zero.

__exit__

```
__exit__(self: Self)
```

Exits the context. This will restore the prior FPState.

polynomial

Module

Provides two implementations for evaluating polynomials.

You can import these APIs from the `math` package. For example:

```
from math.polynomial import polynomial_evaluate
```

polynomial_evaluate

```
polynomial_evaluate[simd_width: Int, dtype: DType, coefficients: VariadicList[SIMD[dtype, simd_width]]](x: SIMD[dtype, simd_width]) -> SIMD[dtype, simd_width]
```

Evaluates the 1st degree polynomial using the passed in value and the specified coefficients.

These methods evaluate the polynomial using either the [Estrin scheme](#) or the Horner scheme. The Estrin scheme is only implemented for polynomials of degrees between 4 and 10. The Horner scheme is implemented for polynomials for any polynomial degree.

Parameters:

- **simd_width** (`Int`): The `simd_width` of the computed value.
- **dtype** (`DType`): The `dtype` of the value.
- **coefficients** (`VariadicList[SIMD[dtype, simd_width]]`): The coefficients.

Args:

- **x** (`SIMD[dtype, simd_width]`): The value to compute the polynomial with.

Returns:

The polynomial evaluation results using the specified value and the constant coefficients.

```
polynomial_evaluate[simd_width: Int, dtype: DType, coefficients: VariadicList[SIMD[dtype, simd_width]]](x: SIMD[dtype, simd_width]) -> SIMD[dtype, simd_width]
```

Evaluates the polynomial using the Horner scheme.

The Horner scheme evaluates the polynomial as `horner(val, coeffs)`, where `val` is a scalar and `coeffs` is a list of coefficients `[c0, c1, c2, ..., cn]`, by performing the following computation:

```
horner(val, coeffs) = c0 + val * (c1 + val * (c2 + val * (... + val * cn)))
= fma(val, horner(val, coeffs[1:]), c0)
```

Parameters:

- **simd_width** (`Int`): The `simd_width` of the computed value.
- **dtype** (`DType`): The `dtype` of the value.
- **coefficients** (`VariadicList[SIMD[dtype, simd_width]]`): The coefficients.

Args:

- **x** (`SIMD[dtype, simd_width]`): The value to compute the polynomial with.

Returns:

The polynomial evaluation results using the specified value and the constant coefficients.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

buffer

Module

Implements the Buffer class.

You can import these APIs from the `memory` package. For example:

```
from memory.buffer import Buffer
```

Buffer

Defines a Buffer which can be parametrized on a static size and Dtype.

The Buffer does not own its underlying pointer.

Parameters:

- **size** (`Dim`): The static size (if known) of the Buffer.
- **type** (`DType`): The element type of the Buffer.

Fields:

- **data** (`DTypePointer[type]`): The underlying data pointer of the data.
- **dynamic_size** (`Int`): The dynamic size of the buffer.
- **dtype** (`DType`): The dynamic data type of the buffer.

Functions:

`__init__`

```
__init__() -> Self
```

Default initializer for Buffer. By default the fields are all initialized to 0.

Returns:

The NDBuffer object.

```
__init__(ptr: Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType> type, "value">>]) -> Self
```

Constructs a Buffer with statically known size and type.

Constraints:

The size is known.

Args:

- **ptr** (`Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType> type, "value">>]`): Pointer to the data.

Returns:

The buffer object.

```
__init__(ptr: DTypePointer[type]) -> Self
```

Constructs a Buffer with statically known size and type.

Constraints:

The size is known.

Args:

- **ptr** (DTypePointer[type]): Pointer to the data.

Returns:

The buffer object.

```
__init__(ptr: Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType> type, "value">>], in_size: Int) -> Self
```

Constructs a Buffer with statically known type.

Constraints:

The size is unknown.

Args:

- **ptr** (Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype":_DType> type, "value">>]): Pointer to the data.
- **in_size** (Int): Dynamic size of the buffer.

Returns:

The buffer object.

```
__init__(ptr: DTypePointer[type], in_size: Int) -> Self
```

Constructs a Buffer with statically known type.

Constraints:

The size is unknown.

Args:

- **ptr** (DTypePointer[type]): Pointer to the data.
- **in_size** (Int): Dynamic size of the buffer.

Returns:

The buffer object.

```
__init__(data: DTypePointer[type], dynamic_size: Int, dtype: DType) -> Self
```

__copyinit__

```
__copyinit__(existing: Self) -> Self
```

__getitem__

```
__getitem__(self: Self, idx: Int) -> SIMD[type, 1]
```

Loads a single element (SIMD of size 1) from the buffer at the specified index.

Args:

- **idx** (Int): The index into the Buffer.

Returns:

The value at the `idx` position.

__setitem__

```
__setitem__(self: Self, idx: Int, val:  
scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype"::_DType> type, "value">>)
```

Stores a single value into the buffer at the specified index.

Args:

- **idx** (Int): The index into the Buffer.
- **val** (scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype"::_DType> type, "value">>): The value to store.

```
__setitem__(self: Self, idx: Int, val: SIMD[type, 1])
```

Stores a single value into the buffer at the specified index.

Args:

- **idx** (Int): The index into the Buffer.
- **val** (SIMD[type, 1]): The value to store.

__len__

```
__len__(self: Self) -> Int
```

Gets the size if it is a known constant, otherwise it gets the `dynamic_size`.

This method is used by `Buffer.__len__` to get the size of the buffer. If the Buffer size is a known constant, then the size is returned. Otherwise, the `dynamic_size` is returned.

Returns:

The size if static otherwise `dynamic_size`.

simd_load

```
simd_load[width: Int](self: Self, idx: Int) -> SIMD[type, width]
```

Loads a simd value from the buffer at the specified index.

Parameters:

- **width** (Int): The `simd_width` of the load.

Args:

- **idx** (Int): The index into the Buffer.

Returns:

The simd value starting at the `idx` position and ending at `idx+width`.

aligned_simd_load

```
aligned_simd_load[width: Int, alignment: Int](self: Self, idx: Int) -> SIMD[type, width]
```

Loads a simd value from the buffer at the specified index.

Parameters:

- **width** (Int): The simd_width of the load.
- **alignment** (Int): The alignment value.

Args:

- **idx** (Int): The index into the Buffer.

Returns:

The simd value starting at the `idx` position and ending at `idx+width`.

simd_store

```
simd_store[width: Int](self: Self, idx: Int, val: SIMD[type, width])
```

Stores a simd value into the buffer at the specified index.

Parameters:

- **width** (Int): The width of the simd vector.

Args:

- **idx** (Int): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

aligned_simd_store

```
aligned_simd_store[width: Int, alignment: Int](self: Self, idx: Int, val: SIMD[type, width])
```

Stores a simd value into the buffer at the specified index.

Parameters:

- **width** (Int): The width of the simd vector.
- **alignment** (Int): The alignment value.

Args:

- **idx** (Int): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

simd_nt_store

```
simd_nt_store[width: Int](self: Self, idx: Int, val: SIMD[type, width])
```

Stores a simd value using non-temporal store.

Constraints:

The address must be properly aligned, 64B for avx512, 32B for avx2, and 16B for avx.

Parameters:

- **width** (Int): The width of the simd vector.

Args:

- **idx** (Int): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

prefetch

```
prefetch[params: PrefetchOptions](self: Self, idx: Int)
```

Prefetches the data at the given index.

Parameters:

- **params** (PrefetchOptions): The prefetch configuration.

Args:

- **idx** (Int): The index of the prefetched location.

bytecount

```
bytecount(self: Self) -> Int
```

Returns the size of the Buffer in bytes.

Returns:

The size of the Buffer in bytes.

zero

```
zero(self: Self)
```

Sets all bytes of the Buffer to 0.

simd_fill

```
simd_fill[simd_width: Int](self: Self, val: SIMD[type, 1])
```

Assigns val to all elements in chunks of size simd_width.

Parameters:

- **simd_width** (Int): The simd_width of the fill.

Args:

- **val** (SIMD[type, 1]): The value to store.

fill

```
fill(self: Self, val: SIMD[type, 1])
```

Assigns val to all elements in the Buffer.

The fill is performed in chunks of size N, where N is the native SIMD width of type on the system.

Args:

- **val** (SIMD[type, 1]): The value to store.

write_file

```
write_file(self: Self, path: Path)
```

Write values to a file.

Args:

- **path** (Path): Path to the output file.

aligned_stack_allocation

```
aligned_stack_allocation[alignment: Int]() -> Self
```

Constructs a buffer instance backed by stack allocated memory space.

Parameters:

- **alignment** (Int): Address alignment requirement for the allocation.

Returns:

Constructed buffer with the allocated space.

stack_allocation

```
stack_allocation() -> Self
```

Constructs a buffer instance backed by stack allocated memory space.

Returns:

Constructed buffer with the allocated space.

NDBuffer

An N-dimensional Buffer.

NDBuffer can be parametrized on rank, static dimensions and Dtype. It does not own its underlying pointer.

Parameters:

- **rank** (Int): The rank of the buffer.
- **shape** (DimList): The static size (if known) of the buffer.
- **type** (DType): The element type of the buffer.

Fields:

- **data** (DTypePointer[type]): The underlying data for the buffer. The pointer is not owned by the NDBuffer.
- **dynamic_shape** (StaticIntTuple[rank]): The dynamic value of the shape.
- **dynamic_stride** (StaticIntTuple[rank]): The dynamic stride of the buffer.
- **is_contiguous** (Bool): True if the contents of the buffer are contiguous in memory.

Functions:

__init__

```
__init__() -> Self
```

Default initializer for NDBuffer. By default the fields are all initialized to 0.

Returns:

The NDBuffer object.

```
__init__(ptr: Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType>  
type, "value">>]) -> Self
```

Constructs an NDBuffer with statically known rank, shapes and type.

Constraints:

The rank, shapes, and type are known.

Args:

- **ptr** (Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype":_DType>
type, "value">>]): Pointer to the data.

Returns:

The NDBuffer object.

```
__init__(ptr: DTypePointer[type]) -> Self
```

Constructs an NDBuffer with statically known rank, shapes and type.

Constraints:

The rank, shapes, and type are known.

Args:

- **ptr** (DTypePointer[type]): Pointer to the data.

Returns:

The NDBuffer object.

```
__init__(ptr: pointer<scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType>  
type, "value">>>, dynamic_shape: StaticIntTuple[rank]) -> Self
```

Constructs an NDBuffer with statically known rank, but dynamic shapes and type.

Constraints:

The rank is known.

Args:

- **ptr** (pointer<scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype":_DType>
type, "value">>>): Pointer to the data.
- **dynamic_shape** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing shapes.

Returns:

The NDBuffer object.

```
__init__(ptr: Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype":_DType>  
type, "value">>], dynamic_shape: StaticIntTuple[rank]) -> Self
```

Constructs an NDBuffer with statically known rank, but dynamic shapes and type.

Constraints:

The rank is known.

Args:

- **ptr** (Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype"::_DType> type, "value">>]): Pointer to the data.
- **dynamic_shape** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing shapes.

Returns:

The NDBuffer object.

```
__init__(ptr: DTypePointer[type], dynamic_shape: StaticIntTuple[rank]) -> Self
```

Constructs an NDBuffer with statically known rank, but dynamic shapes and type.

Constraints:

The rank is known.

Args:

- **ptr** (DTypePointer[type]): Pointer to the data.
- **dynamic_shape** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing shapes.

Returns:

The NDBuffer object.

```
__init__(ptr: Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype"::_DType> type, "value">>], dynamic_shape: StaticIntTuple[rank], dynamic_stride: StaticIntTuple[rank]) -> Self
```

Constructs a strided NDBuffer with statically known rank, but dynamic shapes and type.

Constraints:

The rank is known.

Args:

- **ptr** (Pointer[scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype"::_DType> type, "value">>]): Pointer to the data.
- **dynamic_shape** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing shapes.
- **dynamic_stride** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing strides.

Returns:

The NDBuffer object.

```
__init__(ptr: DTypePointer[type], dynamic_shape: StaticIntTuple[rank], dynamic_stride: StaticIntTuple[rank]) -> Self
```

Constructs a strided NDBuffer with statically known rank, but dynamic shapes and type.

Constraints:

The rank is known.

Args:

- **ptr** (DTypePointer[type]): Pointer to the data.
- **dynamic_shape** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing shapes.
- **dynamic_stride** (StaticIntTuple[rank]): A static tuple of size ‘rank’ representing strides.

Returns:

The NDBuffer object.

```
__init__(data: DTypePointer[type], dynamic_shape: StaticIntTuple[rank], dynamic_stride:  
StaticIntTuple[rank], is_contiguous: Bool) -> Self
```

__getitem__

```
__getitem__(self: Self, *idx: Int) -> SIMD[type, 1]
```

Gets an element from the buffer from the specified index.

Args:

- **idx** (*Int): Index of the element to retrieve.

Returns:

The value of the element.

```
__getitem__(self: Self, idx: StaticIntTuple[rank]) -> SIMD[type, 1]
```

Gets an element from the buffer from the specified index.

Args:

- **idx** (StaticIntTuple[rank]): Index of the element to retrieve.

Returns:

The value of the element.

__setitem__

```
__setitem__(self: Self, idx: StaticIntTuple[rank], val: SIMD[type, 1])
```

Stores a single value into the buffer at the specified index.

Args:

- **idx** (StaticIntTuple[rank]): The index into the Buffer.
- **val** (SIMD[type, 1]): The value to store.

get_rank

```
get_rank(self: Self) -> Int
```

Returns the rank of the buffer.

Returns:

The rank of NDBuffer.

get_shape

```
get_shape(self: Self) -> StaticIntTuple[rank]
```

Returns the shapes of the buffer.

Returns:

A static tuple of size ‘rank’ representing shapes of the NDBuffer.

get_nd_index

```
get_nd_index(self: Self, idx: Int) -> StaticIntTuple[rank]
```

Computes the NDBuffer's ND-index based on the flat index.

Args:

- **idx** (Int): The flat index.

Returns:

The index positions.

__len__

```
__len__(self: Self) -> Int
```

Computes the NDBuffer's number of elements.

Returns:

The total number of elements in the NDBuffer.

num_elements

```
num_elements(self: Self) -> Int
```

Computes the NDBuffer's number of elements.

Returns:

The total number of elements in the NDBuffer.

size

```
size(self: Self) -> Int
```

Computes the NDBuffer's number of elements.

Returns:

The total number of elements in the NDBuffer.

simd_load

```
simd_load[width: Int](self: Self, *idx: Int) -> SIMD[type, width]
```

Loads a simd value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The simd_width of the load.

Args:

- **idx** (*Int): The index into the NDBuffer.

Returns:

The simd value starting at the idx position and ending at idx+width.

```
simd_load[width: Int](self: Self, idx: VariadicList[Int]) -> SIMD[type, width]
```

Loads a SIMD value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The SIMD_width of the load.

Args:

- **idx** (VariadicList[Int]): The index into the NDBuffer.

Returns:

The SIMD value starting at the `idx` position and ending at `idx+width`.

```
simd_load[width: Int](self: Self, idx: StaticIntTuple[rank]) -> SIMD[type, width]
```

Loads a SIMD value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The SIMD_width of the load.

Args:

- **idx** (StaticIntTuple[rank]): The index into the NDBuffer.

Returns:

The SIMD value starting at the `idx` position and ending at `idx+width`.

```
simd_load[width: Int](self: Self, idx: StaticTuple[rank, Int]) -> SIMD[type, width]
```

Loads a SIMD value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The SIMD_width of the load.

Args:

- **idx** (StaticTuple[rank, Int]): The index into the NDBuffer.

Returns:

The SIMD value starting at the `idx` position and ending at `idx+width`.

aligned_simd_load

```
aligned_simd_load[width: Int, alignment: Int](self: Self, *idx: Int) -> SIMD[type, width]
```

Loads a SIMD value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The simd_width of the load.
- **alignment** (Int): The alignment value.

Args:

- **idx** (*Int): The index into the NDBuffer.

Returns:

The simd value starting at the `idx` position and ending at `idx+width`.

```
aligned_simd_load[width: Int, alignment: Int](self: Self, idx: VariadicList[Int]) -> SIMD[type, width]
```

Loads a simd value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The simd_width of the load.
- **alignment** (Int): The alignment value.

Args:

- **idx** (VariadicList[Int]): The index into the NDBuffer.

Returns:

The simd value starting at the `idx` position and ending at `idx+width`.

```
aligned_simd_load[width: Int, alignment: Int](self: Self, idx: StaticIntTuple[rank]) -> SIMD[type, width]
```

Loads a simd value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The simd_width of the load.
- **alignment** (Int): The alignment value.

Args:

- **idx** (StaticIntTuple[rank]): The index into the NDBuffer.

Returns:

The simd value starting at the `idx` position and ending at `idx+width`.

```
aligned_simd_load[width: Int, alignment: Int](self: Self, idx: StaticTuple[rank, Int]) -> SIMD[type, width]
```

Loads a simd value from the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The simd_width of the load.
- **alignment** (Int): The alignment value.

Args:

- **idx** (StaticTuple[rank, Int]): The index into the NDBuffer.

Returns:

The simd value starting at the `idx` position and ending at `idx+width`.

simd_store

```
simd_store[width: Int](self: Self, idx: StaticIntTuple[rank], val: SIMD[type, width])
```

Stores a simd value into the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The width of the simd vector.

Args:

- **idx** (StaticIntTuple[rank]): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

```
simd_store[width: Int](self: Self, idx: StaticTuple[rank, Int], val: SIMD[type, width])
```

Stores a simd value into the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The width of the simd vector.

Args:

- **idx** (StaticTuple[rank, Int]): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

aligned_simd_store

```
aligned_simd_store[width: Int, alignment: Int](self: Self, idx: StaticIntTuple[rank], val: SIMD[type, width])
```

Stores a simd value into the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The width of the SIMD vector.
- **alignment** (Int): The alignment value.

Args:

- **idx** (StaticIntTuple[rank]): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

```
aligned_simd_store[width: Int, alignment: Int](self: Self, idx: StaticTuple[rank, Int], val: SIMD[type, width])
```

Stores a SIMD value into the buffer at the specified index.

Constraints:

The buffer must be contiguous or width must be 1.

Parameters:

- **width** (Int): The width of the SIMD vector.
- **alignment** (Int): The alignment value.

Args:

- **idx** (StaticTuple[rank, Int]): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

simd_nt_store

```
simd_nt_store[width: Int](self: Self, idx: StaticIntTuple[rank], val: SIMD[type, width])
```

Stores a SIMD value using non-temporal store.

Constraints:

The buffer must be contiguous. The address must be properly aligned, 64B for avx512, 32B for avx2, and 16B for avx.

Parameters:

- **width** (Int): The width of the SIMD vector.

Args:

- **idx** (StaticIntTuple[rank]): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

```
simd_nt_store[width: Int](self: Self, idx: StaticTuple[rank, Int], val: SIMD[type, width])
```

Stores a SIMD value using non-temporal store.

Constraints:

The buffer must be contiguous. The address must be properly aligned, 64B for avx512, 32B for avx2, and 16B for avx.

Parameters:

- **width** (Int): The width of the SIMD vector.

Args:

- **idx** (StaticTuple[rank, Int]): The index into the Buffer.
- **val** (SIMD[type, width]): The value to store.

dim

```
dim[index: Int](self: Self) -> Int
```

Gets the buffer dimension at the given index.

Parameters:

- **index** (Int): The number of dimension to get.

Returns:

The buffer size at the given dimension.

```
dim(self: Self, index: Int) -> Int
```

Gets the buffer dimension at the given index.

Args:

- **index** (Int): The number of dimension to get.

Returns:

The buffer size at the given dimension.

stride

```
stride(self: Self, index: Int) -> Int
```

Gets the buffer stride at the given index.

Args:

- **index** (Int): The number of dimension to get the stride for.

Returns:

The stride at the given dimension.

flatten

```
flatten(self: Self) -> Buffer[#pop.variant<:i1 0, 0>, type]
```

Constructs a flattened Buffer counterpart for this NDBuffer.

Constraints:

The buffer must be contiguous.

Returns:

Constructed Buffer object.

make_dims_unknown

```
make_dims_unknown(self: Self) -> NDBuffer[rank, create_unknown[$builtin::$int::Int][rank](), type]
```

Rebinds the NDBuffer to one with unknown shape.

Returns:

The rebound NDBuffer with unknown shape.

bytecount

```
bytecount(self: Self) -> Int
```

Returns the size of the NDBuffer in bytes.

Returns:

The size of the NDBuffer in bytes.

zero

```
zero(self: Self)
```

Sets all bytes of the NDBuffer to 0.

Constraints:

The buffer must be contiguous.

simd_fill

```
simd_fill[simd_width: Int](self: Self, val: SIMD[type, 1])
```

Assigns val to all elements in chunks of size simd_width.

Parameters:

- **simd_width** (Int): The simd_width of the fill.

Args:

- **val** (SIMD[type, 1]): The value to store.

write_file

```
write_file(self: Self, path: Path)
```

Write values to a file.

Args:

- **path** (Path): Path to the output file.

fill

```
fill(self: Self, val: SIMD[type, 1])
```

Assigns val to all elements in the Buffer.

The fill is performed in chunks of size N, where N is the native SIMD width of type on the system.

Args:

- **val** (SIMD[type, 1]): The value to store.

aligned_stack_allocation

```
aligned_stack_allocation[alignment: Int]() -> Self
```

Constructs an NDBuffer instance backed by stack allocated memory space.

Parameters:

- **alignment** (Int): Address alignment requirement for the allocation.

Returns:

Constructed NDBuffer with the allocated space.

stack_allocation

```
stack_allocation() -> Self
```

Constructs an NDBuffer instance backed by stack allocated memory space.

Returns:

Constructed NDBuffer with the allocated space.

prefetch

```
prefetch[params: PrefetchOptions](self: Self, *idx: Int)
```

Prefetches the data at the given index.

Parameters:

- **params** (PrefetchOptions): The prefetch configuration.

Args:

- **idx** (*Int): The N-D index of the prefetched location.

```
prefetch[params: PrefetchOptions](self: Self, indices: StaticIntTuple[rank])
```

Prefetches the data at the given index.

Parameters:

- **params** (PrefetchOptions): The prefetch configuration.

Args:

- **indices** (StaticIntTuple[rank]): The N-D index of the prefetched location.

DynamicRankBuffer

DynamicRankBuffer represents a buffer with unknown rank, shapes and dtype.

It is not as efficient as the statically ranked buffer, but is useful when interacting with external functions. In particular the shape is represented as a fixed (ie `_MAX_RANK`) array of dimensions to simplify the ABI.

Fields:

- **data** (DTypePointer[invalid]): The pointer to the buffer.
- **rank** (Int): The buffer rank. Has a max value of `_MAX_RANK`.
- **shape** (StaticIntTuple[8]): The dynamic shape of the buffer.
- **type** (DType): The dynamic dtype of the buffer.

Functions:

__init__

`__init__(data: DTypePointer[invalid], rank: Int, shape: StaticIntTuple[8], type: DType) -> Self`

Construct DynamicRankBuffer.

Args:

- **data** (DTypePointer[invalid]): Pointer to the underlying data.
- **rank** (Int): Rank of the buffer.
- **shape** (StaticIntTuple[8]): Shapes of the buffer.
- **type** (DType): dtype of the buffer.

Returns:

Constructed DynamicRankBuffer.

to_buffer

`to_buffer[type: DType](self: Self) -> Buffer[#pop.variant<:i1 0, 0>, type]`

Casts DynamicRankBuffer to Buffer.

Parameters:

- **type** (DType): dtype of the buffer.

Returns:

Constructed Buffer.

to_ndbuffer

`to_ndbuffer[rank: Int, type: DType](self: Self) -> NDBuffer[rank, create_unknown[$builtin::$int:Int][rank](), type]`

Casts the buffer to NDBuffer.

Constraints:

Rank of DynamicRankBuffer must equal rank of NDBuffer.

Parameters:

- **rank** (Int): Rank of the buffer.
- **type** (DType): dtype of the buffer.

Returns:

Constructed NDBuffer.

`to_ndbuffer[rank: Int, type: DType](self: Self, stride: StaticIntTuple[rank]) -> NDBuffer[rank, create_unknown[$builtin::$int:Int][rank](), type]`

Casts the buffer to NDBuffer.

Constraints:

Rank of DynamicRankBuffer must equal rank of NDBuffer.

Parameters:

- **rank** (Int): Rank of the buffer.
- **type** (DType): dtype of the buffer.

Args:

- **stride** (StaticIntTuple[rank]): Strides of the buffer.

Returns:

Constructed NDBuffer.

rank_dispatch

```
rank_dispatch[func: fn[Int]() capturing -> None](self: Self)
```

Dispatches the function call based on buffer rank.

Constraints:

Rank must be positive and less or equal to 8.

Parameters:

- **func** (fn[Int]() capturing -> None): Function to dispatch. The function should be parametrized on an index parameter, which will be used for rank when the function will be called.

```
rank_dispatch[func: fn[Int]() capturing -> None](self: Self, out_chain: OutputChainPtr)
```

Dispatches the function call based on buffer rank.

Constraints:

Rank must be positive and less or equal to 8.

Parameters:

- **func** (fn[Int]() capturing -> None): Function to dispatch. The function should be parametrized on an index parameter, which will be used for rank when the function will be called.

Args:

- **out_chain** (OutputChainPtr): The output chain.

num_elements

```
num_elements(self: Self) -> Int
```

Gets number of elements in the buffer.

Returns:

The number of elements in the buffer.

get_shape

```
get_shape[rank: Int](self: Self) -> StaticIntTuple[rank]
```

Gets a static tuple representing the buffer shape.

Parameters:

- **rank** (Int): Rank of the buffer.

Returns:

A static tuple of size ‘Rank’ filled with buffer shapes.

dim

```
dim(self: Self, idx: Int) -> Int
```

Gets given dimension.

Args:

- **idx** (Int): The dimension index.

Returns:

The buffer size on the given dimension.

partial_simd_load

```
partial_simd_load[type: DType, width: Int](storage: DTypePointer[type], lbound: Int, rbound: Int, pad_value: SIMD[type, 1]) -> SIMD[type, width]
```

Loads a vector with dynamic bound.

Out of bound data will be filled with pad value. Data is valid if lbound <= idx < rbound for idx from 0 to (simd_width-1). For example:

```
addr 0 1 2 3
data x 42 43 x
```

```
partial_simd_load[4](addr0,1,3) #gives [0 42 43 0]
```

Parameters:

- **type** (DType): The underlying dtype of computation.
- **width** (Int): The system simd vector size.

Args:

- **storage** (DTypePointer[type]): Pointer to the address to perform load.
- **lbound** (Int): Lower bound of valid index within simd (inclusive).
- **rbound** (Int): Upper bound of valid index within simd (non-inclusive).
- **pad_value** (SIMD[type, 1]): Value to fill for out of bound indices.

Returns:

The SIMD vector loaded and zero-filled.

partial_simd_store

```
partial_simd_store[type: DType, width: Int](storage: DTypePointer[type], lbound: Int, rbound: Int, data: SIMD[type, width])
```

Stores a vector with dynamic bound.

Out of bound data will ignored. Data is valid if lbound <= idx < rbound for idx from 0 to (simd_width-1).

e.g. addr 0 1 2 3 data 0 0 0 0

```
partial_simd_load[4](addr0,1,3, [-1, 42,43, -1]) #gives [0 42 43 0]
```

Parameters:

- **type** (DType): The underlying dtype of computation.
- **width** (Int): The system SIMD vector size.

Args:

- **storage** (DTypePointer[type]): Pointer to the address to perform load.
- **lbound** (Int): Lower bound of valid index within SIMD (inclusive).
- **rbound** (Int): Upper bound of valid index within SIMD (non-inclusive).
- **data** (SIMD[type, width]): The vector value to store.

prod_dims

```
prod_dims[start_dim: Int, end_dim: Int, rank: Int, shape: DimList, type: DType](x: NDBuffer[rank, shape, type]) -> Int
```

Computes the product of a slice of the given buffer's dimensions.

Parameters:

- **start_dim** (Int): The index at which to begin computing the product.
- **end_dim** (Int): The index at which to stop computing the product.
- **rank** (Int): The rank of the NDBuffer.
- **shape** (DimList): The shape of the NDBuffer.
- **type** (DType): The element-type of the NDBuffer.

Args:

- **x** (NDBuffer[rank, shape, type]): The NDBuffer whose dimensions will be multiplied.

Returns:

The product of the specified slice of the buffer's dimensions.

memory

Module

Defines functions for memory manipulations.

You can import these APIs from the `memory` package. For example:

```
from memory import memcmp
```

`memcmp`

```
memcmp[type: DType](s1: DTypePointer[type], s2: DTypePointer[type], count: Int) -> Int
```

Compares two buffers. Both strings are assumed to be of the same length.

Parameters:

- **type** (`DType`): The element dtype.

Args:

- **s1** (`DTypePointer[type]`): The first buffer address.
- **s2** (`DTypePointer[type]`): The second buffer address.
- **count** (`Int`): The number of elements in the buffers.

Returns:

Returns 0 if the bytes buffers are identical, 1 if $s1 > s2$, and -1 if $s1 < s2$. The comparison is performed by the first different byte in the buffer.

```
memcmp[type: AnyType](s1: Pointer[*"type"], s2: Pointer[*"type"], count: Int) -> Int
```

Compares two buffers. Both strings are assumed to be of the same length.

Parameters:

- **type** (`AnyType`): The element type.

Args:

- **s1** (`Pointer[*"type"]`): The first buffer address.
- **s2** (`Pointer[*"type"]`): The second buffer address.
- **count** (`Int`): The number of elements in the buffers.

Returns:

Returns 0 if the bytes strings are identical, 1 if $s1 > s2$, and -1 if $s1 < s2$. The comparison is performed by the first different byte in the byte strings.

`memcpy`

```
memcpy[type: AnyType](dest: Pointer[*"type"], src: Pointer[*"type"], count: Int)
```

Copies a memory area.

Parameters:

- **type** (`AnyType`): The element type.

Args:

- **dest** (Pointer[*"type"]): The destination pointer.
- **src** (Pointer[*"type"]): The source pointer.
- **count** (Int): The number of elements to copy.

```
memcpy[type: DType](dest: DTypePointer[type], src: DTypePointer[type], count: Int)
```

Copies a memory area.

Parameters:

- **type** (DType): The element dtype.

Args:

- **dest** (DTypePointer[type]): The destination pointer.
- **src** (DTypePointer[type]): The source pointer.
- **count** (Int): The number of elements to copy (not bytes!).

```
memcpy[type: DType, size: Dim](dest: Buffer[size, type], src: Buffer[size, type])
```

Copies a memory buffer from `src` to `dest`.

Parameters:

- **type** (DType): The element dtype.
- **size** (Dim): Number of elements in the buffer.

Args:

- **dest** (Buffer[size, type]): The destination buffer.
- **src** (Buffer[size, type]): The source buffer.

memset

```
memset[type: DType](ptr: DTypePointer[type], value: SIMD[ui8, 1], count: Int)
```

Fills memory with the given value.

Parameters:

- **type** (DType): The element dtype.

Args:

- **ptr** (DTypePointer[type]): Pointer to the beginning of the memory block to fill.
- **value** (SIMD[ui8, 1]): The value to fill with.
- **count** (Int): Number of elements to fill (in elements, not bytes).

memset_zero

```
memset_zero[type: DType](ptr: DTypePointer[type], count: Int)
```

Fills memory with zeros.

Parameters:

- **type** (DType): The element dtype.

Args:

- **ptr** (DTypePointer[type]): Pointer to the beginning of the memory block to fill.
- **count** (Int): Number of elements to set (in elements, not bytes).

```
memset_zero[type: AnyType](ptr: Pointer[*"type"], count: Int)
```

Fills memory with zeros.

Parameters:

- **type** (AnyType): The element type.

Args:

- **ptr** (Pointer[*"type"]): Pointer to the beginning of the memory block to fill.
- **count** (Int): Number of elements to fill (in elements, not bytes).

stack_allocation

```
stack_allocation[count: Int, type: DType]() -> DTypePointer[type]
```

Allocates data buffer space on the stack given a data type and number of elements.

Parameters:

- **count** (Int): Number of elements to allocate memory for.
- **type** (DType): The data type of each element.

Returns:

A data pointer of the given type pointing to the allocated space.

```
stack_allocation[count: Int, type: DType, alignment: Int]() -> DTypePointer[type]
```

Allocates data buffer space on the stack given a data type and number of elements.

Parameters:

- **count** (Int): Number of elements to allocate memory for.
- **type** (DType): The data type of each element.
- **alignment** (Int): Address alignment of the allocated data.

Returns:

A data pointer of the given type pointing to the allocated space.

```
stack_allocation[count: Int, type: AnyType]() -> Pointer[*"type"]
```

Allocates data buffer space on the stack given a data type and number of elements.

Parameters:

- **count** (Int): Number of elements to allocate memory for.
- **type** (AnyType): The data type of each element.

Returns:

A data pointer of the given type pointing to the allocated space.

```
stack_allocation[count: Int, type: AnyType, alignment: Int]() -> Pointer[*"type"]
```

Allocates data buffer space on the stack given a data type and number of elements.

Parameters:

- **count** (Int): Number of elements to allocate memory for.
- **type** (AnyType): The data type of each element.

- **alignment** (`Int`): Address alignment of the allocated data.

Returns:

A data pointer of the given type pointing to the allocated space.

unsafe

Module

Implements classes for working with unsafe pointers.

You can import these APIs from the `memory` package. For example:

```
from memory.unsafe import Pointer
```

Pointer

Defines a `Pointer` struct that contains an address of any `mlirtype`.

Parameters:

- **type** (`AnyType`): Type of the underlying data.

Aliases:

- `pointer_type = pointer<*"type">`

Fields:

- **address** (`pointer<*"type">`): The pointed-to address.

Functions:

`__init__`

```
__init__() -> Self
```

Constructs a null `Pointer` from the value of `pop.pointer` type.

Returns:

Constructed `Pointer` object.

```
__init__(address: Self) -> Self
```

Constructs a `Pointer` from the address.

Args:

- **address** (`Self`): The input pointer.

Returns:

Constructed `Pointer` object.

```
__init__(address: pointer<*"type">) -> Self
```

Constructs a `Pointer` from the address.

Args:

- **address** (`pointer<*"type">`): The input pointer address.

Returns:

Constructed `Pointer` object.

`__init__(value: SIMD[address, 1]) -> Self`

Constructs a Pointer from the value of scalar address.

Args:

- **value** (SIMD[address, 1]): The input pointer index.

Returns:

Constructed Pointer object.

__bool__

`__bool__(self: Self) -> Bool`

Checks if the pointer is null.

Returns:

Returns False if the pointer is null and True otherwise.

__getitem__

`__getitem__(self: Self, offset: Int) -> *"type"`

Loads the value the Pointer object points to with the given offset.

Args:

- **offset** (Int): The offset to load from.

Returns:

The loaded value.

__eq__

`__eq__(self: Self, rhs: Self) -> Bool`

Returns True if the two pointers are equal.

Args:

- **rhs** (Self): The value of the other pointer.

Returns:

True if the two pointers are equal and False otherwise.

__ne__

`__ne__(self: Self, rhs: Self) -> Bool`

Returns True if the two pointers are not equal.

Args:

- **rhs** (Self): The value of the other pointer.

Returns:

True if the two pointers are not equal and False otherwise.

__add__

```
__add__(self: Self, rhs: Int) -> Self
```

Returns a new pointer shifted by the specified offset.

Args:

- **rhs** (Int): The offset.

Returns:

The new Pointer shifted by the offset.

__sub__

```
__sub__(self: Self, rhs: Int) -> Self
```

Returns a new pointer shifted back by the specified offset.

Args:

- **rhs** (Int): The offset.

Returns:

The new Pointer shifted back by the offset.

__iadd__

```
__iadd__(inout self: Self, rhs: Int)
```

Shifts the current pointer by the specified offset.

Args:

- **rhs** (Int): The offset.

__isub__

```
__isub__(inout self: Self, rhs: Int)
```

Shifts back the current pointer by the specified offset.

Args:

- **rhs** (Int): The offset.

get_null

```
get_null() -> Self
```

Constructs a Pointer representing nullptr.

Returns:

Constructed nullptr Pointer object.

address_of

```
address_of(inout *arg: "type") -> Self
```

Gets the address of the argument.

Args:

- **arg** (*"type"): The value to get the address of.

Returns:

A pointer struct which contains the address of the argument.

load

```
load(self: Self, offset: Int) -> *"type"
```

Loads the value the Pointer object points to with the given offset.

Args:

- **offset** (Int): The offset to load from.

Returns:

The loaded value.

```
load(self: Self) -> *"type"
```

Loads the value the Pointer object points to.

Returns:

The loaded value.

store

```
store(self: Self, offset: Int, *value: "type")
```

Stores the specified value to the location the Pointer object points to with the given offset.

Args:

- **offset** (Int): The offset to load from.
- **value** (*"type"): The value to store.

```
store(self: Self, *value: "type")
```

Stores the specified value to the location the Pointer object points to.

Args:

- **value** (*"type"): The value to store.

alloc

```
alloc(count: Int) -> Self
```

Heap-allocates a number of element of the specified type.

Args:

- **count** (Int): The number of elements to allocate (note that this is not the bytecount).

Returns:

A new Pointer object which has been allocated on the heap.

aligned_alloc

aligned_alloc(alignment: Int, count: Int) -> Self

Heap-allocates a number of element of the specified type using the specified alignment.

Args:

- **alignment** (Int): The alignment used for the allocation.
- **count** (Int): The number of elements to allocate (note that this is not the bytecount).

Returns:

A new Pointer object which has been allocated on the heap.

free

free(self: Self)

Frees the heap allocated memory.

bitcast

bitcast[new_type: AnyType](self: Self) -> Pointer[new_type]

Bitcasts a Pointer to a different type.

Parameters:

- **new_type** (AnyType): The target type.

Returns:

A new Pointer object with the specified type and the same address, as the original Pointer.

offset

offset(self: Self, idx: Int) -> Self

Returns a new pointer shifted by the specified offset.

Args:

- **idx** (Int): The offset.

Returns:

The new Pointer shifted by the offset.

DTypePointer

Defines a DTypePointer struct that contains an address of the given dtype.

Parameters:

- **type** (DType): DType of the underlying data.

Aliases:

- element_type = scalar<#lit.struct.extract<:!kgen.declref<@"\$builtin": @"\$dtype"::@DType> type, "value">>
- pointer_type = pointer<scalar<#lit.struct.extract<:!kgen.declref<@"\$builtin": @"\$dtype"::@DType> type, "value">>>

Fields:

- **address** (pointer<scalar<#lit.struct.extract<:_!kgen.declref<@"\$builtin":@"\$dtype"::_DType> type, "value">>>>): The pointed-to address.

Functions:

`__init__`

```
__init__() -> Self
```

Constructs a null `DTypePointer` from the given type.

Returns:

Constructed `DTypePointer` object.

```
__init__(address: Self) -> Self
```

Constructs a `DTypePointer` from the address.

Args:

- **address** (`Self`): The input pointer.

Returns:

Constructed Pointer object.

```
__init__(address: pointer<scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype"::_DType> type, "value">>>>) -> Self
```

Constructs a `DTypePointer` from the given address.

Args:

- **address** (pointer<scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype"::_DType> type, "value">>>>): The input pointer.

Returns:

Constructed `DTypePointer` object.

```
__init__(value: Pointer[ SIMD[type, 1] ]) -> Self
```

Constructs a `DTypePointer` from a scalar pointer of the same type.

Args:

- **value** (`Pointer[SIMD[type, 1]]`): The scalar pointer.

Returns:

Constructed `DTypePointer`.

```
__init__(value: SIMD[address, 1]) -> Self
```

Constructs a `DTypePointer` from the value of scalar address.

Args:

- **value** (`SIMD[address, 1]`): The input pointer index.

Returns:

Constructed DTypePointer object.

__bool__

`__bool__(self: Self) -> Bool`

Checks if the pointer is *null*.

Returns:

Returns False if the pointer is *null* and True otherwise.

__lt__

`__lt__(self: Self, rhs: Self) -> Bool`

Returns True if this pointer represents a lower address than rhs.

Args:

- **rhs** (Self): The value of the other pointer.

Returns:

True if this pointer represents a lower address and False otherwise.

__eq__

`__eq__(self: Self, rhs: Self) -> Bool`

Returns True if the two pointers are equal.

Args:

- **rhs** (Self): The value of the other pointer.

Returns:

True if the two pointers are equal and False otherwise.

__ne__

`__ne__(self: Self, rhs: Self) -> Bool`

Returns True if the two pointers are not equal.

Args:

- **rhs** (Self): The value of the other pointer.

Returns:

True if the two pointers are not equal and False otherwise.

__add__

`__add__(self: Self, rhs: Int) -> Self`

Returns a new pointer shifted by the specified offset.

Args:

- **rhs** (Int): The offset.

Returns:

The new DTypePointer shifted by the offset.

__sub__

```
__sub__(self: Self, rhs: Int) -> Self
```

Returns a new pointer shifted back by the specified offset.

Args:

- **rhs** (Int): The offset.

Returns:

The new DTypePointer shifted by the offset.

__iadd__

```
__iadd__(inout self: Self, rhs: Int)
```

Shifts the current pointer by the specified offset.

Args:

- **rhs** (Int): The offset.

__isub__

```
__isub__(inout self: Self, rhs: Int)
```

Shifts back the current pointer by the specified offset.

Args:

- **rhs** (Int): The offset.

get_null

```
get_null() -> Self
```

Constructs a DTypePointer representing *nullptr*.

Returns:

Constructed *nullptr* DTypePointer object.

address_of

```
address_of(inout arg: scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin":_"$dtype"::_DType> type, "value">>) -> Self
```

Gets the address of the argument.

Args:

- **arg** (scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin":_"\$dtype"::_DType> type, "value">>): The value to get the address of.

Returns:

A pointer struct which contains the address of the argument.

alloc

```
alloc(count: Int) -> Self
```

Heap-allocates a number of element of the specified type.

Args:

- **count** (Int): The number of elements to allocate (note that this is not the bytecount).

Returns:

A new DTypePointer object which has been allocated on the heap.

aligned_alloc

```
aligned_alloc(alignment: Int, count: Int) -> Self
```

Heap-allocates a number of element of the specified type using the specified alignment.

Args:

- **alignment** (Int): The alignment used for the allocation.
- **count** (Int): The number of elements to allocate (note that this is not the bytecount).

Returns:

A new DTypePointer object which has been allocated on the heap.

free

```
free(self: Self)
```

Frees the heap allocates memory.

bitcast

```
bitcast[new_type: DType](self: Self) -> DTypePointer[new_type]
```

Bitcasts DTypePointer to a different dtype.

Parameters:

- **new_type** (DType): The target dtype.

Returns:

A new DTypePointer object with the specified dtype and the same address, as the original DTypePointer.

as_scalar_pointer

```
as_scalar_pointer(self: Self) -> Pointer[ SIMD[type, 1] ]
```

Converts the DTypePointer to a scalar pointer of the same dtype.

Returns:

A Pointer to a scalar of the same dtype.

load

```
load(self: Self, offset: Int) -> SIMD[type, 1]
```

Loads a single element (SIMD of size 1) from the pointer at the specified index.

Args:

- **offset** (Int): The offset to load from.

Returns:

The loaded value.

```
load(self: Self) -> SIMD[type, 1]
```

Loads a single element (SIMD of size 1) from the pointer.

Returns:

The loaded value.

prefetch

```
prefetch[params: PrefetchOptions](self: Self)
```

Prefetches memory at the underlying address.

Parameters:

- **params** (PrefetchOptions): Prefetch options (see `PrefetchOptions` for details).

simd_load

```
simd_load[width: Int](self: Self, offset: Int) -> SIMD[type, width]
```

Loads a SIMD vector of elements from the pointer at the specified offset.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **offset** (Int): The offset to load from.

Returns:

The loaded value.

```
simd_load[width: Int](self: Self) -> SIMD[type, width]
```

Loads a SIMD vector of elements from the pointer.

Parameters:

- **width** (Int): The SIMD width.

Returns:

The loaded SIMD value.

aligned_simd_load

```
aligned_simd_load[width: Int, alignment: Int](self: Self, offset: Int) -> SIMD[type, width]
```

Loads a SIMD vector of elements from the pointer at the specified offset with the guaranteed specified alignment.

Parameters:

- **width** (Int): The SIMD width.
- **alignment** (Int): The minimal alignment of the address.

Args:

- **offset** (Int): The offset to load from.

Returns:

The loaded SIMD value.

```
aligned_simd_load[width: Int, alignment: Int](self: Self) -> SIMD[type, width]
```

Loads a SIMD vector of elements from the pointer with the guaranteed specified alignment.

Parameters:

- **width** (Int): The SIMD width.
- **alignment** (Int): The minimal alignment of the address.

Returns:

The loaded SIMD value.

store

```
store(self: Self, offset: Int, val: SIMD[type, 1])
```

Stores a single element value at the given offset.

Args:

- **offset** (Int): The offset to store to.
- **val** (SIMD[type, 1]): The value to store.

```
store(self: Self, val: SIMD[type, 1])
```

Stores a single element value.

Args:

- **val** (SIMD[type, 1]): The value to store.

simd_store

```
simd_store[width: Int](self: Self, offset: Int, val: SIMD[type, width])
```

Stores a SIMD vector at the given offset.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **offset** (Int): The offset to store to.
- **val** (SIMD[type, width]): The SIMD value to store.

```
simd_store[width: Int](self: Self, val: SIMD[type, width])
```

Stores a SIMD vector.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **val** (SIMD[type, width]): The SIMD value to store.

simd_nt_store

```
simd_nt_store[width: Int](self: Self, offset: Int, val: SIMD[type, width])
```

Stores a SIMD vector using non-temporal store.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **offset** (Int): The offset to store to.
- **val** (SIMD[type, width]): The SIMD value to store.

```
simd_nt_store[width: Int](self: Self, val: SIMD[type, width])
```

Stores a SIMD vector using non-temporal store.

The address must be properly aligned, 64B for avx512, 32B for avx2, and 16B for avx.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **val** (SIMD[type, width]): The SIMD value to store.

simd_strided_load

```
simd_strided_load[width: Int](self: Self, stride: Int) -> SIMD[type, width]
```

Performs a strided load of the SIMD vector.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **stride** (Int): The stride between loads.

Returns:

A vector which is stride loaded.

simd_strided_store

```
simd_strided_store[width: Int](self: Self, val: SIMD[type, width], stride: Int)
```

Performs a strided store of the SIMD vector.

Parameters:

- **width** (Int): The SIMD width.

Args:

- **val** (SIMD[type, width]): The SIMD value to store.
- **stride** (Int): The stride between stores.

aligned_simd_store

```
aligned_simd_store[width: Int, alignment: Int](self: Self, offset: Int, val: SIMD[type, width])
```

Stores a SIMD vector at the given offset with a guaranteed alignment.

Parameters:

- **width** (Int): The SIMD width.
- **alignment** (Int): The minimal alignment of the address.

Args:

- **offset** (Int): The offset to store to.
- **val** (SIMD[type, width]): The SIMD value to store.

```
aligned_simd_store[width: Int, alignment: Int](self: Self, val: SIMD[type, width])
```

Stores a SIMD vector with a guaranteed alignment.

Parameters:

- **width** (Int): The SIMD width.
- **alignment** (Int): The minimal alignment of the address.

Args:

- **val** (SIMD[type, width]): The SIMD value to store.

is_aligned

```
is_aligned[alignment: Int](self: Self) -> Bool
```

Checks if the pointer is aligned.

Parameters:

- **alignment** (Int): The minimal desired alignment.

Returns:

True if the pointer is at least alignment-aligned or False otherwise.

offset

```
offset(self: Self, idx: Int) -> Self
```

Returns a new pointer shifted by the specified offset.

Args:

- **idx** (Int): The offset of the new pointer.

Returns:

The new constructed DTypePointer.

bitcast

bitcast[new_type: AnyType, src_type: AnyType](ptr: Pointer[src_type]) -> Pointer[new_type]

Bitcasts a Pointer to a different type.

Parameters:

- **new_type** (AnyType): The target type.
- **src_type** (AnyType): The source type.

Args:

- **ptr** (Pointer[src_type]): The source pointer.

Returns:

A new Pointer with the specified type and the same address, as the original Pointer.

bitcast[new_type: DType, src_type: DType](ptr: DTypePointer[src_type]) -> DTypePointer[new_type]

Bitcasts a DTypePointer to a different type.

Parameters:

- **new_type** (DType): The target type.
- **src_type** (DType): The source type.

Args:

- **ptr** (DTypePointer[src_type]): The source pointer.

Returns:

A new DTypePointer with the specified type and the same address, as the original DTypePointer.

bitcast[new_type: DType, new_width: Int, src_type: DType, src_width: Int](val: SIMD[src_type, src_width]) -> SIMD[new_type, new_width]

Bitcasts a SIMD value to another SIMD value.

Constraints:

The bitwidth of the two types must be the same.

Parameters:

- **new_type** (DType): The target type.
- **new_width** (Int): The target width.
- **src_type** (DType): The source type.
- **src_width** (Int): The source width.

Args:

- **val** (SIMD[src_type, src_width]): The source value.

Returns:

A new SIMD value with the specified type and width with a bitcopy of the source SIMD value.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

atomic

Module

Implements the Atomic class.

You can import these APIs from the `os` package. For example:

```
from os.atomic import Atomic
```

Atomic

Represents a value with atomic operations.

The class provides `atomic.add` and `sub` methods for mutating the value.

Parameters:

- **_type** (DType): DType of the value.

Aliases:

- `type = _21x15__type`
- `scalar_type = scalar<#lit.struct.extract<:_!kgen.declref<@"$builtin"::@"$dtype"::@DType> _type, "value">>`

Fields:

- **value** (scalar<#lit.struct.extract<:_!kgen.declref<@"\$builtin"::@"\$dtype"::@DType> _type, "value">>): The atomic value.

Functions:

__init__

```
__init__(inout self: Self, value: SIMD[_type, 1])
```

Constructs a new atomic value.

Args:

- **value** (SIMD[_type, 1]): Initial value represented as SIMD[type, 1] type.

```
__init__(inout self: Self, value: Int)
```

Constructs a new atomic value.

Args:

- **value** (Int): Initial value represented as mlir.index type.

```
__init__(inout self: Self, value:  
scalar<#lit.struct.extract<:_!kgen.declref<_"$builtin"::_"$dtype"::_DType> _type, "value">>)
```

Constructs a new atomic value.

Args:

- **value** (scalar<#lit.struct.extract<:_!kgen.declref<_"\$builtin"::_"\$dtype"::_DType> _type, "value">>): Initial value represented as pop.scalar type.

iadd

`__iadd__(inout self: Self, rhs: SIMD[_type, 1])`

Performs atomic in-place add.

Atomically replaces the current value with the result of arithmetic addition of the value and arg. That is, it performs atomic post-increment. The operation is a read-modify-write operation. Memory is affected according to the value of order which is sequentially consistent.

Args:

- **rhs** (SIMD[_type, 1]): Value to add.

isub

`__isub__(inout self: Self, rhs: SIMD[_type, 1])`

Performs atomic in-place sub.

Atomically replaces the current value with the result of arithmetic subtraction of the value and arg. That is, it performs atomic post-decrement. The operation is a read-modify-write operation. Memory is affected according to the value of order which is sequentially consistent.

Args:

- **rhs** (SIMD[_type, 1]): Value to subtract.

fetch_add

`fetch_add(inout self: Self, rhs: SIMD[_type, 1]) -> SIMD[_type, 1]`

Performs atomic in-place add.

Atomically replaces the current value with the result of arithmetic addition of the value and arg. That is, it performs atomic post-increment. The operation is a read-modify-write operation. Memory is affected according to the value of order which is sequentially consistent.

Args:

- **rhs** (SIMD[_type, 1]): Value to add.

Returns:

The original value before addition.

fetch_sub

`fetch_sub(inout self: Self, rhs: SIMD[_type, 1]) -> SIMD[_type, 1]`

Performs atomic in-place sub.

Atomically replaces the current value with the result of arithmetic subtraction of the value and arg. That is, it performs atomic post-decrement. The operation is a read-modify-write operation. Memory is affected according to the value of order which is sequentially consistent.

Args:

- **rhs** (SIMD[_type, 1]): Value to subtract.

Returns:

The original value before subtraction.

max

```
max(inout self: Self, rhs: SIMD[_type, 1])
```

Performs atomic in-place max.

Atomically replaces the current value with the result of max of the value and arg. The operation is a read-modify-write operation perform according to sequential consistency semantics.

Constraints:

The input type must be either integral or floating-point type.

Args:

- **rhs** (SIMD[_type, 1]): Value to max.

min

```
min(inout self: Self, rhs: SIMD[_type, 1])
```

Performs atomic in-place min.

Atomically replaces the current value with the result of min of the value and arg. The operation is a read-modify-write operation. The operation is a read-modify-write operation perform according to sequential consistency semantics.

Constraints:

The input type must be either integral or floating-point type.

Args:

- **rhs** (SIMD[_type, 1]): Value to min.

env

Module

Implements basic routines for working with the OS.

You can import these APIs from the `os` package. For example:

```
from os import setenv
```

setenv

```
setenv(name: StringRef, value: StringRef, overwrite: Bool) -> Bool
```

Changes or adds an environment variable.

Constraints:

The function only works on macOS or Linux and returns False otherwise.

Args:

- **name** (StringRef): The name of the environment variable.
- **value** (StringRef): The value of the environment variable.
- **overwrite** (Bool): If an environment variable with the given name already exists, its value is not changed unless `overwrite` is True.

Returns:

False if the name is empty or contains an = character. In any other case, True is returned.

getenv

```
getenv(name: StringRef, default: StringRef) -> StringRef
```

Returns the value of the given environment variable.

Constraints:

The function only works on macOS or Linux and returns an empty string otherwise.

Args:

- **name** (StringRef): The name of the environment variable.
- **default** (StringRef): The default value to return if the environment variable doesn't exist.

Returns:

The value of the environment variable.

```
getenv(name: StringRef) -> StringRef
```

Returns the value of the given environment variable. If the environment variable is not found, then an empty string is returned.

Constraints:

The function only works on macOS or Linux and returns an empty string otherwise.

Args:

- **name** (StringRef): The name of the environment variable.

Returns:

The value of the environment variable.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

path

Module

Aliases:

- DIR_SEPARATOR = cond(apply(:!lit.signature<("self": !kgen.declref<@"\$builtin"::@">\$bool"::@Bool> borrow) -> i1> @"\$builtin"::@"\$bool"::@Bool:@"_mlir_il_(\$builtin:\$bool::Bool)", apply(:!lit.signature<() -> !kgen.declref<@"\$builtin"::@"\$bool"::@Bool>> @"\$sys"::@"\$info"::@"os_is_windows()")), #lit.struct<{value: string = "\\\"}>, #lit.struct<{value: string = "/">})

Path

The Path object.

Fields:

- **path** (String): The underlying path string representation.

Functions:

__init__

__init__(inout self: Self)

Initializes a path with the current directory.

__init__(inout self: Self, path: StringLiteral)

Initializes a path with the provided path.

Args:

- **path** (StringLiteral): The file system path.

__init__(inout self: Self, path: StringRef)

Initializes a path with the provided path.

Args:

- **path** (StringRef): The file system path.

__init__(inout self: Self, path: String)

Initializes a path with the provided path.

Args:

- **path** (String): The file system path.

__copyinit__

__copyinit__(inout self: Self, existing: Self)

Copy constructor for the path struct.

Args:

- **existing** (Self): The existing struct to copy from.

__del__

`__del__(owned self: Self)`

__truediv__

`__truediv__(self: Self, suffix: Self) -> Self`

Joins two paths using the system-defined path separator.

Args:

- **suffix** (Self): The suffix to append to the path.

Returns:

A new path with the suffix appended to the current path.

`__truediv__(self: Self, suffix: StringLiteral) -> Self`

Joins two paths using the system-defined path separator.

Args:

- **suffix** (StringLiteral): The suffix to append to the path.

Returns:

A new path with the suffix appended to the current path.

`__truediv__(self: Self, suffix: StringRef) -> Self`

Joins two paths using the system-defined path separator.

Args:

- **suffix** (StringRef): The suffix to append to the path.

Returns:

A new path with the suffix appended to the current path.

`__truediv__(self: Self, suffix: String) -> Self`

Joins two paths using the system-defined path separator.

Args:

- **suffix** (String): The suffix to append to the path.

Returns:

A new path with the suffix appended to the current path.

__str__

`__str__(self: Self) -> String`

Returns a string representation of the path.

Returns:

A string representation of the path.

__repr__

`__repr__(self: Self) -> String`

Returns a printable representation of the path.

Returns:

A printable representation of the path.

cwd

`cwd() -> Path`

Gets the current directory.

Returns:

The current directory.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

object

Module

Implements PyObject.

You can import these APIs from the `python` package. For example:

```
from python.object import PyObject
```

PythonObject

A Python object.

Fields:

- **py_object** (PyObjectPtr): A pointer to the underlying Python object.

Functions:

`__init__`

```
__init__(inout self: Self)
```

Initialize the object with a `None` value.

```
__init__(inout self: Self, none: None)
```

Initialize a `None` value object from a `None` literal.

Args:

- **none** (`None`): `None`.

```
__init__(inout self: Self, integer: Int)
```

Initialize the object with an integer value.

Args:

- **integer** (`Int`): The integer value.

```
__init__(inout self: Self, float: FloatLiteral)
```

Initialize the object with an floating-point value.

Args:

- **float** (`FloatLiteral`): The float value.

```
__init__[dt: DType](inout self: Self, value: SIMD[dt, 1])
```

Initialize the object with a generic scalar value. If the scalar value type is `bool`, it is converted to a `boolean`. Otherwise, it is converted to the appropriate integer or floating point type.

Parameters:

- **dt** (`DType`): The scalar value type.

Args:

- **value** (`SIMD[dt, 1]`): The scalar value.

`__init__(inout self: Self, value: Bool)`

Initialize the object from a bool.

Args:

- **value** (Bool): The boolean value.

`__init__(inout self: Self, str: StringLiteral)`

Initialize the object from a string literal.

Args:

- **str** (StringLiteral): The string value.

`__init__(inout self: Self, str: StringRef)`

Initialize the object from a string reference.

Args:

- **str** (StringRef): The string value.

`__init__(inout self: Self, str: String)`

Initialize the object from a string.

Args:

- **str** (String): The string value.

`__init__[*Ts: AnyType](inout self: Self, value: ListLiteral[Ts])`

Initialize the object from a list literal.

Parameters:

- **Ts** (*AnyType): The list element types.

Args:

- **value** (ListLiteral[Ts]): The list value.

`__init__[*Ts: AnyType](inout self: Self, value: Tuple[Ts])`

Initialize the object from a tuple literal.

Parameters:

- **Ts** (*AnyType): The tuple element types.

Args:

- **value** (Tuple[Ts]): The tuple value.

`__init__(inout self: Self, py_object: PyObjectPtr)`

`__copyinit__`

`__copyinit__(inout self: Self, existing: Self)`

Copy the object.

This increments the underlying refcount of the existing object.

Args:

- **existing** (`Self`): The value to copy.

`__moveinit__`

```
__moveinit__(inout self: Self, owned existing: Self)
```

`__del__`

```
__del__(owned self: Self)
```

Destroy the object.

This decrements the underlying refcount of the pointed-to object.

`__bool__`

```
__bool__(self: Self) -> Bool
```

Evaluate the boolean value of the object.

Returns:

Whether the object evaluates as true.

`__getitem__`

```
__getitem__(self: Self, *args: Self) -> Self
```

Return the value for the given key or keys.

Args:

- **args** (*`Self`): The key or keys to access on this object.

Returns:

The value corresponding to the given key for this object.

`__neg__`

```
__neg__(self: Self) -> Self
```

Negative.

Calls the underlying object's `__neg__` method.

Returns:

The result of prefixing this object with a - operator. For most numerical objects, this returns the negative.

`__pos__`

```
__pos__(self: Self) -> Self
```

Positive.

Calls the underlying object's `__pos__` method.

Returns:

The result of prefixing this object with a + operator. For most numerical objects, this does nothing.

__invert__

__invert__(self: Self) -> Self

Inversion.

Calls the underlying object's __invert__ method.

Returns:

The logical inverse of this object: a bitwise representation where all bits are flipped, from zero to one, and from one to zero.

__lt__

__lt__(self: Self, rhs: Self) -> Self

Less than comparator. This lexicographically compares strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the object is less than the right hard argument.

__le__

__le__(self: Self, rhs: Self) -> Self

Less than or equal to comparator. This lexicographically compares strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the object is less than or equal to the right hard argument.

__eq__

__eq__(self: Self, rhs: Self) -> Self

Equality comparator. This compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the objects are equal.

__ne__

__ne__(self: Self, rhs: Self) -> Self

Inequality comparator. This compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the objects are not equal.

__gt__

__gt__(self: Self, rhs: Self) -> Self

Greater than comparator. This lexicographically compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the left hand value is greater.

__ge__

__ge__(self: Self, rhs: Self) -> Self

Greater than or equal to comparator. This lexicographically compares the elements of strings and lists.

Args:

- **rhs** (Self): Right hand value.

Returns:

True if the left hand value is greater than or equal to the right hand value.

__add__

__add__(self: Self, rhs: Self) -> Self

Addition and concatenation.

Calls the underlying object's __add__ method.

Args:

- **rhs** (Self): Right hand value.

Returns:

The sum or concatenated values.

__sub__

__sub__(self: Self, rhs: Self) -> Self

Subtraction.

Calls the underlying object's __sub__ method.

Args:

- **rhs** (Self): Right hand value.

Returns:

The difference.

__mul__

__mul__(self: Self, rhs: Self) -> Self

Multiplication.

Calls the underlying object's __mul__ method.

Args:

- **rhs** (Self): Right hand value.

Returns:

The product.

__truediv__

__truediv__(self: Self, rhs: Self) -> Self

Division.

Calls the underlying object's __truediv__ method.

Args:

- **rhs** (Self): The right-hand-side value by which this object is divided.

Returns:

The result of dividing the right-hand-side value by this.

__floordiv__

__floordiv__(self: Self, rhs: Self) -> Self

Return the division of self and rhs rounded down to the nearest integer.

Calls the underlying object's __floordiv__ method.

Args:

- **rhs** (Self): The right-hand-side value by which this object is divided.

Returns:

The result of dividing this by the right-hand-side value, modulo any remainder.

__mod__

__mod__(self: Self, rhs: Self) -> Self

Return the remainder of self divided by rhs.

Calls the underlying object's __mod__ method.

Args:

- **rhs** (Self): The value to divide on.

Returns:

The remainder of dividing self by rhs.

__pow__

`__pow__(self: Self, rhs: Self) -> Self`

Raises this object to the power of the given value.

Args:

- **rhs** (Self): The exponent.

Returns:

The result of raising this by the given exponent.

__lshift__

`__lshift__(self: Self, rhs: Self) -> Self`

Bitwise left shift.

Args:

- **rhs** (Self): The right-hand-side value by which this object is bitwise shifted to the left.

Returns:

This value, shifted left by the given value.

__rshift__

`__rshift__(self: Self, rhs: Self) -> Self`

Bitwise right shift.

Args:

- **rhs** (Self): The right-hand-side value by which this object is bitwise shifted to the right.

Returns:

This value, shifted right by the given value.

__and__

`__and__(self: Self, rhs: Self) -> Self`

Bitwise AND.

Args:

- **rhs** (Self): The right-hand-side value with which this object is bitwise AND'ed.

Returns:

The bitwise AND result of this and the given value.

__or__

`__or__(self: Self, rhs: Self) -> Self`

Bitwise OR.

Args:

- **rhs** (`Self`): The right-hand-side value with which this object is bitwise OR'ed.

Returns:

The bitwise OR result of this and the given value.

`__xor__`

`__xor__(self: Self, rhs: Self) -> Self`

Exclusive OR.

Args:

- **rhs** (`Self`): The right-hand-side value with which this object is exclusive OR'ed.

Returns:

The exclusive OR result of this and the given value.

`__radd__`

`__radd__(self: Self, lhs: Self) -> Self`

Reverse addition and concatenation.

Calls the underlying object's `__radd__` method.

Args:

- **lhs** (`Self`): The left-hand-side value to which this object is added or concatenated.

Returns:

The sum.

`__rsub__`

`__rsub__(self: Self, lhs: Self) -> Self`

Reverse subtraction.

Calls the underlying object's `__rsub__` method.

Args:

- **lhs** (`Self`): The left-hand-side value from which this object is subtracted.

Returns:

The result of subtracting this from the given value.

`__rmul__`

`__rmul__(self: Self, lhs: Self) -> Self`

Reverse multiplication.

Calls the underlying object's `__rmul__` method.

Args:

- **Lhs** (`Self`): The left-hand-side value that is multiplied by this object.

Returns:

The product of the multiplication.

`__rtruediv__`

`__rtruediv__`(`self: Self, lhs: Self`) -> `Self`

Reverse division.

Calls the underlying object's `__rtruediv__` method.

Args:

- **Lhs** (`Self`): The left-hand-side value that is divided by this object.

Returns:

The result of dividing the given value by this.

`__rfloordiv__`

`__rfloordiv__`(`self: Self, lhs: Self`) -> `Self`

Reverse floor division.

Calls the underlying object's `__rfloordiv__` method.

Args:

- **Lhs** (`Self`): The left-hand-side value that is divided by this object.

Returns:

The result of dividing the given value by this, modulo any remainder.

`__rmod__`

`__rmod__`(`self: Self, lhs: Self`) -> `Self`

Reverse modulo.

Calls the underlying object's `__rmod__` method.

Args:

- **Lhs** (`Self`): The left-hand-side value that is divided by this object.

Returns:

The remainder from dividing the given value by this.

`__rpow__`

`__rpow__`(`self: Self, lhs: Self`) -> `Self`

Reverse power of.

Args:

- **lhs** (Self): The number that is raised to the power of this object.

Returns:

The result of raising the given value by this exponent.

`__rlshift__`

`__rlshift__(self: Self, lhs: Self) -> Self`

Reverse bitwise left shift.

Args:

- **lhs** (Self): The left-hand-side value that is bitwise shifted to the left by this object.

Returns:

The given value, shifted left by this.

`__rrshift__`

`__rrshift__(self: Self, lhs: Self) -> Self`

Reverse bitwise right shift.

Args:

- **lhs** (Self): The left-hand-side value that is bitwise shifted to the right by this object.

Returns:

The given value, shifted right by this.

`__rand__`

`__rand__(self: Self, lhs: Self) -> Self`

Reverse bitwise AND.

Args:

- **lhs** (Self): The left-hand-side value that is bitwise AND'ed with this object.

Returns:

The bitwise AND result of the given value and this.

`__ror__`

`__ror__(self: Self, lhs: Self) -> Self`

Reverse bitwise OR.

Args:

- **lhs** (Self): The left-hand-side value that is bitwise OR'ed with this object.

Returns:

The bitwise OR result of the given value and this.

__rxor__

__rxor__(self: Self, lhs: Self) -> Self

Reverse exclusive OR.

Args:

- **lhs** (Self): The left-hand-side value that is exclusive OR'ed with this object.

Returns:

The exclusive OR result of the given value and this.

__iadd__

__iadd__(inout self: Self, rhs: Self)

Immediate addition and concatenation.

Args:

- **rhs** (Self): The right-hand-side value that is added to this object.

__isub__

__isub__(inout self: Self, rhs: Self)

Immediate subtraction.

Args:

- **rhs** (Self): The right-hand-side value that is subtracted from this object.

__imul__

__imul__(inout self: Self, rhs: Self)

In-place multiplication.

Calls the underlying object's __imul__ method.

Args:

- **rhs** (Self): The right-hand-side value by which this object is multiplied.

__itruediv__

__itruediv__(inout self: Self, rhs: Self)

Immediate division.

Args:

- **rhs** (Self): The value by which this object is divided.

__ifloordiv__

__ifloordiv__(inout self: Self, rhs: Self)

Immediate floor division.

Args:

- **rhs** (Self): The value by which this object is divided.

__imod__

__imod__(inout self: Self, rhs: Self)

Immediate modulo.

Args:

- **rhs** (Self): The right-hand-side value that is used to divide this object.

__ipow__

__ipow__(inout self: Self, rhs: Self)

Immediate power of.

Args:

- **rhs** (Self): The exponent.

__ilshift__

__ilshift__(inout self: Self, rhs: Self)

Immediate bitwise left shift.

Args:

- **rhs** (Self): The right-hand-side value by which this object is bitwise shifted to the left.

__irshift__

__irshift__(inout self: Self, rhs: Self)

Immediate bitwise right shift.

Args:

- **rhs** (Self): The right-hand-side value by which this object is bitwise shifted to the right.

__iand__

__iand__(inout self: Self, rhs: Self)

Immediate bitwise AND.

Args:

- **rhs** (Self): The right-hand-side value with which this object is bitwise AND'ed.

__ixor__

__ixor__(inout self: Self, rhs: Self)

Immediate exclusive OR.

Args:

- **rhs** (Self): The right-hand-side value with which this object is exclusive OR'ed.

__ior__

`__ior__(inout self: Self, rhs: Self)`

Immediate bitwise OR.

Args:

- **rhs** (`Self`): The right-hand-side value with which this object is bitwise OR'ed.

`__iter__`

`__iter__(inout self: Self) -> _PyIter`

Iterate over object if supported.

Returns:

An iterator object.

`__getattr__`

`__getattr__(self: Self, name: StringLiteral) -> Self`

Return the value of the object attribute with the given name.

Args:

- **name** (`StringLiteral`): The name of the object attribute to return.

Returns:

The value of the object attribute with the given name.

`__setattr__`

`__setattr__(self: Self, name: StringLiteral, newValue: Self)`

Set the given value for the object attribute with the given name.

Args:

- **name** (`StringLiteral`): The name of the object attribute to set.
- **newValue** (`Self`): The new value to be set for that attribute.

`__setattr__(self: Self, name: StringLiteral, newValue: Dictionary)`

Set the given dict for the object attribute with the given name.

Args:

- **name** (`StringLiteral`): The name of the object attribute to set.
- **newValue** (`Dictionary`): The new dict value to be set for that attribute.

`__call__`

`__call__(self: Self, *args: Self) -> Self`

Call the underlying object as if it were a function.

Returns:

The return value from the called object.

`to_float64`

`to_float64(self: Self) -> SIMD[f64, 1]`

Returns a float representation of the object.

Returns:

A floating point value that represents this object.

__index__

`__index__(self: Self) -> Int`

Returns an index representation of the object.

Returns:

An index value that represents this object.

to_string

`to_string(self: Self) -> String`

Returns a string representation of the object.

Calls the underlying object's `__str__` method.

Returns:

A string that represents this object.

python

Module

Implements Python interoperability.

You can import these APIs from the `python` package. For example:

```
from python import Python
```

Python

Provides methods that help you use Python code in Mojo.

Fields:

- **impl** (`_PythonInterfaceImpl`): The underlying implementation of Mojo's Python interface.

Functions:

`__init__`

```
__init__(inout self: Self)
```

Default constructor.

`__copyinit__`

```
__copyinit__(inout self: Self, existing: Self)
```

Copy constructor.

Args:

- **existing** (`Self`): The existing instance to copy from.

`eval`

```
eval(inout self: Self, str: StringRef) -> Bool
```

Executes the given Python code.

Args:

- **str** (`StringRef`): The python code to execute.

Returns:

True if the code executed successfully or `False` if the code raised an exception.

`evaluate`

```
evaluate(str: StringRef) -> PythonObject
```

Executes the given Python code.

Args:

- **str** (`StringRef`): The Python expression to evaluate.

Returns:

PythonObject containing the result of the evaluation.

add_to_path

```
add_to_path(str: StringRef)
```

Adds a directory to the Python path.

This might be necessary to import a Python module via `import_module()`. For example:

```
from python import Python  
  
# Specify path to `mypython.py` module  
Python.add_to_path("path/to/module")  
let mypython = Python.import_module("mypython")  
  
let c = mypython.my_algorithm(2, 3)___
```

Args:

- **str** (StringRef): The path to a Python module you want to import.

import_module

```
import_module(str: StringRef) -> PythonObject
```

Imports a Python module.

This provides you with a module object you can use just like you would in Python. For example:

```
from python import Python  
  
# This is equivalent to Python's `import numpy as np`  
let np = Python.import_module("numpy")  
a = np.array([1, 2, 3])___
```

Args:

- **str** (StringRef): The Python module name. This module must be visible from the list of available Python paths (you might need to add the module's path with `add_to_path()`).

Returns:

The Python module.

dict

```
dict() -> Dictionary
```

Construct an empty Python dictionary.

Returns:

The constructed empty Python dictionary.

__str__

```
__str__(inout self: Self, str: PythonObject) -> StringRef
```

Return a string representing the given Python object.

This function allows to convert Python objects to Mojo string type.

Returns:

Mojo string representing the given Python object.

throw_python_exception_if_error_state

```
throw_python_exception_if_error_state(inout cpython: CPython)
```

Raise an exception if CPython interpreter is in an error state.

Args:

- **cpython** (CPython): The cpython instance we wish to error check.

is_type

```
is_type(x: PythonObject, y: PythonObject) -> Bool
```

Test if the x object is the y object, the same as x is y in Python.

Args:

- **x** (PythonObject): The left-hand-side value in the comparison.
- **y** (PythonObject): The right-hand-side type value in the comparison.

Returns:

True if x and y are the same object and False otherwise.

type

```
type(obj: PythonObject) -> PythonObject
```

Return Type of this PyObject.

Args:

- **obj** (PythonObject): PyObject we want the type of.

Returns:

A PyObject that holds the type object.

none

```
none() -> PythonObject
```

Get a PyObject representing None.

Returns:

PyObject representing None.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

random

Module

Provides functions for random numbers.

You can import these APIs from the `random` package. For example:

```
from random import seed
```

seed

```
seed()
```

Seeds the random number generator using the current time.

```
seed(a: Int)
```

Seeds the random number generator using the value provided.

Args:

- **a** (Int): The seed value.

random_float64

```
random_float64(min: SIMD[f64, 1], max: SIMD[f64, 1]) -> SIMD[f64, 1]
```

Returns a random `Float64` number from the given range.

Args:

- **min** (`SIMD[f64, 1]`): The minimum number in the range (default is 0.0).
- **max** (`SIMD[f64, 1]`): The maximum number in the range (default is 1.0).

Returns:

A random number from the specified range.

random_si64

```
random_si64(min: SIMD[si64, 1], max: SIMD[si64, 1]) -> SIMD[si64, 1]
```

Returns a random `Int64` number from the given range.

Args:

- **min** (`SIMD[si64, 1]`): The minimum number in the range.
- **max** (`SIMD[si64, 1]`): The maximum number in the range.

Returns:

A random number from the specified range.

random_ui64

```
random_ui64(min: SIMD[ui64, 1], max: SIMD[ui64, 1]) -> SIMD[ui64, 1]
```

Returns a random `UInt64` number from the given range.

Args:

- **min** (SIMD[ui64, 1]): The minimum number in the range.
- **max** (SIMD[ui64, 1]): The maximum number in the range.

Returns:

A random number from the specified range.

randint

```
randint[type: DType](ptr: DTypePointer[type], size: Int, low: Int, high: Int)
```

Fills memory with uniform random in range [low, high].

Constraints:

The type should be integral.

Parameters:

- **type** (DType): The dtype of the pointer.

Args:

- **ptr** (DTypePointer[type]): The pointer to the memory area to fill.
- **size** (Int): The number of elements to fill.
- **low** (Int): The minimal value for random.
- **high** (Int): The maximal value for random.

rand

```
rand[type: DType](ptr: DTypePointer[type], size: Int)
```

Fills memory with random values from a uniform distribution.

Parameters:

- **type** (DType): The dtype of the pointer.

Args:

- **ptr** (DTypePointer[type]): The pointer to the memory area to fill.
- **size** (Int): The number of elements to fill.

```
rand[type: DType](*shape: Int) -> Tensor[type]
```

Constructs a new tensor with the specified shape and fills it with random elements.

Parameters:

- **type** (DType): The dtype of the tensor.

Args:

- **shape** (*Int): The tensor shape.

Returns:

A new tensor of specified shape and filled with random elements.

```
rand[type: DType](owned shape: TensorShape) -> Tensor[type]
```

Constructs a new tensor with the specified shape and fills it with random elements.

Parameters:

- **type** (DType): The dtype of the tensor.

Args:

- **shape** (TensorShape): The tensor shape.

Returns:

A new tensor of specified shape and filled with random elements.

```
rand[type: DType](owned spec: TensorSpec) -> Tensor[type]
```

Constructs a new tensor with the specified specification and fills it with random elements.

Parameters:

- **type** (DType): The dtype of the tensor.

Args:

- **spec** (TensorSpec): The tensor specification.

Returns:

A new tensor of specified specification and filled with random elements.

randn_float64

```
randn_float64(mean: SIMD[f64, 1], variance: SIMD[f64, 1]) -> SIMD[f64, 1]
```

Returns a random double sampled from Normal(mean, variance) distribution.

Args:

- **mean** (SIMD[f64, 1]): Normal distribution mean.
- **variance** (SIMD[f64, 1]): Normal distribution variance.

Returns:

A random float64 sampled from Normal(mean, variance).

randn

```
randn[type: DType](ptr: DTypePointer[type], size: Int, mean: SIMD[f64, 1], variance: SIMD[f64, 1])
```

Fills memory with random values from a Normal(mean, variance) distribution.

Constraints:

The type should be floating point.

Parameters:

- **type** (DType): The dtype of the pointer.

Args:

- **ptr** (DTypePointer[type]): The pointer to the memory area to fill.
- **size** (Int): The number of elements to fill.

- **mean** (SIMD[f64, 1]): Normal distribution mean.
- **variance** (SIMD[f64, 1]): Normal distribution variance.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

arg

Module

Implements functions and variables for interacting with execution and system environment.

You can import these APIs from the `sys` package. For example:

```
from sys import argv
```

argv

`argv() -> VariadicList[StringRef]`

The list of command line arguments.

Returns:

The list of command line arguments provided when mojo was invoked.

info

Module

Implements methods for querying the host target info.

You can import these APIs from the `sys` package. For example:

```
from sys.info import is_x86
```

is_x86

```
is_x86() -> Bool
```

Returns True if the host system architecture is X86 and False otherwise.

Returns:

True if the host system architecture is X86 and False otherwise.

has_sse4

```
has_sse4() -> Bool
```

Returns True if the host system has sse4, otherwise returns False.

Returns:

True if the host system has sse4, otherwise returns False.

has_avx

```
has_avx() -> Bool
```

Returns True if the host system has AVX, otherwise returns False.

Returns:

True if the host system has AVX, otherwise returns False.

has_avx2

```
has_avx2() -> Bool
```

Returns True if the host system has AVX2, otherwise returns False.

Returns:

True if the host system has AVX2, otherwise returns False.

has_avx512f

```
has_avx512f() -> Bool
```

Returns True if the host system has AVX512, otherwise returns False.

Returns:

True if the host system has AVX512, otherwise returns False.

has_avx512_vnni

`has_avx512_vnni() -> Bool`

Returns True if the host system has avx512_vnni, otherwise returns False.

Returns:

True if the host system has avx512_vnni, otherwise returns False.

has_neon

`has_neon() -> Bool`

Returns True if the host system has Neon support, otherwise returns False.

Returns:

True if the host system support the Neon instruction set.

is_apple_m1

`is_apple_m1() -> Bool`

Returns True if the host system is an Apple M1 with AMX support, otherwise returns False.

Returns:

True if the host system is an Apple M1 with AMX support and False otherwise.

is_neoverse_n1

`is_neoverse_n1() -> Bool`

Returns True if the host system is a Neoverse N1 system, otherwise returns False.

Returns:

True if the host system is a Neoverse N1 system and False otherwise.

has_intel_amx

`has_intel_amx() -> Bool`

Returns True if the host system has Intel AMX support, otherwise returns False.

Returns:

True if the host system has Intel AMX and False otherwise.

os_is_macos

`os_is_macos() -> Bool`

Returns True if the host operating system is macOS.

Returns:

True if the host operating system is macOS and False otherwise.

os_is_linux

`os_is_linux() -> Bool`

Returns True if the host operating system is Linux.

Returns:

True if the host operating system is Linux and False otherwise.

os_is_windows

`os_is_windows() -> Bool`

Returns True if the host operating system is Windows.

Returns:

True if the host operating system is Windows and False otherwise.

is_triple

`is_triple[triple: StringLiteral]() -> Bool`

Returns True if the target triple of the compiler matches the input and False otherwise.

Parameters:

- **triple** (StringLiteral): The triple value to be checked against.

Returns:

True if the triple matches and False otherwise.

triple_is_nvidia_cuda

`triple_is_nvidia_cuda() -> Bool`

Returns True if the target triple of the compiler is nvptx64-nvidia-cuda False otherwise.

Returns:

True if the triple target is cuda and False otherwise.

simdbitwidth

`simdbitwidth() -> Int`

Returns the vector size (in bits) of the host system.

Returns:

The vector size (in bits) of the host system.

is_little_endian

`is_little_endian() -> Bool`

Returns True if the host endianness is little and False otherwise.

Returns:

True if the host target is little endian and False otherwise.

is_big_endian

`is_big_endian() -> Bool`

Returns True if the host endianness is big and False otherwise.

Returns:

True if the host target is big endian and False otherwise.

simd_byte_width

`simd_byte_width() -> Int`

Returns the vector size (in bytes) of the host system.

Returns:

The vector size (in bytes) of the host system.

sizeof

`sizeof[type: AnyType]() -> Int`

Returns the size of (in bytes) of the type.

Parameters:

- **type** (AnyType): The type in question.

Returns:

The size of the type in bytes.

`sizeof[type: DType]() -> Int`

Returns the size of (in bytes) of the dtype.

Parameters:

- **type** (DType): The DType in question.

Returns:

The size of the dtype in bytes.

alignof

`alignof[type: AnyType]() -> Int`

Returns the align of (in bytes) of the type.

Parameters:

- **type** (AnyType): The type in question.

Returns:

The alignment of the type in bytes.

`alignof[type: DType]() -> Int`

Returns the align of (in bytes) of the dtype.

Parameters:

- **type** (`DType`): The `DType` in question.

Returns:

The alignment of the dtype in bytes.

bitwidthof

`bitwidthof[type: AnyType]() -> Int`

Returns the size of (in bits) of the type.

Parameters:

- **type** (`AnyType`): The type in question.

Returns:

The size of the type in bits.

`bitwidthof[type: DType]() -> Int`

Returns the size of (in bits) of the dtype.

Parameters:

- **type** (`DType`): The type in question.

Returns:

The size of the dtype in bits.

simdwidthof

`simdwidthof[type: AnyType]() -> Int`

Returns the vector size of the type on the host system.

Parameters:

- **type** (`AnyType`): The type in question.

Returns:

The vector size of the type on the host system.

`simdwidthof[type: DType]() -> Int`

Returns the vector size of the type on the host system.

Parameters:

- **type** (`DType`): The `DType` in question.

Returns:

The vector size of the dtype on the host system.

intrinsics

Module

Defines intrinsics.

You can import these APIs from the `complex` package. For example:

```
from sys.intrinsics import PrefetchLocality
```

PrefetchLocality

The prefetch locality.

The locality, rw, and cache type correspond to LLVM prefetch intrinsic's inputs (see [LLVM prefetch locality](#))

Aliases:

- `NONE` = `__init__(0)`: No locality.
- `LOW` = `__init__(1)`: Low locality.
- `MEDIUM` = `__init__(2)`: Medium locality.
- `HIGH` = `__init__(3)`: Extremely local locality (keep in cache).

Fields:

- **value** (`SIMD[si32, 1]`): The prefetch locality to use. It should be a value in [0, 3].

Functions:

`__init__`

```
__init__(value: Int) -> Self
```

Constructs a prefetch locality option.

Args:

- **value** (`Int`): An integer value representing the locality. Should be a value in the range [0, 3].

Returns:

The prefetch locality constructed.

PrefetchRW

Prefetch read or write.

Aliases:

- `READ` = `__init__(0)`: Read prefetch.
- `WRITE` = `__init__(1)`: Write prefetch.

Fields:

- **value** (`SIMD[si32, 1]`): The read-write prefetch. It should be in [0, 1].

Functions:

__init__

`__init__(value: Int) -> Self`

Constructs a prefetch read-write option.

Args:

- **value** (Int): An integer value representing the prefetch read-write option to be used. Should be a value in the range [0, 1].

Returns:

The prefetch read-write option constructed.

PrefetchCache

Prefetch cache type.

Aliases:

- INSTRUCTION = `__init__(0)`: The instruction prefetching option.
- DATA = `__init__(1)`: The data prefetching option.

Fields:

- **value** (SIMD[si32, 1]): The cache prefetch. It should be in [0, 1].

Functions:

__init__

`__init__(value: Int) -> Self`

Constructs a prefetch option.

Args:

- **value** (Int): An integer value representing the prefetch cache option to be used. Should be a value in the range [0, 1].

Returns:

The prefetch cache type that was constructed.

PrefetchOptions

Collection of configuration parameters for a prefetch intrinsic call.

The op configuration follows similar interface as LLVM intrinsic prefetch op, with a “locality” attribute that specifies the level of temporal locality in the application, that is, how soon would the same data be visited again. Possible locality values are: NONE, LOW, MEDIUM, and HIGH.

The op also takes a “cache tag” attribute giving hints on how the prefetched data will be used. Possible tags are: ReadICache, ReadDCache and WriteDCache.

Note: the actual behavior of the prefetch op and concrete interpretation of these attributes are target-dependent.

Fields:

- **rw** (PrefetchRW): Indicates prefetching for read or write.
- **locality** (PrefetchLocality): Indicates locality level.
- **cache** (PrefetchCache): Indicates i-cache or d-cache prefetching.

Functions:

`__init__`

```
__init__() -> Self
```

Constructs an instance of PrefetchOptions with default params.

Returns:

The Prefetch configuration constructed.

`for_read`

```
for_read(self: Self) -> Self
```

Sets the prefetch purpose to read.

Returns:

The updated prefetch parameter.

`for_write`

```
for_write(self: Self) -> Self
```

Sets the prefetch purpose to write.

Returns:

The updated prefetch parameter.

`no_locality`

```
no_locality(self: Self) -> Self
```

Sets the prefetch locality to none.

Returns:

The updated prefetch parameter.

`low_locality`

```
low_locality(self: Self) -> Self
```

Sets the prefetch locality to low.

Returns:

The updated prefetch parameter.

`medium_locality`

```
medium_locality(self: Self) -> Self
```

Sets the prefetch locality to medium.

Returns:

The updated prefetch parameter.

high_locality

```
high_locality(self: Self) -> Self
```

Sets the prefetch locality to high.

Returns:

The updated prefetch parameter.

to_data_cache

```
to_data_cache(self: Self) -> Self
```

Sets the prefetch target to data cache.

Returns:

The updated prefetch parameter.

to_instruction_cache

```
to_instruction_cache(self: Self) -> Self
```

Sets the prefetch target to instruction cache.

Returns:

The updated prefetch parameter.

llvm_intrinsic

```
llvm_intrinsic[intrin: StringLiteral, type: AnyType]() -> *"type"
```

Calls an LLVM intrinsic with no arguments.

Calls an LLVM intrinsic with the name intrin and return type type.

Parameters:

- **intrin** (StringLiteral): The name of the llvm intrinsic.
- **type** (AnyType): The return type of the intrinsic.

Returns:

The result of calling the llvm intrinsic with no arguments.

```
llvm_intrinsic[intrin: StringLiteral, type: AnyType, T0: AnyType](arg0: T0) -> *"type"
```

Calls an LLVM intrinsic with one argument.

Calls the intrinsic with the name intrin and return type type on argument arg0.

Parameters:

- **intrin** (StringLiteral): The name of the llvm intrinsic.
- **type** (AnyType): The return type of the intrinsic.

- **T0** (AnyType): The type of the first argument to the intrinsic (arg0).

Args:

- **arg0** (T0): The argument to call the LLVM intrinsic with. The type of arg0 must be T0.

Returns:

The result of calling the llvm intrinsic with arg0 as an argument.

```
llvm_intrinsic[intrin: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType](arg0: T0, arg1: T1)
-> *"type"
```

Calls an LLVM intrinsic with two arguments.

Calls the LLVM intrinsic with the name intrin and return type type on arguments arg0 and arg1.

Parameters:

- **intrin** (StringLiteral): The name of the llvm intrinsic.
- **type** (AnyType): The return type of the intrinsic.
- **T0** (AnyType): The type of the first argument to the intrinsic (arg0).
- **T1** (AnyType): The type of the second argument to the intrinsic (arg1).

Args:

- **arg0** (T0): The first argument to call the LLVM intrinsic with. The type of arg0 must be T0.
- **arg1** (T1): The second argument to call the LLVM intrinsic with. The type of arg1 must be T1.

Returns:

The result of calling the llvm intrinsic with arg0 and arg1 as arguments.

```
llvm_intrinsic[intrin: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType, T2: AnyType](arg0: T0,
arg1: T1, arg2: T2) -> *"type"
```

Calls an LLVM intrinsic with three arguments.

Calls the LLVM intrinsic with the name intrin and return type type on arguments arg0, arg1 and arg2.

Parameters:

- **intrin** (StringLiteral): The name of the llvm intrinsic.
- **type** (AnyType): The return type of the intrinsic.
- **T0** (AnyType): The type of the first argument to the intrinsic (arg0).
- **T1** (AnyType): The type of the second argument to the intrinsic (arg1).
- **T2** (AnyType): The type of the third argument to the intrinsic (arg2).

Args:

- **arg0** (T0): The first argument to call the LLVM intrinsic with. The type of arg0 must be T0.
- **arg1** (T1): The second argument to call the LLVM intrinsic with. The type of arg1 must be T1.
- **arg2** (T2): The third argument to call the LLVM intrinsic with. The type of arg2 must be T2.

Returns:

The result of calling the llvm intrinsic with arg0, arg1 and arg2 as arguments.

```
llvm_intrinsic[intrin: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType, T2: AnyType, T3: AnyType](arg0: T0, arg1: T1, arg2: T2, arg3: T3) -> *"type"
```

Calls an LLVM intrinsic with four arguments.

Calls the LLVM intrinsic with the name intrin and return type type on arguments arg0, arg1, arg2 and arg3.

Parameters:

- **intrin** (StringLiteral): The name of the llvm intrinsic.
- **type** (AnyType): The return type of the intrinsic.
- **T0** (AnyType): The type of the first argument to the intrinsic (arg0).
- **T1** (AnyType): The type of the second argument to the intrinsic (arg1).
- **T2** (AnyType): The type of the third argument to the intrinsic (arg2).
- **T3** (AnyType): The type of the fourth argument to the intrinsic (arg3).

Args:

- **arg0** (T0): The first argument to call the LLVM intrinsic with. The type of arg0 must be T0.
- **arg1** (T1): The second argument to call the LLVM intrinsic with. The type of arg1 must be T1.
- **arg2** (T2): The third argument to call the LLVM intrinsic with. The type of arg2 must be T2.
- **arg3** (T3): The fourth argument to call the LLVM intrinsic with. The type of arg3 must be T3.

Returns:

The result of calling the llvm intrinsic with arg0, arg1, arg2 and arg3 as arguments.

```
llvm_intrinsic[intrin: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType, T2: AnyType, T3: AnyType, T4: AnyType](arg0: T0, arg1: T1, arg2: T2, arg3: T3, arg4: T4) -> *"type"
```

Calls an LLVM intrinsic with five arguments.

Calls the LLVM intrinsic with the name intrin and return type type on arguments arg0, arg1, arg2, arg3 and arg4.

Parameters:

- **intrin** (StringLiteral): The name of the llvm intrinsic.
- **type** (AnyType): The return type of the intrinsic.
- **T0** (AnyType): The type of the first argument to the intrinsic (arg0).
- **T1** (AnyType): The type of the second argument to the intrinsic (arg1).
- **T2** (AnyType): The type of the third argument to the intrinsic (arg2).
- **T3** (AnyType): The type of the fourth argument to the intrinsic (arg3).
- **T4** (AnyType): The type of the fifth argument to the intrinsic (arg4).

Args:

- **arg0** (T0): The first argument to call the LLVM intrinsic with. The type of arg0 must be T0.
- **arg1** (T1): The second argument to call the LLVM intrinsic with. The type of arg1 must be T1.
- **arg2** (T2): The third argument to call the LLVM intrinsic with. The type of arg2 must be T2.
- **arg3** (T3): The fourth argument to call the LLVM intrinsic with. The type of arg3 must be T3.
- **arg4** (T4): The fourth argument to call the LLVM intrinsic with. The type of arg4 must be T4.

Returns:

The result of calling the llvm intrinsic with arg0, arg1, arg2, arg3 and arg4 as arguments.

external_call

```
external_call[callee: StringLiteral, type: AnyType]() -> *"type"
```

Calls an external function.

Parameters:

- **callee** (StringLiteral): The name of the external function.

- **type** (AnyType): The return type.

Returns:

The external call result.

```
external_call[callee: StringLiteral, type: AnyType, T0: AnyType](arg0: T0) -> *"type"
```

Calls an external function.

Parameters:

- **callee** (StringLiteral): The name of the external function.
- **type** (AnyType): The return type.
- **T0** (AnyType): The first argument type.

Args:

- **arg0** (T0): The first argument.

Returns:

The external call result.

```
external_call[callee: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType](arg0: T0, arg1: T1) -> *"type"
```

Calls an external function.

Parameters:

- **callee** (StringLiteral): The name of the external function.
- **type** (AnyType): The return type.
- **T0** (AnyType): The first argument type.
- **T1** (AnyType): The second argument type.

Args:

- **arg0** (T0): The first argument.
- **arg1** (T1): The second argument.

Returns:

The external call result.

```
external_call[callee: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType, T2: AnyType](arg0: T0, arg1: T1, arg2: T2) -> *"type"
```

Calls an external function.

Parameters:

- **callee** (StringLiteral): The name of the external function.
- **type** (AnyType): The return type.
- **T0** (AnyType): The first argument type.
- **T1** (AnyType): The second argument type.
- **T2** (AnyType): The third argument type.

Args:

- **arg0** (T0): The first argument.
- **arg1** (T1): The second argument.
- **arg2** (T2): The third argument.

Returns:

The external call result.

```
external_call[callee: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType, T2: AnyType, T3: AnyType](arg0: T0, arg1: T1, arg2: T2, arg3: T3) -> *"type"
```

Calls an external function.

Parameters:

- **callee** (StringLiteral): The name of the external function.
- **type** (AnyType): The return type.
- **T0** (AnyType): The first argument type.
- **T1** (AnyType): The second argument type.
- **T2** (AnyType): The third argument type.
- **T3** (AnyType): The fourth argument type.

Args:

- **arg0** (T0): The first argument.
- **arg1** (T1): The second argument.
- **arg2** (T2): The third argument.
- **arg3** (T3): The fourth argument.

Returns:

The external call result.

```
external_call[callee: StringLiteral, type: AnyType, T0: AnyType, T1: AnyType, T2: AnyType, T3: AnyType, T4: AnyType](arg0: T0, arg1: T1, arg2: T2, arg3: T3, arg4: T4) -> *"type"
```

Calls an external function.

Parameters:

- **callee** (StringLiteral): The name of the external function.
- **type** (AnyType): The return type.
- **T0** (AnyType): The first argument type.
- **T1** (AnyType): The second argument type.
- **T2** (AnyType): The third argument type.
- **T3** (AnyType): The fourth argument type.
- **T4** (AnyType): The fifth argument type.

Args:

- **arg0** (T0): The first argument.
- **arg1** (T1): The second argument.
- **arg2** (T2): The third argument.
- **arg3** (T3): The fourth argument.
- **arg4** (T4): The fifth argument.

Returns:

The external call result.

gather

```
gather[type: DType, size: Int](base: SIMD[address, size], mask: SIMD[bool, size], passthrough: SIMD[type, size], alignment: Int) -> SIMD[type, size]
```

Reads scalar values from a SIMD vector, and gathers them into one vector.

The gather function reads scalar values from a SIMD vector of memory locations and gathers them into one vector. The memory locations are provided in the vector of pointers `base` as addresses. The memory is accessed according to the provided mask. The mask holds a bit for each vector lane, and is used to prevent memory accesses to the masked-off lanes. The masked-off lanes in the result vector are taken from the corresponding lanes of the passthrough operand.

In general, for some vector of pointers `base`, mask `mask`, and passthrough `pass` a call of the form:

```
gather(base, mask, pass)
```

is equivalent to the following sequence of scalar loads in C++:

```
for (int i = 0; i < N; i++)
    result[i] = mask[i] ? *base[i] : passthrough[i];
```

Parameters:

- **type** (DType): DType of the return SIMD buffer.
- **size** (Int): Size of the return SIMD buffer.

Args:

- **base** (SIMD[address, size]): The vector containing memory addresses that gather will access.
- **mask** (SIMD[bool, size]): A binary vector which prevents memory access to certain lanes of the base vector.
- **passthrough** (SIMD[type, size]): In the result vector, the masked-off lanes are replaced with the passthrough vector.
- **alignment** (Int): The alignment of the source addresses. Must be 0 or a power of two constant integer value.

Returns:

A SIMD[type, size] containing the result of the gather operation.

scatter

```
scatter[type: DType, size: Int](value: SIMD[type, size], base: SIMD[address, size], mask: SIMD[bool, size], alignment: Int)
```

Takes scalar values from a SIMD vector and scatters them into a vector of pointers.

The scatter operation stores scalar values from a SIMD vector of memory locations and scatters them into a vector of pointers. The memory locations are provided in the vector of pointers `base` as addresses. The memory is stored according to the provided mask. The mask holds a bit for each vector lane, and is used to prevent memory accesses to the masked-off lanes.

The `value` operand is a vector value to be written to memory. The `base` operand is a vector of pointers, pointing to where the value elements should be stored. It has the same underlying type as the `value` operand. The `mask` operand, `mask`, is a vector of boolean values. The types of the `mask` and the `value` operand must have the same number of vector elements.

The behavior of the `_scatter` is undefined if the op stores into the same memory location more than once.

In general, for some vector `%value`, vector of pointers `%base`, and mask `%mask` instructions of the form:

```
%0 = pop.simd.scatter %value, %base[%mask] : !pop.simd<N, type>
```

is equivalent to the following sequence of scalar loads in C++:

```
for (int i = 0; i < N; i++)
    if (mask[i])
        base[i] = value[i];
```

Parameters:

- **type** (DType): DType of value, the result SIMD buffer.
- **size** (Int): Size of value, the result SIMD buffer.

Args:

- **value** (SIMD[type, size]): The vector that will contain the result of the scatter operation.
- **base** (SIMD[address, size]): The vector containing memory addresses that scatter will access.
- **mask** (SIMD[bool, size]): A binary vector which prevents memory access to certain lanes of the base vector.
- **alignment** (Int): The alignment of the source addresses. Must be 0 or a power of two constant integer value.

prefetch

```
prefetch[type: DType, params: PrefetchOptions](addr: DTypePointer[type])
```

Prefetches an instruction or data into cache before it is used.

The prefetch function provides prefetching hints for the target to prefetch instruction or data into cache before they are used.

Parameters:

- **type** (DType): The DType of value stored in addr.
- **params** (PrefetchOptions): Configuration options for the prefetch intrinsic.

Args:

- **addr** (DTypePointer[type]): The data pointer to prefetch.

masked_load

```
masked_load[type: DType, size: Int](addr: DTypePointer[type], mask: SIMD[bool, size], passthrough: SIMD[type, size], alignment: Int) -> SIMD[type, size]
```

Loads data from memory and return it, replacing masked lanes with values from the passthrough vector.

Parameters:

- **type** (DType): DType of the return SIMD buffer.
- **size** (Int): Size of the return SIMD buffer.

Args:

- **addr** (DTypePointer[type]): The base pointer for the load.
- **mask** (SIMD[bool, size]): A binary vector which prevents memory access to certain lanes of the memory stored at addr.
- **passthrough** (SIMD[type, size]): In the result vector, the masked-off lanes are replaced with the passthrough vector.
- **alignment** (Int): The alignment of the source addresses. Must be 0 or a power of two constant integer value. Default is 1.

Returns:

The loaded memory stored in a vector of type SIMD[type, size].

masked_store

```
masked_store[type: DType, size: Int](value: SIMD[type, size], addr: DTypePointer[type], mask: SIMD[bool, size], alignment: Int)
```

Stores a value at a memory location, skipping masked lanes.

Parameters:

- **type** (DType): DType of value, the data to store.
- **size** (Int): Size of value, the data to store.

Args:

- **value** (SIMD[type, size]): The vector containing data to store.
- **addr** (DTypePointer[type]): A vector of memory location to store data at.
- **mask** (SIMD[bool, size]): A binary vector which prevents memory access to certain lanes of value.
- **alignment** (Int): The alignment of the destination locations. Must be 0 or a power of two constant integer value.

compressed_store

```
compressed_store[type: DType, size: Int](value: SIMD[type, size], addr: DTypePointer[type], mask: SIMD[bool, size])
```

Compresses the lanes of value, skipping mask lanes, and stores at addr.

Parameters:

- **type** (DType): DType of value, the value to store.
- **size** (Int): Size of value, the value to store.

Args:

- **value** (SIMD[type, size]): The vector containing data to store.
- **addr** (DTypePointer[type]): The memory location to store the compressed data.
- **mask** (SIMD[bool, size]): A binary vector which prevents memory access to certain lanes of value.

strided_load

```
strided_load[type: DType, simd_width: Int](addr: DTypePointer[type], stride: Int, mask: SIMD[bool, simd_width]) -> SIMD[type, simd_width]
```

Loads values from addr according to a specific stride.

Parameters:

- **type** (DType): DType of value, the value to store.
- **simd_width** (Int): The width of the SIMD vectors.

Args:

- **addr** (DTypePointer[type]): The memory location to load data from.
- **stride** (Int): How many lanes to skip before loading again.
- **mask** (SIMD[bool, simd_width]): A binary vector which prevents memory access to certain lanes of value.

Returns:

A vector containing the loaded data.

```
strided_load[type: DType, simd_width: Int](addr: DTypePointer[type], stride: Int) -> SIMD[type, simd_width]
```

Loads values from addr according to a specific stride.

Parameters:

- **type** (DType): DType of value, the value to store.
- **simd_width** (Int): The width of the SIMD vectors.

Args:

- **addr** (DTypePointer[type]): The memory location to load data from.
- **stride** (Int): How many lanes to skip before loading again.

Returns:

A vector containing the loaded data.

strided_store

```
strided_store[type: DType, simd_width: Int](value: SIMD[type, simd_width], addr: DTypePointer[type], stride: Int, mask: SIMD[bool, simd_width])
```

Loads values from addr according to a specific stride.

Parameters:

- **type** (DType): DType of value, the value to store.
- **simd_width** (Int): The width of the SIMD vectors.

Args:

- **value** (SIMD[type, simd_width]): The values to store.
- **addr** (DTypePointer[type]): The location to store values at.
- **stride** (Int): How many lanes to skip before storing again.
- **mask** (SIMD[bool, simd_width]): A binary vector which prevents memory access to certain lanes of value.

```
strided_store[type: DType, simd_width: Int](value: SIMD[type, simd_width], addr: DTypePointer[type], stride: Int)
```

Loads values from addr according to a specific stride.

Parameters:

- **type** (DType): DType of value, the value to store.
- **simd_width** (Int): The width of the SIMD vectors.

Args:

- **value** (SIMD[type, simd_width]): The values to store.
- **addr** (DTypePointer[type]): The location to store values at.
- **stride** (Int): How many lanes to skip before storing again.

param_env

Module

Implements functions for retrieving compile-time defines.

You can use these functions to set parameter values or runtime constants based on name-value pairs defined on the command line. For example:

```
from sys.param_env import is_defined
from tensor import Tensor, TensorSpec

alias float_type: DType = DType.float32 if is_defined["FLOAT32"]() else DType.float64

let spec = TensorSpec(float_type, 256, 256)
var image = Tensor[float_type](spec)
```

And on the command line:

```
mojo -D FLOAT_32 main.mojo
```

For more information, see the [Mojo build docs](#). The `mojo run` command also supports the `-D` option.

You can import these APIs from the `sys` package. For example:

```
from sys.param_env import is_defined
```

is_defined

```
is_defined[name: StringLiteral]() -> Bool
```

Return true if the named value is defined.

Parameters:

- **name** (StringLiteral): The name to test.

Returns:

True if the name is defined.

env_get_int

```
env_get_int[name: StringLiteral]() -> Int
```

Try to get an integer-valued define. Compilation fails if the name is not defined.

Parameters:

- **name** (StringLiteral): The name of the define.

Returns:

An integer parameter value.

```
env_get_int[name: StringLiteral, default: Int]() -> Int
```

Try to get an integer-valued define. If the name is not defined, return a default value instead.

Parameters:

- **name** (StringLiteral): The name of the define.
- **default** (Int): The default value to use.

Returns:

An integer parameter value.

env_get_string

```
env_get_string[name: StringLiteral]() -> StringLiteral
```

Try to get a string-valued define. Compilation fails if the name is not defined.

Parameters:

- **name** (StringLiteral): The name of the define.

Returns:

A string parameter value.

```
env_get_string[name: StringLiteral, default: StringLiteral]() -> StringLiteral
```

Try to get a string-valued define. If the name is not defined, return a default value instead.

Parameters:

- **name** (StringLiteral): The name of the define.
- **default** (StringLiteral): The default value to use.

Returns:

A string parameter value.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

tensor

Module

Implements the Tensor type.

Example:

```
from tensor import Tensor, TensorSpec, TensorShape
from utils.index import Index
from random import rand

let height = 256
let width = 256
let channels = 3

# Create the tensor of dimensions height, width, channels
# and fill with random values.
let image = rand[DType.float32](height, width, channels)

# Declare the grayscale image.
let spec = TensorSpec(DType.float32, height, width)
var gray_scale_image = Tensor[DType.float32](spec)

# Perform the RGB to grayscale transform.
for y in range(height):
    for x in range(width):
        let r = image[y,x,0]
        let g = image[y,x,1]
        let b = image[y,x,2]
        gray_scale_image[Index(y,x)] = 0.299 * r + 0.587 * g + 0.114 * b

print(gray_scale_image.shape().__str__())
```

Tensor

A tensor type which owns its underlying data and is parameterized on DType.

Parameters:

- **dtype** (DType): The underlying element type of the tensor.

Functions:

__init__

```
__init__(inout self: Self)
```

Default initializer for TensorShape.

```
__init__(inout self: Self, *dims: Int)
```

Allocates a tensor using the shape provided.

Args:

- **dims** (*Int): The tensor dimensions.

```
__init__(inout self: Self, owned shape: TensorShape)
```

Allocates a tensor using the shape provided.

Args:

- **shape** (TensorShape): The tensor shape.

`__init__(inout self: Self, owned spec: TensorSpec)`

Allocates a tensor using the spec provided.

Args:

- **spec** (TensorSpec): The tensor spec.

`__init__(inout self: Self, owned ptr: DTypePointer[dtype], owned shape: TensorShape)`

Initializes a Tensor from the pointer and shape provided. The caller relinquishes the ownership of the pointer being passed in.

Args:

- **ptr** (DTypePointer[dtype]): The data pointer.
- **shape** (TensorShape): The tensor shapes.

`__init__(inout self: Self, owned ptr: DTypePointer[dtype], owned spec: TensorSpec)`

Initializes a Tensor from the pointer and shape provided. The caller relinquishes the ownership of the pointer being passed in.

Args:

- **ptr** (DTypePointer[dtype]): The data pointer.
- **spec** (TensorSpec): The tensor spec.

`__copyinit__`

`__copyinit__(inout self: Self, other: Self)`

Creates a deep copy of an existing tensor.

Args:

- **other** (Self): The tensor to copy from.

`__moveinit__`

`__moveinit__(inout self: Self, owned existing: Self)`

Move initializer for the tensor.

Args:

- **existing** (Self): The tensor to move.

`__del__`

`__del__(owned self: Self)`

Delete the spec and release any owned memory.

`__getitem__`

`__getitem__(self: Self, index: Int) -> SIMD[dtype, 1]`

Gets the value at the specified index.

Args:

- **index** (Int): The index of the value to retrieve.

Returns:

The value at the specified indices.

```
__getitem__(self: Self, *indices: Int) -> SIMD[dtype, 1]
```

Gets the value at the specified indices.

Args:

- **indices** (*Int): The indices of the value to retrieve.

Returns:

The value at the specified indices.

```
__getitem__(self: Self, indices: VariadicList[Int]) -> SIMD[dtype, 1]
```

Gets the value at the specified indices.

Args:

- **indices** (VariadicList[Int]): The indices of the value to retrieve.

Returns:

The value at the specified indices.

```
__getitem__[len: Int](self: Self, indices: StaticIntTuple[len]) -> SIMD[dtype, 1]
```

Gets the SIMD value at the specified indices.

Parameters:

- **len** (Int): The length of the indecies.

Args:

- **indices** (StaticIntTuple[len]): The indices of the value to retrieve.

Returns:

The value at the specified indices.

__setitem__

```
__setitem__(inout self: Self, index: Int, val: SIMD[dtype, 1])
```

Sets the value at the specified index.

Args:

- **index** (Int): The index of the value to set.
- **val** (SIMD[dtype, 1]): The value to store.

```
__setitem__(inout self: Self, indices: VariadicList[Int], val: SIMD[dtype, 1])
```

Sets the value at the specified indices.

Args:

- **indices** (VariadicList[Int]): The indices of the value to set.
- **val** (SIMD[dtype, 1]): The value to store.

```
__setitem__[len: Int](inout self: Self, indices: StaticIntTuple[len], val: SIMD[dtype, 1])
```

Sets the value at the specified indices.

Parameters:

- **len** (Int): The length of the indecies.

Args:

- **indices** (StaticIntTuple[len]): The indices of the value to set.
- **val** (SIMD[dtype, 1]): The value to store.

__eq__

```
__eq__(self: Self, other: Self) -> Bool
```

Returns True if the two tensors are the same and False otherwise.

Args:

- **other** (Self): The other Tensor to compare against.

Returns:

True if the two tensors are the same and False otherwise.

__ne__

```
__ne__(self: Self, other: Self) -> Bool
```

Returns True if the two tensors are not the same and False otherwise.

Args:

- **other** (Self): The other Tensor to compare against.

Returns:

True if the two tensors are the not the same and False otherwise.

data

```
data(self: Self) -> DTypePointer[dtype]
```

Gets the underlying Data pointer to the Tensor.

Returns:

The underlying data pointer of the tensor.

type

```
type(self: Self) -> DType
```

Gets the underlying DType of the tensor.

Returns:

The underlying DType of the tensor.

rank

```
rank(self: Self) -> Int
```

Gets the rank of the tensor.

Returns:

The rank of the tensor.

num_elements

```
num_elements(self: Self) -> Int
```

Gets the total number of elements in the tensor.

Returns:

The total number of elements in the tensor.

bytecount

```
bytecount(self: Self) -> Int
```

Gets the total bytecount of the tensor.

Returns:

The total bytecount of the tensor.

spec

```
spec(self: Self) -> TensorSpec
```

Gets the specification of the tensor.

Returns:

The underlying tensor spec of the tensor.

shape

```
shape(self: Self) -> TensorShape
```

Gets the shape of the tensor.

Returns:

The underlying tensor shape of the tensor.

dim

```
dim(self: Self, idx: Int) -> Int
```

Gets the dimension at the specified index.

Args:

- **idx** (Int): The dimension index.

Returns:

The dimension at the specified index.

simd_load

```
simd_load[simd_width: Int](self: Self, index: Int) -> SIMD[dtype, simd_width]
```

Gets the SIMD value at the specified index.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.

Args:

- **index** (Int): The index of the value to retrieve.

Returns:

The SIMD value at the specified indices.

```
simd_load[simd_width: Int](self: Self, *indices: Int) -> SIMD[dtype, simd_width]
```

Gets the SIMD value at the specified indices.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.

Args:

- **indices** (*Int): The indices of the value to retrieve.

Returns:

The SIMD value at the specified indices.

```
simd_load[simd_width: Int](self: Self, indices: VariadicList[Int]) -> SIMD[dtype, simd_width]
```

Gets the SIMD value at the specified indices.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.

Args:

- **indices** (VariadicList[Int]): The indices of the value to retrieve.

Returns:

The SIMD value at the specified indices.

```
simd_load[simd_width: Int, len: Int](self: Self, indices: StaticIntTuple[len]) -> SIMD[dtype, simd_width]
```

Gets the SIMD value at the specified indices.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.
- **len** (Int): The length of the indecies.

Args:

- **indices** (StaticIntTuple[len]): The indices of the value to retrieve.

Returns:

The SIMD value at the specified indices.

simd_store

```
simd_store[simd_width: Int](inout self: Self, index: Int, val: SIMD[dtype, simd_width])
```

Sets the SIMD value at the specified index.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.

Args:

- **index** (Int): The index of the value to set.
- **val** (SIMD[dtype, simd_width]): The SIMD value to store.

```
simd_store[simd_width: Int](inout self: Self, indices: VariadicList[Int], val: SIMD[dtype, simd_width])
```

Sets the SIMD value at the specified indices.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.

Args:

- **indices** (VariadicList[Int]): The indices of the value to set.
- **val** (SIMD[dtype, simd_width]): The SIMD value to store.

```
simd_store[simd_width: Int, len: Int](inout self: Self, indices: StaticIntTuple[len], val: SIMD[dtype, simd_width])
```

Sets the SIMD value at the specified indices.

Parameters:

- **simd_width** (Int): The SIMD width of the vector.
- **len** (Int): The length of the indecies.

Args:

- **indices** (StaticIntTuple[len]): The indices of the value to set.
- **val** (SIMD[dtype, simd_width]): The SIMD value to store.

tensor_shape

Module

Implements the `TensorShape` type.

You can import these APIs from the `tensor` package. For example:

```
from tensor import TensorShape
```

TensorShape

A space efficient representation of a tensor shape. This struct implements value semantics and owns its underlying data.

Functions:

`__init__`

```
__init__(inout self: Self)
```

Default initializer for `TensorShape`.

```
__init__(inout self: Self, *shapes: Int)
```

Initializes a `TensorShape` from the values provided.

Args:

- **shapes** (*Int): The shapes to initialize the shape with.

```
__init__(inout self: Self, shapes: VariadicList[Int])
```

Initializes a `TensorShape` from the values provided.

Args:

- **shapes** (VariadicList[Int]): The shapes to initialize the shape with.

`__copyinit__`

```
__copyinit__(inout self: Self, other: Self)
```

Creates a deep copy of an existing shape.

Args:

- **other** (Self): The shape to copy.

`__moveinit__`

```
__moveinit__(inout self: Self, owned existing: Self)
```

Move initializer for the shape.

Args:

- **existing** (Self): The shape to move.

`__del__`

`__del__(owned self: Self)`

Delete the shape and release any owned memory.

`__getitem__`

`__getitem__(self: Self, index: Int) -> Int`

Gets the dimension at the specified index.

Args:

- **index** (Int): The dimension index.

Returns:

The dimension at the specified index.

`__eq__`

`__eq__(self: Self, other: Self) -> Bool`

Returns True if the two values are the same and False otherwise.

Args:

- **other** (Self): The other TensorShape to compare against.

Returns:

True if the two shapes are the same and False otherwise.

`__ne__`

`__ne__(self: Self, other: Self) -> Bool`

Returns True if the two values are not the same and False otherwise.

Args:

- **other** (Self): The other TensorShape to compare against.

Returns:

True if the two shapes are the not the same and False otherwise.

`rank`

`rank(self: Self) -> Int`

Gets the rank of the shape.

Returns:

The rank of the shape.

`num_elements`

`num_elements(self: Self) -> Int`

Gets the total number of elements in the shape.

Returns:

The total number of elements in the shape.

__repr__

`__repr__(self: Self) -> String`

Returns the string representation of the shape.

Returns:

The string representation of the shape.

__str__

`__str__(self: Self) -> String`

Returns the string representation of the shape.

Returns:

The string representation of the shape.

tensor_spec

Module

Implements the `TensorSpec` type.

You can import these APIs from the `tensor` package. For example:

```
from tensor import TensorSpec
```

TensorSpec

A space efficient representation of a tensor shape and dtype. This struct implements value semantics and owns its underlying data.

Fields:

- **shape** (`TensorShape`): The underlying shape of the specification.

Functions:

`__init__`

```
__init__(inout self: Self)
```

Default initializer for `TensorShape`.

```
__init__(inout self: Self, type: DType, *shapes: Int)
```

Initializes a `Tensorspec` from the dtype and shapes provided.

Args:

- **type** (`DType`): The dtype of the specification.
- **shapes** (*`Int`): The shapes to initialize the shape with.

```
__init__(inout self: Self, type: DType, shapes: VariadicList[Int])
```

Initializes a `Tensorspec` from the dtype and shapes provided.

Args:

- **type** (`DType`): The dtype of the specification.
- **shapes** (`VariadicList[Int]`): The shapes to initialize the shape with.

```
__init__(inout self: Self, type: DType, owned shape: TensorShape)
```

Initializes a `Tensorspec` from the dtype and shape provided.

Args:

- **type** (`DType`): The dtype of the specification.
- **shape** (`TensorShape`): The shapes to initialize the shape with.

`__copyinit__`

```
__copyinit__(inout self: Self, other: Self)
```

Creates a deep copy of an existing spec.

Args:

- **other** (Self): The spec to copy.

__moveinit__

`__moveinit__(inout self: Self, owned existing: Self)`

Move initializer for the spec.

Args:

- **existing** (Self): The spec to move.

__del__

`__del__(owned self: Self)`

__getitem__

`__getitem__(self: Self, index: Int) -> Int`

Gets the dimension at the specified index.

Args:

- **index** (Int): The dimension index.

Returns:

The dimension at the specified index.

__eq__

`__eq__(self: Self, other: Self) -> Bool`

Returns True if the two values are the same and False otherwise.

Args:

- **other** (Self): The other TensorSpec to compare against.

Returns:

True if the two specs are the same and False otherwise.

__ne__

`__ne__(self: Self, other: Self) -> Bool`

Returns True if the two values are not the same and False otherwise.

Args:

- **other** (Self): The other TensorSpec to compare against.

Returns:

True if the two specs are not the same and False otherwise.

rank

`rank(self: Self) -> Int`

Gets the rank of the spec.

Returns:

The rank of the spec.

dtype

```
dtype(self: Self) -> DType
```

Gets the rank of the DType of the spec.

Returns:

The DType of the spec.

num_elements

```
num_elements(self: Self) -> Int
```

Gets the total number of elements in the spec.

Returns:

The total number of elements in the spec.

bytecount

```
bytecount(self: Self) -> Int
```

Gets the total byte count.

Returns:

The total byte count.

__repr__

```
__repr__(self: Self) -> String
```

Returns the string representation of the spec.

Returns:

The string representation of the spec.

__str__

```
__str__(self: Self) -> String
```

Returns the string representation of the spec.

Returns:

The string representation of the spec.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

testing

Module

Implements various testing utils.

You can import these APIs from the `testing` package. For example:

```
from testing import assert_true
```

assert_true

```
assert_true(val: Bool, msg: String) -> Bool
```

Asserts that the input value is True. If it is not then a message is printed which contains the input message.

Args:

- **val** (Bool): The value to assert to be True.
- **msg** (String): The message to be printed if the assertion fails.

Returns:

True if the assert succeeds and False otherwise.

assert_false

```
assert_false(val: Bool, msg: String) -> Bool
```

Asserts that the input value is False. If it is not then a message is printed which contains the input message.

Args:

- **val** (Bool): The value to assert to be False.
- **msg** (String): The message to be printed if the assertion fails.

Returns:

True if the assert succeeds and False otherwise.

```
assert_false(val: Bool) -> Bool
```

Asserts that the input value is False. If it is not then a message is printed.

Args:

- **val** (Bool): The value to assert to be False.

Returns:

True if the assert succeeds and False otherwise.

assert_equal

```
assert_equal(lhs: Int, rhs: Int) -> Bool
```

Asserts that the input values are equal. If it is not then a message is printed.

Args:

- **lhs** (Int): The lhs of the equality.
- **rhs** (Int): The rhs of the equality.

Returns:

True if the assert succeeds and False otherwise.

```
assert_equal(lhs: String, rhs: String) -> Bool
```

Asserts that the input values are equal. If it is not then a message is printed.

Args:

- **lhs** (String): The lhs of the equality.
- **rhs** (String): The rhs of the equality.

Returns:

True if the assert succeeds and False otherwise.

```
assert_equal[type: DType, size: Int](lhs: SIMD[type, size], rhs: SIMD[type, size]) -> Bool
```

Asserts that the input values are equal. If it is not then a message is printed.

Parameters:

- **type** (DType): The dtype of the left- and right-hand-side SIMD vectors.
- **size** (Int): The width of the left- and right-hand-side SIMD vectors.

Args:

- **lhs** (SIMD[type, size]): The lhs of the equality.
- **rhs** (SIMD[type, size]): The rhs of the equality.

Returns:

True if the assert succeeds and False otherwise.

assert_not_equal

```
assert_not_equal(lhs: Int, rhs: Int) -> Bool
```

Asserts that the input values are not equal. If it is not then a message is printed.

Args:

- **lhs** (Int): The lhs of the inequality.
- **rhs** (Int): The rhs of the inequality.

Returns:

True if the assert succeeds and False otherwise.

```
assert_not_equal(lhs: String, rhs: String) -> Bool
```

Asserts that the input values are not equal. If it is not then a message is printed.

Args:

- **lhs** (String): The lhs of the inequality.
- **rhs** (String): The rhs of the inequality.

Returns:

True if the assert succeeds and False otherwise.

```
assert_not_equal[type: DType, size: Int](lhs: SIMD[type, size], rhs: SIMD[type, size]) -> Bool
```

Asserts that the input values are not equal. If it is not then a message is printed.

Parameters:

- **type** (DType): The dtype of the left- and right-hand-side SIMD vectors.
- **size** (Int): The width of the left- and right-hand-side SIMD vectors.

Args:

- **lhs** (SIMD[type, size]): The lhs of the inequality.
- **rhs** (SIMD[type, size]): The rhs of the inequality.

Returns:

True if the assert succeeds and False otherwise.

assert_almost_equal

```
assert_almost_equal[type: DType, size: Int](lhs: SIMD[type, size], rhs: SIMD[type, size]) -> Bool
```

Asserts that the input values are equal up to a tolerance. If it is not then a message is printed.

Parameters:

- **type** (DType): The dtype of the left- and right-hand-side SIMD vectors.
- **size** (Int): The width of the left- and right-hand-side SIMD vectors.

Args:

- **lhs** (SIMD[type, size]): The lhs of the equality.
- **rhs** (SIMD[type, size]): The rhs of the equality.

Returns:

True if the assert succeeds and False otherwise.

```
assert_almost_equal[type: DType, size: Int](lhs: SIMD[type, size], rhs: SIMD[type, size], absolute_tolerance: SIMD[type, 1], relative_tolerance: SIMD[type, 1]) -> Bool
```

Asserts that the input values are equal up to a tolerance. If it is not then a message is printed.

Parameters:

- **type** (DType): The dtype of the left- and right-hand-side SIMD vectors.
- **size** (Int): The width of the left- and right-hand-side SIMD vectors.

Args:

- **lhs** (SIMD[type, size]): The lhs of the equality.
- **rhs** (SIMD[type, size]): The rhs of the equality.
- **absolute_tolerance** (SIMD[type, 1]): The absolute tolerance.
- **relative_tolerance** (SIMD[type, 1]): The relative tolerance.

Returns:

True if the assert succeeds and False otherwise.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

time

Module

Implements basic utils for working with time.

You can import these APIs from the `time` package. For example:

```
from time import now
```

now

```
now() -> Int
```

Returns the current monotonic time in nanoseconds. This function queries the current platform's monotonic clock, making it useful for measuring time differences, but the significance of the returned value varies depending on the underlying implementation.

Returns:

The current time in ns.

time_function

```
time_function[func: fn() capturing -> None]() -> Int
```

Measures the time spent in the function.

Parameters:

- **func** (`fn() capturing -> None`): The function to time.

Returns:

The time elapsed in the function in ns.

sleep

```
sleep(sec: SIMD[f64, 1])
```

Suspends the current thread for the seconds specified.

Args:

- **sec** (`SIMD[f64, 1]`): The number of seconds to sleep for.

```
sleep(sec: Int)
```

Suspends the current thread for the seconds specified.

Args:

- **sec** (`Int`): The number of seconds to sleep for.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[**Get started with Mojo**](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[**Why Mojo**](#)

[A backstory and rationale for why we created the Mojo language.](#)

[**Mojo programming manual**](#)

[A tour of major Mojo language features with code examples.](#)

[**Mojo modules**](#)

[A list of all modules in the current standard library.](#)

[**Mojo notebooks**](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[**Mojo roadmap & sharp edges**](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[**Mojo FAQ**](#)

[Answers to questions we expect about Mojo.](#)

[**Mojo changelog**](#)

[A history of significant Mojo changes.](#)

[**Mojo community**](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

index

Module

Implements StaticIntTuple which is commonly used to represent N-D indices.

You can import these APIs from the `utils` package. For example:

```
from utils.index import StaticIntTuple
```

Aliases:

- `mlir_bool = scalar<bool>`

StaticIntTuple

A base struct that implements size agnostic index functions.

Parameters:

- **size** (`Int`): The size of the tuple.

Fields:

- **data** (`StaticTuple[size, Int]`): The underlying storage of the tuple value.

Functions:

`__init__`

```
__init__() -> Self
```

Constructs a static int tuple of the given size.

Returns:

The constructed tuple.

```
__init__(value: index) -> Self
```

Constructs a sized 1 static int tuple of given the element value.

Args:

- **value** (`index`): The initial value.

Returns:

The constructed tuple.

```
__init__(*elems: Int) -> Self
```

Constructs a static int tuple given a set of arguments.

Args:

- **elems** (*`Int`): The elements to construct the tuple.

Returns:

The constructed tuple.

`__init__(elem: Int) -> Self`

Constructs a static int tuple given a set of arguments.

Args:

- **elem** (Int): The elem to splat into the tuple.

Returns:

The constructed tuple.

`__init__(values: VariadicList[Int]) -> Self`

Creates a tuple constant using the specified values.

Args:

- **values** (VariadicList[Int]): The list of values.

Returns:

A tuple with the values filled in.

`__init__(values: DimList) -> Self`

Creates a tuple constant using the specified values.

Args:

- **values** (DimList): The list of values.

Returns:

A tuple with the values filled in.

`__init__(data: StaticTuple[size, Int]) -> Self`

`__getitem__`

`__getitem__(self: Self, index: Int) -> Int`

Gets an element from the tuple by index.

Args:

- **index** (Int): The element index.

Returns:

The tuple element value.

`__setitem__`

`__setitem__[index: Int](inout self: Self, val: Int)`

Sets an element in the tuple at the given static index.

Parameters:

- **index** (Int): The element index.

Args:

- **val** (Int): The value to store.

__setitem__(inout self: Self, index: Int, val: Int)

Sets an element in the tuple at the given index.

Args:

- **index** (Int): The element index.
- **val** (Int): The value to store.

__lt__

__lt__(self: Self, rhs: Self) -> Bool

Compares this tuple to another tuple using LT comparison.

A tuple is less-than another tuple if all corresponding elements of lhs is less than rhs.

Note: This is **not** a lexical comparison.

Args:

- **rhs** (Self): Right hand side tuple.

Returns:

The comparison result.

__le__

__le__(self: Self, rhs: Self) -> Bool

Compares this tuple to another tuple using LE comparison.

A tuple is less-or-equal than another tuple if all corresponding elements of lhs is less-or-equal than rhs.

Note: This is **not** a lexical comparison.

Args:

- **rhs** (Self): Right hand side tuple.

Returns:

The comparison result.

__eq__

__eq__(self: Self, rhs: Self) -> Bool

Compares this tuple to another tuple for equality.

The tuples are equal if all corresponding elements are equal.

Args:

- **rhs** (Self): The other tuple.

Returns:

The comparison result.

__ne__

`__ne__(self: Self, rhs: Self) -> Bool`

Compares this tuple to another tuple for non-equality.

The tuples are non-equal if at least one element of LHS isn't equal to the corresponding element from RHS.

Args:

- **rhs** (Self): The other tuple.

Returns:

The comparison result.

__gt__

`__gt__(self: Self, rhs: Self) -> Bool`

Compares this tuple to another tuple using GT comparison.

A tuple is greater-than than another tuple if all corresponding elements of lhs is greater-than than rhs.

Note: This is **not** a lexical comparison.

Args:

- **rhs** (Self): Right hand side tuple.

Returns:

The comparison result.

__ge__

`__ge__(self: Self, rhs: Self) -> Bool`

Compares this tuple to another tuple using GE comparison.

A tuple is greater-or-equal than another tuple if all corresponding elements of lhs is greater-or-equal than rhs.

Note: This is **not** a lexical comparison.

Args:

- **rhs** (Self): Right hand side tuple.

Returns:

The comparison result.

__add__

`__add__(self: Self, rhs: Self) -> Self`

Performs element-wise integer add.

Args:

- **rhs** (Self): Right hand side operand.

Returns:

The resulting index tuple.

__sub__

`__sub__(self: Self, rhs: Self) -> Self`

Performs element-wise integer subtract.

Args:

- **rhs** (Self): Right hand side operand.

Returns:

The resulting index tuple.

__mul__

`__mul__(self: Self, rhs: Self) -> Self`

Performs element-wise integer multiply.

Args:

- **rhs** (Self): Right hand side operand.

Returns:

The resulting index tuple.

__floordiv__

`__floordiv__(self: Self, rhs: Self) -> Self`

Performs element-wise integer floor division.

Args:

- **rhs** (Self): Right hand side operand.

Returns:

The resulting index tuple.

__len__

`__len__(self: Self) -> Int`

Returns the size of the tuple.

Returns:

The tuple size.

as_tuple

`as_tuple(self: Self) -> StaticTuple[size, Int]`

Converts this `StaticIntTuple` to `StaticTuple`.

Returns:

The corresponding StaticTuple object.

flattened_length

```
flattened_length(self: Self) -> Int
```

Returns the flattened length of the tuple.

Returns:

The flattened length of the tuple.

remu

```
remu(self: Self, rhs: Self) -> Self
```

Performs element-wise integer unsigned modulo.

Args:

- **rhs** (Self): Right hand side operand.

Returns:

The resulting index tuple.

Index

```
Index(x: Int) -> StaticIntTuple[1]
```

Constructs a 1-D Index from the given value.

Args:

- **x** (Int): The initial value.

Returns:

The constructed StaticIntTuple.

```
Index(x: Int, y: Int) -> StaticIntTuple[2]
```

Constructs a 2-D Index from the given values.

Args:

- **x** (Int): The 1st initial value.
- **y** (Int): The 2nd initial value.

Returns:

The constructed StaticIntTuple.

```
Index(x: Int, y: Int, z: Int) -> StaticIntTuple[3]
```

Constructs a 3-D Index from the given values.

Args:

- **x** (Int): The 1st initial value.
- **y** (Int): The 2nd initial value.

- **z** (Int): The 3rd initial value.

Returns:

The constructed StaticIntTuple.

```
Index(x: Int, y: Int, z: Int, w: Int) -> StaticIntTuple[4]
```

Constructs a 4-D Index from the given values.

Args:

- **x** (Int): The 1st initial value.
- **y** (Int): The 2nd initial value.
- **z** (Int): The 3rd initial value.
- **w** (Int): The 4th initial value.

Returns:

The constructed StaticIntTuple.

```
Index(x: Int, y: Int, z: Int, w: Int, v: Int) -> StaticIntTuple[5]
```

Constructs a 5-D Index from the given values.

Args:

- **x** (Int): The 1st initial value.
- **y** (Int): The 2nd initial value.
- **z** (Int): The 3rd initial value.
- **w** (Int): The 4th initial value.
- **v** (Int): The 5th initial value.

Returns:

The constructed StaticIntTuple.

product

```
product[size: Int](tuple: StaticIntTuple[size], end_idx: Int) -> Int
```

Computes a product of values in the tuple up to the given index.

Parameters:

- **size** (Int): The tuple size.

Args:

- **tuple** (StaticIntTuple[size]): The tuple to get a product of.
- **end_idx** (Int): The end index.

Returns:

The product of all tuple elements in the given range.

```
product[size: Int](tuple: StaticIntTuple[size], start_idx: Int, end_idx: Int) -> Int
```

Computes a product of values in the tuple in the given index range.

Parameters:

- **size** (Int): The tuple size.

Args:

- **tuple** (`StaticIntTuple[size]`): The tuple to get a product of.
- **start_idx** (`Int`): The start index of the range.
- **end_idx** (`Int`): The end index of the range.

Returns:

The product of all tuple elements in the given range.

list

Module

Provides utilities for working with static and variadic lists.

You can import these APIs from the `utils` package. For example:

```
from utils.list import Dim
```

Dim

A static or dynamic dimension modeled with an optional integer.

This class is meant to represent an optional static dimension. When a value is present, the dimension has that static value. When a value is not present, the dimension is dynamic.

Aliases:

- `type` = `Variant[i1, Int]`

Fields:

- **value** (`Variant[i1, Int]`): Either a boolean indicating that the dimension is dynamic, or the static value of the dimension.

Functions:

`__init__`

```
__init__(value: Int) -> Self
```

Creates a statically-known dimension.

Args:

- **value** (`Int`): The static dimension value.

Returns:

A dimension with a static value.

```
__init__(value: index) -> Self
```

Creates a statically-known dimension.

Args:

- **value** (`index`): The static dimension value.

Returns:

A dimension with a static value.

```
__init__() -> Self
```

Creates a dynamic dimension.

Returns:

A dimension value with no static value.

`__init__(value: Variant[i1, Int]) -> Self`

`__bool__`

`__bool__(self: Self) -> Bool`

Returns True if the dimension has a static value.

Returns:

Whether the dimension has a static value.

`__eq__`

`__eq__(self: Self, rhs: Self) -> Bool`

Compares two dimensions for equality.

Args:

- **rhs** (`Self`): The other dimension.

Returns:

True if the dimensions are the same.

`__mul__`

`__mul__(self: Self, rhs: Self) -> Self`

Multiplies two dimensions.

If either are unknown, the result is unknown as well.

Args:

- **rhs** (`Self`): The other dimension.

Returns:

The product of the two dimensions.

`has_value`

`has_value(self: Self) -> Bool`

Returns True if the dimension has a static value.

Returns:

Whether the dimension has a static value.

`is_dynamic`

`is_dynamic(self: Self) -> Bool`

Returns True if the dimension has a dynamic value.

Returns:

Whether the dimension is dynamic.

`get`

```
get(self: Self) -> Int
```

Gets the static dimension value.

Returns:

The static dimension value.

is_multiple

```
is_multiple[alignment: Int](self: Self) -> Bool
```

Checks if the dimension is aligned.

Parameters:

- **alignment** (Int): The alignment requirement.

Returns:

Whether the dimension is aligned.

DimList

This type represents a list of dimensions. Each dimension may have a static value or not have a value, which represents a dynamic dimension.

Fields:

- **value** (VariadicList[Dim]): The underlying storage for the list of dimensions.

Functions:

__init__

```
__init__(values: VariadicList[Dim]) -> Self
```

Creates a dimension list from the given list of values.

Args:

- **values** (VariadicList[Dim]): The initial dim values list.

Returns:

A dimension list.

```
__init__(*values: Dim) -> Self
```

Creates a dimension list from the given Dim values.

Args:

- **values** (*Dim): The initial dim values.

Returns:

A dimension list.

__len__

```
__len__(self: Self) -> Int
```

Gets the size of the DimList.

Returns:

The number of elements in the DimList.

at

```
at[i: Int](self: Self) -> Dim
```

Gets the dimension at a specified index.

Parameters:

- **i** (Int): The dimension index.

Returns:

The dimension at the specified index.

product

```
product[length: Int](self: Self) -> Dim
```

Computes the product of all the dimensions in the list.

If any are dynamic, the result is a dynamic dimension value.

Parameters:

- **length** (Int): The number of elements in the list.

Returns:

The product of all the dimensions.

product_range

```
product_range[start: Int, end: Int](self: Self) -> Dim
```

Computes the product of a range of the dimensions in the list.

If any in the range are dynamic, the result is a dynamic dimension value.

Parameters:

- **start** (Int): The starting index.
- **end** (Int): The end index.

Returns:

The product of all the dimensions.

contains

```
contains[length: Int](self: Self, value: Dim) -> Bool
```

Determines whether the dimension list contains a specified dimension value.

Parameters:

- **length** (Int): The number of elements in the list.

Args:

- **value** (Dim): The value to find.

Returns:

True if the list contains a dimension of the specified value.

all_known

```
all_known[length: Int](self: Self) -> Bool
```

Determines whether all dimensions are statically known.

Parameters:

- **length** (Int): The number of elements in the list.

Returns:

True if all dimensions have a static value.

create_unknown

```
create_unknown[length: Int]() -> Self
```

Creates a dimension list of all dynamic dimension values.

Parameters:

- **length** (Int): The number of elements in the list.

Returns:

A list of all dynamic dimension values.

VariadicList

A utility class to access variadic function arguments. Provides a “list” view of the function argument so that the size of the argument list and each individual argument can be accessed.

Parameters:

- **type** (AnyType): The type of the elements in the list.

Aliases:

- StorageType = variadic<"type">

Fields:

- **value** (variadic<"type">): The underlying storage for the variadic list.

Functions:

__init__

```
__init__(*value: *"type") -> Self
```

Constructs a VariadicList from a variadic list of arguments.

Args:

- **value** (**"type"): The variadic argument list to construct the variadic list with.

Returns:

The VariadicList constructed.

```
__init__(*value: *"type") -> Self
```

Constructs a VariadicList from a variadic argument type.

Args:

- **value** (**"type"): The variadic argument to construct the list with.

Returns:

The VariadicList constructed.

```
__getitem__
```

```
__getitem__(self: Self, index: Int) -> *"type"
```

Gets a single element on the variadic list.

Args:

- **index** (Int): The index of the element to access on the list.

Returns:

The element on the list corresponding to the given index.

```
__len__
```

```
__len__(self: Self) -> Int
```

Gets the size of the list.

Returns:

The number of elements on the variadic list.

VariadicListMem

A utility class to access variadic function arguments of memory-only types that may have ownership. It exposes pointers to the elements in a way that can be enumerated. Each element may be accessed with `__get_address_as_lvalue`.

Parameters:

- **type** (AnyType): The type of the elements in the list.

Aliases:

- StorageType = variadic<pointer<*"type">>

Fields:

- **value** (variadic<pointer<*"type">>): The underlying storage, a variadic list of pointers to elements of the given type.

Functions:

__init__

`__init__(*value: pointer<*"type">) -> Self`

Constructs a VariadicList from a variadic argument type.

Args:

- **value** (`*pointer<*"type">`): The variadic argument to construct the list with.

Returns:

The VariadicList constructed.

__getitem__

`__getitem__(self: Self, index: Int) -> pointer<*"type">`

Gets a single element on the variadic list.

Args:

- **index** (`Int`): The index of the element to access on the list.

Returns:

A low-level pointer to the element on the list corresponding to the given index.

__len__

`__len__(self: Self) -> Int`

Gets the size of the list.

Returns:

The number of elements on the variadic list.

static_tuple

Module

Implements StaticTuple, a statically-sized uniform container.

You can import these APIs from the `utils` package. For example:

```
from utils.static_tuple import StaticTuple
```

StaticTuple

A statically sized tuple type which contains elements of homogeneous types.

Parameters:

- **size** (Int): The size of the tuple.
- **_element_type** (AnyType): The type of the elements in the tuple.

Aliases:

- `element_type = _90x31__element_type`
- `type = array<#lit.struct.extract<:!kgen.declref<@"$builtin":@"$int"::@Int> size, "value">, _element_type>`

Fields:

- **array** (`array<#lit.struct.extract<:!kgen.declref<@"$builtin":@"$int"::@Int> size, "value">, _element_type>`): The underlying storage for the static tuple.

Functions:

`__init__`

```
__init__() -> Self
```

Constructs an empty (undefined) tuple.

Returns:

The tuple.

```
__init__(*elems: _element_type) -> Self
```

Constructs a static tuple given a set of arguments.

Args:

- **elems** (*`_element_type`): The element types.

Returns:

The tuple.

```
__init__(values: VariadicList[_element_type]) -> Self
```

Creates a tuple constant using the specified values.

Args:

- **values** (`VariadicList[_element_type]`): The list of values.

Returns:

A tuple with the values filled in.

```
__init__(array: array<#lit.struct.extract<!kgen.declref<_"$builtin":_"$int">:_Int> size,  
"value">, _element_type) -> Self
```

__getitem__

```
__getitem__[index: Int](self: Self) -> _element_type
```

Returns the value of the tuple at the given index.

Parameters:

- **index** (Int): The index into the tuple.

Returns:

The value at the specified position.

```
__getitem__(self: Self, index: Int) -> _element_type
```

Returns the value of the tuple at the given dynamic index.

Args:

- **index** (Int): The index into the tuple.

Returns:

The value at the specified position.

__setitem__

```
__setitem__[index: Int](inout self: Self, val: _element_type)
```

Stores a single value into the tuple at the specified index.

Parameters:

- **index** (Int): The index into the tuple.

Args:

- **val** (_element_type): The value to store.

```
__setitem__(inout self: Self, index: Int, val: _element_type)
```

Stores a single value into the tuple at the specified dynamic index.

Args:

- **index** (Int): The index into the tuple.
- **val** (_element_type): The value to store.

__len__

```
__len__(self: Self) -> Int
```

Returns the length of the array. This is a known constant value.

Returns:

The size of the list.

vector

Module

Defines several vector-like classes.

You can import these APIs from the `utils` package. For example:

```
from utils.vector import InlinedFixedVector
```

InlinedFixedVector

The `InlinedFixedVector` type is a dynamically-allocated vector with small- vector optimization.

The `InlinedFixedVector` does not resize or implement bounds checks, it is initialized with both a small-vector size (statically known) and a dynamic (not known at compile time) number of slots, and when it is deallocated, it frees its memory.

TODO: It should call its element destructors once we have traits.

This data structure is useful for applications where the number of required elements is not known at compile time, but once known at runtime, is guaranteed to be equal to or less than a certain capacity.

Parameters:

- **size** (Int): The statically known-small vector size.
- **type** (AnyType): The type of the elements.

Aliases:

- `static_size = _26x27_size`
- `static_data_type = StaticTuple[size, *"type"]`

Fields:

- **static_data** (StaticTuple[size, *"type"]): The underlying static storage, used for small vectors.
- **dynamic_data** (Pointer[*"type"]): The underlying dynamic storage, used to grow large vectors.
- **current_size** (Int): The number of elements in the vector.
- **capacity** (Int): The amount of elements that can fit in the vector without resizing it.

Functions:

`__init__`

```
__init__(inout self: Self, capacity: Int)
```

Constructs `InlinedFixedVector` with the given capacity.

The dynamically allocated portion is capacity - size.

Args:

- **capacity** (Int): The requested capacity of the vector.

copyinit

`copyinit_(inout self: Self, existing: Self)`

Creates a shallow copy (it doesn't copy the data).

Args:

- **existing** (`Self`): The `InlinedFixedVector` to copy.

getitem

`getitem_(self: Self, i: Int) -> *"type"`

Gets a vector element at the given index.

Args:

- **i** (`Int`): The index of the element.

Returns:

The element at the given index.

setitem

`setitem_(inout self: Self, i: Int, *value: "type")`

Sets a vector element at the given index.

Args:

- **i** (`Int`): The index of the element.
- **value** (`*"type"`): The value to assign.

deepcopy

`deepcopy(self: Self) -> Self`

Creates a deepcopy of this vector.

Returns:

The created copy of this vector.

append

`append(inout self: Self, *value: "type")`

Appends a value to this vector.

Args:

- **value** (`*"type"`): The value to append.

len

`len_(self: Self) -> Int`

Gets the number of elements in the vector.

Returns:

The number of elements in the vector.

clear

```
clear(inout self: Self)
```

Clears the elements in the vector.

UnsafeFixedVector

The `UnsafeFixedVector` type is a dynamically-allocated vector that does not resize or implement bounds checks.

It is initialized with a dynamic (not known at compile time) number of slots, and when it is deallocated, it frees its memory.

TODO: It should call its element destructors once we have traits.

This data structure is useful for applications where the number of required elements is not known at compile time, but once known at runtime, is guaranteed to be equal to or less than a certain capacity.

Parameters:

- **type** (`AnyType`): The type of the elements.

Fields:

- **data** (`Pointer[*"type"]`): The underlying storage for the vector.
- **size** (`Int`): The number of elements in the vector.
- **capacity** (`Int`): The amount of elements that can fit in the vector.

Functions:

__init__

```
__init__(inout self: Self, capacity: Int)
```

Constructs `UnsafeFixedVector` with the given capacity.

Args:

- **capacity** (`Int`): The requested capacity of the vector.

__copyinit__

```
__copyinit__(inout self: Self, existing: Self)
```

Creates a shallow copy (it doesn't copy the data).

Args:

- **existing** (`Self`): The `UnsafeFixedVector` to copy.

__getitem__

```
__getitem__(self: Self, i: Int) -> *"type"
```

Gets a vector element at the given index.

Args:

- **i** (Int): The index of the element.

Returns:

The element at the given index.

__setitem__

```
__setitem__(self: Self, i: Int, *value: "type")
```

Sets a vector element at the given index.

Args:

- **i** (Int): The index of the element.
- **value** (*"type"): The value to assign.

__len__

```
__len__(self: Self) -> Int
```

Gets the number of elements in the vector.

Returns:

The number of elements in the vector.

append

```
append(inout self: Self, *value: "type")
```

Appends a value to this vector.

Args:

- **value** (*"type"): The value to append.

clear

```
clear(inout self: Self)
```

Clears the elements in the vector.

DynamicVector

The `DynamicVector` type is a dynamically-allocated vector.

It supports pushing and popping from the back resizing the underlying storage as needed. When it is deallocated, it frees its memory.

TODO: It should call its element destructors once we have traits. TODO: It should perform bound checks.

Parameters:

- **type** (AnyType): The type of the elements.

Fields:

- **data** (Pointer[*"type"]): The underlying storage for the vector.

- **size** (Int): The number of elements in the vector.
- **capacity** (Int): The amount of elements that can fit in the vector without resizing it.

Functions:

__init__

```
__init__(inout self: Self)
```

Constructs an empty vector.

```
__init__(inout self: Self, capacity: Int)
```

Constructs a vector with the given capacity.

Args:

- **capacity** (Int): The requested capacity of the vector.

```
__init__(inout self: Self, pointer: Pointer[*"type"], size: Int)
```

Constructs a vector with the given pointer and size.

Args:

- **pointer** (Pointer[*"type"]): The pointer to the buffer.
- **size** (Int): The size of the buffer.

__copyinit__

```
__copyinit__(inout self: Self, existing: Self)
```

Creates a shallow copy (it doesn't copy the data).

Args:

- **existing** (Self): The DynamicVector to copy.

__getitem__

```
__getitem__(self: Self, i: Int) -> *"type"
```

Gets a vector element at the given index.

Args:

- **i** (Int): The index of the element.

Returns:

The element at the given index.

__setitem__

```
__setitem__(inout self: Self, i: Int, *value: "type")
```

Sets a vector element at the given index.

Args:

- **i** (Int): The index of the element.
- **value** (*"type"): The value to assign.

__len__

```
__len__(self: Self) -> Int
```

Gets the number of elements in the vector.

Returns:

The number of elements in the vector.

resize

```
resize(inout self: Self, size: Int)
```

Resizes the vector to the given new size.

If the new size is smaller than the current one, elements at the end are discarded. If the new size is larger than the current one, the vector is appended with non-initialized elements up to the requested size.

Args:

- **size** (Int): The new size.

deepcopy

```
deepcopy(self: Self) -> Self
```

Creates a deepcopy of this vector.

Returns:

The created copy of this vector.

reserve

```
reserve(inout self: Self, new_capacity: Int)
```

Reserves the requested capacity.

If the current capacity is greater or equal, this is a no-op. Otherwise, the storage is reallocated and the date is moved.

Args:

- **new_capacity** (Int): The new capacity.

push_back

```
push_back(inout self: Self, *value: "type")
```

Appends a value to this vector.

Args:

- **value** (*"type"): The value to append.

pop_back

```
pop_back(inout self: Self) -> *"type"
```

Pops a value from the back of this vector.

Returns:

The popped value.

clear

```
clear(inout self: Self)
```

Clears the elements in the vector.

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items

mojo

The Mojo command line interface.

Synopsis

```
mojo <command>
mojo [run-options] <path>
mojo [options]
mojo
```

Description

The `mojo` CLI provides all the tools you need for Mojo development, such as commands to run, compile, and package Mojo code. A list of all commands are listed below, and you can learn more about each one by adding the `--help` option to the command (for example, `mojo package --help`).

However, you may omit the `run` and `repl` commands. That is, you can run a Mojo file by simply passing the filename to `mojo`:

```
mojo hello.mojo
```

And you can start a REPL session by running `mojo` with no commands.

To update Mojo to the latest version, use the [modular tool](#):

```
modular update mojo
```

You can check your current version with `mojo --version`. For information about Mojo updates, see the [Mojo changelog](#).

Commands

`run` — Builds and executes a Mojo file.

`build` — Builds an executable from a Mojo file.

`repl` — Launches the Mojo REPL.

`debug` — Launches the LLDB debugger with support for debugging Mojo programs.

`package` — Compiles a Mojo package.

`format` — Formats Mojo source files.

`doc` — Compiles docstrings from a Mojo file.

`demangle` — Demangles the given name.

Options

Diagnostic options

`--version, -v`

Prints the Mojo version and exits.

Common options

--help, -h

Displays help information.

mojo build

Builds an executable from a Mojo file.

Synopsis

```
mojo build [options] <path>
```

Description

Compiles the Mojo file at the given path into an executable.

By default, the executable is saved to the current directory and named the same as the input file, but without a file extension.

Options

Output options

```
-o <PATH>
```

Sets the path and filename for the executable output. By default, it outputs the executable to the same location as the Mojo file, with the same name and no extension.

Compilation options

```
--no-optimization, -O0
```

Disables compiler optimizations. This might reduce the amount of time it takes to compile the Mojo source file. It might also reduce the runtime performance of the compiled executable.

```
--target-triple <TRIPLE>
```

Sets the compilation target triple. Defaults to the host target.

```
--target-cpu <CPU>
```

Sets the compilation target CPU. Defaults to the host CPU.

```
--target-features <FEATURES>
```

Sets the compilation target CPU features. Defaults to the host features.

```
-march <ARCHITECTURE>
```

Sets the architecture to generate code for.

```
-mcpu <CPU>
```

Sets the CPU to generate code for.

```
-mtune <TUNE>
```

Sets the CPU to tune code for.

-I <PATH>

Appends the given path to the list of directories to search for imported Mojo files.

-D <KEY=VALUE>

Defines a named value that can be used from within the Mojo source file being executed. For example, -D foo=42 defines a name `foo` that, when queried with the `sys.param_env` module from within the Mojo program, would yield the compile-time value 42.

--parsing-stdlib

Parses the input file(s) as the Mojo standard library.

Diagnostic options

--warn-missing-doc-strings

Emits warnings for missing or partial docstrings.

--max-notes-per-diagnostic <INTEGER>

When the Mojo compiler emits diagnostics, it sometimes also prints notes with additional information. This option sets an upper threshold on the number of notes that can be printed with a diagnostic. If not specified, the default maximum is 10.

Experimental compilation options

--debug-level <LEVEL>

Sets the level of debug info to use at compilation. The value must be one of: `none` (the default value), `line-tables`, or `full`. Please note that there are issues when generating debug info for some Mojo programs that have yet to be addressed.

--sanitize <CHECK>

Turns on runtime checks. The following values are supported: `address` (detects memory issues), and `thread` (detects multi-threading issues). Please note that these checks are not currently supported when executing Mojo programs.

--debug-info-language <LANGUAGE>

Sets the language to emit as part of the debug info. The supported languages are: `Mojo`, and `c`. `c` is the default, and is useful to enable rudimentary debugging and binary introspection in tools that don't understand Mojo.

Common options

--help, -h

Displays help information.

mojo debug

Launches the LLDB debugger with support for debugging Mojo programs.

Synopsis

```
mojo debug [debug-options]
```

Description

Launches the LLDB debugger with support for debugging programs written in Mojo, as well as other standard languages like C and C++. This feature is still experimental.

Options

Common options

--help, -h

Displays help information.

mojo demangle

Demangles the given name.

Synopsis

```
mojo demangle [options] <name>
```

Description

If the given name is a mangled Mojo symbol name, prints the demangled name. If no name is provided, one is read from standard input.

Options

Common options

--help, -h

Displays help information.

mojo doc

Compiles docstrings from a Mojo file.

Synopsis

```
mojo doc [options] <path>
```

Description

This is an early version of a documentation tool that generates an API reference from Mojo code comments. Currently, it generates a structured output of all docstrings into a JSON file, and it does not generate HTML. This output format is subject to change.

The input must be the path to a single Mojo source file.

Options

Output options

```
-o <PATH>
```

Sets the path and filename for the JSON output. If not provided, output is written to stdout.

Compilation options

```
-I <PATH>
```

Appends the given path to the list of directories that Mojo will search for any package/module dependencies. That is, if the file you pass to `mojo doc` imports any packages that do not reside in the local path and are not part of the Mojo standard library, use this to specify the path where Mojo can find those packages.

```
--parsing-stdlib
```

For internal use only.

Validation options

The following validation options help ensure that your docstrings use valid structure and meet other style criteria. By default, warnings are emitted only if the docstrings contain errors that prevent translation to the output format. (More options coming later.)

```
--warn-missing-doc-strings
```

Emits warnings for missing or partial docstrings.

Common options

```
--help, -h
```

Displays help information.

mojo format

Formats Mojo source files.

Synopsis

```
mojo format [options] <sources...>
```

Description

Formats the given set of Mojo sources using a Mojo-specific lint tool.

Options

Format options

```
--line-length <INTEGER>, -l <INTEGER>
```

Sets the max character line length. Default is 80.

Diagnostic options

```
--quiet, -q
```

Disables non-error messages.

Common options

```
--help, -h
```

Displays help information.

mojo package

Compiles a Mojo package.

Synopsis

```
mojo package [options] <path>
```

Description

Compiles a directory of Mojo source files into a binary package suitable to share and import into other Mojo programs and modules.

To create a Mojo package, first add an `__init__.mojo` file to your package directory. Then pass that directory name to this command, and specify the output path and filename with `-o`.

For more information, see [Mojo modules and packages](#).

Options

Output options

`-o <PATH>`

Sets the path and filename for the output package. The filename must end with either `.mojopkg` or `.mojo`. The filename given here defines the package name you can then use to import the code (minus the file extension). If you don't specify this option, output is written to stdout.

Compilation options

`--no-optimization, -O0`

Disables compiler optimizations. This might reduce the amount of time it takes to compile the Mojo source file. It might also reduce the runtime performance of the compiled executable.

`--target-triple <TRIPLE>`

Sets the compilation target triple. Defaults to the host target.

`--target-cpu <CPU>`

Sets the compilation target CPU. Defaults to the host CPU.

`--target-features <FEATURES>`

Sets the compilation target CPU features. Defaults to the host features.

`-march <ARCHITECTURE>`

Sets the architecture to generate code for.

`-mcpu <CPU>`

Sets the CPU to generate code for.

-mtune <TUNE>

Sets the CPU to tune code for.

-I <PATH>

Appends the given path to the list of directories to search for imported Mojo files.

-D <KEY=VALUE>

Defines a named value that can be used from within the Mojo source file being executed. For example, -D foo=42 defines a name `foo` that, when queried with the `sys.param_env` module from within the Mojo program, would yield the compile-time value 42.

--parsing-stdlib

Parses the input file(s) as the Mojo standard library.

Diagnostic options

--warn-missing-doc-strings

Emits warnings for missing or partial docstrings.

--max-notes-per-diagnostic <INTEGER>

When the Mojo compiler emits diagnostics, it sometimes also prints notes with additional information. This option sets an upper threshold on the number of notes that can be printed with a diagnostic. If not specified, the default maximum is 10.

Experimental compilation options

--debug-level <LEVEL>

Sets the level of debug info to use at compilation. The value must be one of: `none` (the default value), `line-tables`, or `full`. Please note that there are issues when generating debug info for some Mojo programs that have yet to be addressed.

--sanitize <CHECK>

Turns on runtime checks. The following values are supported: `address` (detects memory issues), and `thread` (detects multi-threading issues). Please note that these checks are not currently supported when executing Mojo programs.

--debug-info-language <LANGUAGE>

Sets the language to emit as part of the debug info. The supported languages are: `Mojo`, and `c`. `c` is the default, and is useful to enable rudimentary debugging and binary introspection in tools that don't understand Mojo.

Common options

--help, -h

Displays help information.

mojo repl

Launches the Mojo REPL.

Synopsis

```
mojo repl [lldb-options]
```

Description

Launches a Mojo read-evaluate-print loop (REPL) environment, which provides interactive development in the terminal. You can also start the REPL by simply running `mojo`.

Any number of options and arguments may be specified on the command line. These are then forwarded to the underlying `lldb` tool, which runs the REPL.

Options

Common options

--help, -h

Displays help information.

mojo run

Builds and executes a Mojo file.

Synopsis

```
mojo run [options] <path>
```

Description

Compiles the Mojo file at the given path and immediately executes it. Another way to execute this command is to simply pass a file to `mojo`. For example:

```
mojo hello.mojo
```

Options for this command itself, such as the ones listed below, must appear before the input file path argument. Any command line arguments that appear after the Mojo source file path are interpreted as arguments for that Mojo program.

Options

Compilation options

--no-optimization, -O0

Disables compiler optimizations. This might reduce the amount of time it takes to compile the Mojo source file. It might also reduce the runtime performance of the compiled executable.

--target-triple <TRIPLE>

Sets the compilation target triple. Defaults to the host target.

--target-cpu <CPU>

Sets the compilation target CPU. Defaults to the host CPU.

--target-features <FEATURES>

Sets the compilation target CPU features. Defaults to the host features.

-march <ARCHITECTURE>

Sets the architecture to generate code for.

-mcpu <CPU>

Sets the CPU to generate code for.

-mtune <TUNE>

Sets the CPU to tune code for.

-I <PATH>

Appends the given path to the list of directories to search for imported Mojo files.

-D <KEY=VALUE>

Defines a named value that can be used from within the Mojo source file being executed. For example, `-D foo=42` defines a name `foo` that, when queried with the `sys.param_env` module from within the Mojo program, would yield the compile-time value 42.

--parsing-stdlib

Parses the input file(s) as the Mojo standard library.

Diagnostic options

--warn-missing-doc-strings

Emits warnings for missing or partial docstrings.

--max-notes-per-diagnostic <INTEGER>

When the Mojo compiler emits diagnostics, it sometimes also prints notes with additional information. This option sets an upper threshold on the number of notes that can be printed with a diagnostic. If not specified, the default maximum is 10.

Experimental compilation options

--debug-level <LEVEL>

Sets the level of debug info to use at compilation. The value must be one of: `none` (the default value), `line-tables`, or `full`. Please note that there are issues when generating debug info for some Mojo programs that have yet to be addressed.

--sanitize <CHECK>

Turns on runtime checks. The following values are supported: `address` (detects memory issues), and `thread` (detects multi-threading issues). Please note that these checks are not currently supported when executing Mojo programs.

--debug-info-language <LANGUAGE>

Sets the language to emit as part of the debug info. The supported languages are: `Mojo`, and `c`. `c` is the default, and is useful to enable rudimentary debugging and binary introspection in tools that don't understand Mojo.

Common options

--help, -h

Displays help information.

Mojo[] roadmap & sharp edges

A summary of our Mojo plans, including upcoming features and things we need to fix.

This document captures the broad plan about how we plan to implement things in Mojo, and some early thoughts about key design decisions. This is not a full design spec for any of these features, but it can provide a “big picture” view of what to expect over time. It is also an acknowledgement of major missing components that we plan to add.

Overall priorities

Mojo is still in early development and many language features will arrive in the coming months. We are highly focused on building Mojo the right way (for the long-term), so we want to fully build-out the core Mojo language features before we work on other dependent features and enhancements.

Currently, that means we are focused on the core system programming features that are essential to [Mojo’s mission](#), and as outlined in the following sections of this roadmap.

In the near-term, we will **not** prioritize “general goodness” work such as:

- Adding syntactic sugar and short-hands for Python.
- Adding features from other languages that are missing from Python (such as public/private declarations).
- Tackling broad Python ecosystem challenges like packaging.

If you have encountered any bugs with current Mojo behavior, please [submit an issue on GitHub](#).

If you have ideas about how to improve the core Mojo features, we prefer that you first look for similar topics or start a new conversation about it in our [GitHub Discussions](#).

We also consider Mojo to be a new member of the Python family, so if you have suggestions to improve the experience with Python, we encourage you to propose these “general goodness” enhancements through the formal [PEP process](#).

Why not add syntactic sugar or other minor new features?

We are frequently asked whether Mojo will add minor features that people love in other languages but that are missing in Python, such as “implicit return” at the end of a function, public/private access control, fixing Python packaging, and various syntactic shorthands. As mentioned above, we are intentionally *not* adding these kinds of features to Mojo right now. There are three major reasons for this:

- First, Mojo is still young: we are still “building a house” by laying down major bricks in the type system and adding system programming features that Python lacks. We know we need to implement support for many existing Python features (compatibility a massive and important goal of Mojo) and this work is not done yet. We have limited engineering bandwidth and want focus on building essential functionality, and we will not debate whether certain syntactic sugar is important or not.
- Second, syntactic sugar is like mortar in a building—its best use is to hold the building together by filling in usability gaps. Sugar (and mortar) is problematic to add early into a system: you can run into problems with laying the next bricks because the sugar gets in the way. We have experience building other languages (such as Swift) that added sugar early, which could have been subsumed by more general features if time and care were given to broader evaluation.

- Third, the Python community should tackle some of these ideas first. It is important to us that Mojo be a good member of the Python family (a “Python++”), not just a language with Pythonic syntax. As such, we don’t want to needlessly diverge from Python evolution: adding a bunch of features could lead to problems down the road if Python makes incompatible decisions. Such a future would fracture the community which would cause massively more harm than any minor language feature could offset.

For all these reasons, “nice to have” syntactic sugar is not a priority, and we will quickly close such proposals to avoid cluttering the issue tracker. If you’d like to propose a “general goodness” syntactic feature, please do so with the existing [Python PEP process](#). If/when Python adopts a feature, Mojo will also add it, because Mojo’s goal is to be a superset. We are happy with this approach because the Python community is better equipped to evaluate these features, they have mature code bases to evaluate them with, and they have processes and infrastructure for making structured language evolution features.

Mojo SDK known issues

The [Mojo SDK](#) is still in early development and currently only available for Ubuntu Linux and macOS (Apple silicon) systems. Here are some of the notable issues that we plan to fix:

- Missing native support for Windows, Intel Macs, and Linux distributions other than Ubuntu. Currently, we support Ubuntu systems with x86-64 processors only. Support for more Linux distributions (including Debian and RHEL) and Windows is in progress.
- Python interoperability might fail when running a compiled Mojo program, with the message `Unable to locate a suitable libpython, please set MOJO_PYTHON_LIBRARY`. This is because we currently do not embed the Python version into the Mojo binary. For details and the workaround, see [issue #551](#).
- Modular CLI install might fail and require `modular clean` before you re-install.

If it asks you to perform auth, run `modular auth <MODULAR_AUTH>` and use the `MODULAR_AUTH` value shown for the `curl` command on [the download page](#).

- `modular install mojo` is slow and might appear unresponsive (as the installer is downloading packages in the background). We will add a progress bar in a future release.
- If you attempt to uninstall Mojo with `modular uninstall`, your subsequent attempt to install Mojo might fail with an HTTP 500 error code. If so, run `modular clean` and try again.
- Mojo REPL might hang (become unresponsive for more than 10 seconds) when interpreting an expression if your system has 4 GiB or less RAM. If you encounter this issue, please report it with your system specs.

Additionally, we’re aware of some issues that we might not be able to solve, but we mention them here with some more information:

- When installing Mojo, if you receive the error, `failed to reach URL https://cas.modular.com`, it could be because your network connection is behind a firewall. Try updating your firewall settings to allow access to these end points: `https://packages.modular.com` and `https://cas.modular.com`. Then retry with `modular clean` and `modular install mojo`.
- When installing Mojo, if you receive the error, `gpg: no valid OpenGPG data found`, this is likely because you are located outside our supported geographies. Due to US export control restrictions, we are unable to provide access to Mojo to users situated in specific countries.
- If using Windows Subsystem for Linux (WSL), you might face issues with WSL1. We recommend you upgrade to WSL2. To check the version, run `wsl -l -v`.

You can see other [reported issues on GitHub](#).

Small independent features

There are a number of features that are missing that are important to round out the language fully, but which don't depend strongly on other features. These include things like:

- Improved package management support.
- Many standard library features, including canonical arrays and dictionary types, copy-on-write data structures, etc.
- Support for “top level code” at file scope.
- Algebraic data types like `enum` in Swift/Rust, and pattern matching.
- Many standard library types, including `Optional[T]` and `Result[T, Error]` types when we have algebraic datatypes and basic traits.

Ownership and Lifetimes

The ownership system is partially implemented, and is expected to get built out in the next couple of months. The basic support for ownership includes features like:

- Capture declarations in closures.
- Borrow checker: complain about invalid mutable references.

The next step in this is to bring proper lifetime support in. This will add the ability to return references and store references in structures safely. In the immediate future, one can use the unsafe `Pointer` struct to do this like in C++.

Protocols / Traits

Unlike C++, Mojo does not “instantiate templates” in its parser. Instead, it has a separate phase that works later in the compilation pipeline (the “Elaborator”) that instantiates parametric code, which is aware of autotuning and caching. This means that the parser has to perform full type checking and IR generation without instantiating algorithms.

The planned solution is to implement language support for Protocols - variants of this feature exist in many languages (e.g. Swift protocols, Rust traits, Haskell typeclasses, C++ concepts) all with different details. This feature allows defining requirements for types that conform to them, and dovetails into static and dynamic metaprogramming features.

Classes

Mojo still doesn’t support classes, the primary thing Python programmers use pervasively! This isn’t because we hate dynamism - quite the opposite. It is because we need to get the core language semantics nailed down before adding them. We expect to provide full support for all the dynamic features in Python classes, and want the right framework to hang that off of.

When we get here, we will discuss what the right default is: for example, is full Python hash-table dynamism the default? Or do we use a more efficient model by default (e.g. vtable-based dispatch and explicitly declared stored properties) and allow opt’ing into dynamism with a `@dynamic` decorator on the class. The latter approach worked well for Swift (its [@objc attribute](#)), but we’ll have to prototype to better understand the tradeoffs.

C/C++ Interop

Integration to transparently import Clang C/C++ modules. Mojo’s type system and C++’s are pretty compatible, so we should be able to have something pretty nice here. Mojo can leverage Clang to transparently generate a foreign function interface between C/C++ and Mojo, with the ability to directly import functions:

```
from "math.h" import cos  
print(cos(0))
```

Full MLIR decorator reflection

All decorators in Mojo have hard-coded behavior in the parser. In time, we will move these decorators to being compile-time metaprograms that use MLIR integration. This may depend on C++ interop for talking to MLIR. This completely opens up the compiler to programmers. Static decorators are functions executed at compile-time with the capability to inspect and modify the IR of functions and types.

```
fn value(t: TypeSpec):  
    t.__copyinit__ = # synthesize dunder copyinit automatically  
  
@value  
struct TrivialType: pass  
  
fn full_unroll(loop: mlir.Operation):  
    # unrolling of structured loop  
  
fn main():  
    @full_unroll  
    for i in range(10):  
        print(i)
```

Sharp Edges

The entire Modular kernel library is written in Mojo, and its development has been prioritized based on the internal needs of those users. Given that Mojo is still a young language, there are a litany of missing small features that many Python and systems programmers may expect from their language, as well as features that don't quite work the way we want to yet, and in ways that can be surprising or unexpected. This section of the document describes a variety of "sharp edges" in Mojo, and potentially how to work around them if needed. We expect all of these to be resolved in time, but in the meantime, they are documented here.

No list or dict comprehensions

Mojo does not yet support Python list or dictionary comprehension expressions, like `[x for x in range(10)]`, because Mojo's standard library has not yet grown a standard list or dictionary type.

No lambda syntax

Mojo does not yet support defining anonymous functions with the `lambda` keyword.

No parametric aliases

Mojo aliases can refer to parametric values but cannot themselves be a parameter. We would like this example to work, however:

```
alias Scalar[dt: DType] = SIMD[dt, 1]  
alias mul2[x: Int] = x * 2
```

Exception is actually called Error

In Python, programmers expect that exceptions all subclass the `Exception` builtin class. The only available type for Mojo "exceptions" is `Error`:

```
fn raise_an_error() raises:  
    raise Error("I'm an error!")
```

The reason we call this type `Error` instead of `Exception` is because it's not really an exception. It's not an exception, because raising an error does not cause stack unwinding, but most importantly it does not have a stack trace. And without polymorphism, the `Error` type is the only kind of error that can be raised in Mojo right now.

No Python-style generator functions

Mojo does not yet support Python-style generator functions (`yield` syntax). These are “synchronous co-routines” – functions with multiple suspend points.

No `async for` or `async with`

Although Mojo has support for `async` functions with `async fn` and `async def`, Mojo does not yet support the `async for` and `async with` statements.

The `rebind` builtin

One of the consequences of Mojo not performing function instantiation in the parser like C++ is that Mojo cannot always figure out whether some parametric types are equal and complain about an invalid conversion. This typically occurs in static dispatch patterns, like:

```
fn take_simd8(x: SIMD[DType.float32, 8]): pass

fn generic_simd[nelts: Int](x: SIMD[DType.float32, nelts]):
    @parameter
    if nelts == 8:
        take_simd8(x)___
```

The parser will complain,

```
error: invalid call to 'take_simd8': argument #0 cannot be converted from
'SIMD[f32, nelts]' to 'SIMD[f32, 8]'
    take_simd8(x)
~~~~~^~~
```

This is because the parser fully type-checks the function without instantiation, and the type of `x` is still `SIMD[f32, nelts]`, and not `SIMD[f32, 8]`, despite the static conditional. The remedy is to manually “rebind” the type of `x`, using the `rebind` builtin, which inserts a compile-time assert that the input and result types resolve to the same type after function instantiation.

```
fn generic_simd[nelts: Int](x: SIMD[DType.float32, nelts]):
    @parameter
    if nelts == 8:
        take_simd8(rebind[SIMD[DType.float32, 8]](x))___
```

Scoping and mutability of statement variables

Python programmers understand that local variables are implicitly declared and scoped at the function level. As the programming manual explains, this feature is supported in Mojo only inside `def` functions. However, there are some nuances to Python’s implicit declaration rules that Mojo does not match 1-to-1.

For example, the scope of `for` loop iteration variables and caught exceptions in `except` statements is limited to the next indentation block, for both `def` and `fn` functions. Python programmers will expect the following program to print “2”:

```
for i in range(3): pass
print(i)___
```

However, Mojo will complain that `print(i)` is a use of an unknown declaration. This is because whether `i` is defined at this line is dynamic in Python. For instance the following Python program will fail:

```
for i range(0): pass  
print(i)___
```

With `NameError: name 'i' is not defined`, because the definition of `i` is a dynamic characteristic of the function. Mojo's lifetime tracker is intentionally simple (so lifetimes are easy to use!), and cannot reason that `i` would be defined even when the loop bounds are constant.

Also stated in the programming manual: in `def` functions, the function arguments are mutable and re-assignable, whereas in `fn`, function arguments are rvalues and cannot be re-assigned. The same logic extends to statement variables, like `for` loop iteration variables or caught exceptions:

```
def foo():  
    try:  
        bad_function()  
    except e:  
        e = Error() # ok: we can overwrite 'e'  
  
fn bar():  
    try:  
        bad_function()  
    except e:  
        e = Error() # error: 'e' is not mutable___
```

Name scoping of nested function declarations

In Python, nested function declarations produce dynamic values. They are essentially syntax sugar for `bar = lambda ...`.

```
def foo():  
    def bar(): # creates a function bound to the dynamic value 'bar'  
        pass  
    bar() # indirect call___
```

In Mojo, nested function declarations are static, so calls to them are direct unless made otherwise.

```
fn foo():  
    fn bar(): # static function definition bound to 'bar'  
        pass  
    bar() # direct call  
    let f = bar # materialize 'bar' as a dynamic value  
    f() # indirect call___
```

Currently, this means you cannot declare two nested functions with the same name. For instance, the following example does not work in Mojo:

```
def pick_func(cond):  
    if cond:  
        def bar(): return 42  
    else:  
        def bar(): return 3 # error: redeclaration of 'bar'  
    return bar___
```

The functions in each conditional must be explicitly materialized as dynamic values.

```
def pick_func(cond):  
    let result: def() capturing # Mojo function type  
    if cond:  
        def bar0(): return 42  
        result = bar0  
    else:  
        def bar1(): return 3 # error: redeclaration of 'bar'  
        result = bar1  
    return result___
```

We hope to sort out these oddities with nested function naming as our model of closures in Mojo develops further.

No polymorphism

Mojo will implement static polymorphism through traits/protocols in the near future and dynamic polymorphism through classes and MLIR reflection. None of those things exist today, which presents several limitations to the language.

Python programmers are used to implementing special dunder methods on their classes to interface with generic methods like `print` and `len`. For instance, one expects that implementing `__repr__` or `__str__` on a class will enable that class to be printed via `print`.

```
class One:  
    def __init__(self): pass  
    def __repr__(self): return '1'  
  
print(One()) # prints '1'
```

This is not currently possible in Mojo. Overloads of `print` are provided for common types, like `Int` and `SIMD` and `String`, but otherwise the builtin is not extensible. Overloads also have the limitation that they must be defined within the same module, so you cannot add a new overload of `print` for your struct types.

The same extends to `range`, `len`, and other builtins.

No lifetime tracking inside collections

Due to the aforementioned lack of polymorphism, collections like lists, maps, and sets are unable to invoke element destructors. For collections of trivial types, like `DynamicVector[Int]`, this is no problem, but for collections of types with lifetimes, like `DynamicVector[String]`, the elements have to be manually destructed. Doing so requires quite an ugly pattern, shown in the next section.

No safe value references

Mojo does not have proper lifetime marker support yet, and that means it cannot reason about returned references, so Mojo doesn't support them. You can return or keep unsafe references by passing explicit pointers around.

```
struct StringRef:  
    var ref: Pointer[SI8]  
    var size: Int  
    # ...  
  
fn bar(x: StringRef): pass  
  
fn foo():  
    let s: String = "1234"  
    let ref: StringRef = s # unsafe reference  
    bar(ref)  
    _ = s # keep the backing memory alive!
```

Mojo will destruct objects as soon as it thinks it can. That means the lifetime of objects to which there are unsafe references must be manually extended. See the [lifetime document](#) for more details. This disables the RAI pattern in Mojo. Context managers and `with` statements are your friends in Mojo.

No lvalue returns also mean that implementing certain patterns require magic keywords until proper lifetime support is built. One such pattern is retrieving an unsafe reference from an object.

```
struct UnsafeIntRef:  
    var ptr: Pointer[Int]  
  
fn printIntRef(x: UnsafeIntRef):  
    # "dereference" operator  
    print(__get_address_as_lvalue(x.ptr)) # Pointer[Int] -> &Int
```

```
var c: Int = 10
# "reference" operator
let ref = UnsafeIntRef(__get_lvalue_as_address(c)) # &Int -> Pointer[Int]
```

Parameter closure captures are unsafe references

You may have seen nested functions, or “closures”, annotated with the `@parameter` decorator. This creates a “parameter closure”, which behaves differently than a normal “stateful” closure. A parameter closure declares a compile-time value, similar to an alias declaration. That means parameter closures can be passed as parameters:

```
fn take_func[f: fn() capturing -> Int]():
    pass

fn call_it(a: Int):
    @parameter
    fn inner() -> Int:
        return a # capture 'a'

    take_func[inner]() # pass 'inner' as a parameter
```

Parameter closures can even be parametric and capturing:

```
fn take_func[f: fn[a: Int]() capturing -> Int]():
    pass

fn call_it(a: Int):
    @parameter
    fn inner[b: Int]() -> Int:
        return a + b # capture 'a'

    take_func[inner]() # pass 'inner' as a parameter
```

However, note that parameter closures are always capture by *unsafe* reference. Mojo’s lifetime tracking is not yet sophisticated enough to form safe references to objects (see above section). This means that variable lifetimes need to be manually extended according to the lifetime of the parametric closure:

```
fn print_it[f: fn() capturing -> String]():
    print(f())

fn call_it():
    let s: String = "hello world"
    @parameter
    fn inner() -> String:
        return s # 's' captured by reference, so a copy is made here
    # lifetime tracker destroys 's' here

    print_it[inner]() # crash! 's' has been destroyed
```

The lifetime of the variable can be manually extended by discarding it explicitly.

```
fn call_it():
    let s: String = "hello world"
    @parameter
    fn inner() -> String:
        return s

    print_it[inner]()
    _ = s^ # discard 's' explicitly
```

A quick note on the behaviour of “stateful” closures. One sharp edge here is that stateful closures are *always* capture-by-copy; Mojo lacks syntax for move-captures and the lifetime tracking necessary for capture-by-reference. Stateful closures are runtime values – they cannot be passed as parameters, and they cannot be parametric. However, a nested function is promoted to a parametric closure if it does not capture anything. That is:

```
fn foo0[f: fn() capturing -> String](): pass
fn foo1[f: fn[a: Int]() capturing -> None](): pass
```

```

fn main():
    let s: String = "hello world"
    fn stateful_captures() -> String:
        return s # 's' is captured by copy

foo0[stateful_captures]() # not ok: 'stateful_captures' is not a parameter
fn stateful_nocapture[a: Int](): # ok: can be parametric, since no captures
    print(a)

fool[stateful_nocapture]() # ok: 'stateful_nocapture' is a parameter

```

The standard library has limited exceptions use

For historic and performance reasons, core standard library types typically do not use exceptions. For instance, `DynamicVector` will not raise an out-of-bounds access (it will crash), and `Int` does not throw on divide by zero. In other words, most standard library types are considered “unsafe”.

```

let v = DynamicVector[Int](0)
print(v[1]) # could crash or print garbage values (undefined behaviour)

print(1//0) # does not raise and could print anything (undefined behaviour)

```

This is clearly unacceptable given the strong memory safety goals of Mojo. We will circle back to this when more language features and language-level optimizations are available.

Nested functions cannot be recursive

Nested functions (any function that is not a top-level function) cannot be recursive in any way. Nested functions are considered “parameters”, and although parameter values do not have to obey lexical order, their uses and definitions cannot form a cycle. Current limitations in Mojo mean that nested functions, which are considered parameter values, cannot be cyclic.

```

fn try_recursion():
    fn bar(x: Int): # error: circular reference :(
        if x < 10:
            bar(x + 1)

```

Only certain loaded MLIR dialects can be accessed

Although Mojo provides features to access the full power of MLIR, in reality only a certain number of loaded MLIR dialects can be accessed in the Playground at the moment.

The upstream dialects available in the Playground are the [index](#) dialect and the [LLVM](#) dialect.

Mojo changelog

A history of significant Mojo changes.

This is a running list of significant changes for the Mojo language and tools. It doesn't include all internal implementation changes.

Update Mojo

If you don't have Mojo yet, see the [get started guide](#).

To see your current Mojo version, run this:

```
mojo --version
```

To update Mojo to the latest release, run this:

```
modular update mojo
```

However, if your current version is 0.3.0 or lower, you'll need these additional commands:

```
sudo apt-get update
sudo apt-get install modular
modular clean
modular install mojo
```

v0.4.0 for Mac (2023-10-19)

Legendary

- Mojo for Mac!

The Mojo SDK now works on macOS (Apple silicon). This is the same version previously released for Linux. Get the latest version of the SDK for your Mac system:

[Download Now!](#)

v0.4.0 (2023-10-05)

New

- Mojo now supports default parameter values. For example:

```
fn foo[a: Int = 3, msg: StringLiteral = "woof"]():
    print(msg, a)

fn main():
    foo() # prints 'woof 3'
    foo[5]() # prints 'woof 5'
    foo[7, "meow"]() # prints 'meow 7'
```

Inferred parameter values take precedence over defaults:

```
@value
struct Bar[v: Int]:
    pass

fn foo[a: Int = 42, msg: StringLiteral = "quack"](bar: Bar[a]):
    print(msg, a)

fn main():
    foo(Bar[9]()) # prints 'quack 9'
```

Structs also support default parameters:

```
@value
struct DefaultParams[msg: StringLiteral = "woof"]:
    alias message = msg

fn main():
    print(DefaultParams[]().message) # prints 'woof'
    print(DefaultParams["meow"]().message) # prints 'meow'
```

- The new [file](#) module adds basic file I/O support. You can now write:

```
var f = open("my_file.txt", "r")
print(f.read())
f.close()
```

or

```
with open("my_file.txt", "r") as f:
    print(f.read())
```

- Mojo now allows context managers to support an `_enter_` method without implementing support for an `_exit_` method, enabling idioms like this:

```
# This context manager consumes itself and returns it as the value.
fn _enter_(owned self) -> Self:
    return self^
```

Here Mojo *cannot* invoke a `noop __exit__` method because the context manager is consumed by the `__enter__` method. This can be used for types (like file descriptors) that are traditionally used with `with` statements, even though Mojo's guaranteed early destruction doesn't require that.

- A very basic version of `pathlib` has been implemented in Mojo. The module will be improved to achieve functional parity with Python in the next few releases.
- The `memory.unsafe` module now contains a `bitcast` function. This is a low-level operation that enables bitcasting between pointers and scalars.
- The input parameters of a parametric type can now be directly accessed as attribute references on the type or variables of the type. For example:

```
@value
struct Thing[param: Int]:
    pass

fn main():
    print(Thing[2].param) # prints '2'
    let x = Thing[9]()
    print(x.param) # prints '9'
```

Input parameters on values can even be accessed in parameter contexts. For example:

```
fn foo[value: Int]():
    print(value)

let y = Thing[12]()
alias constant = y.param + 4
foo[constant]() # prints '16'
```

- The Mojo REPL now supports code completion. Press `Tab` while typing to query potential completion results.
- Error messages from Python are now exposed in Mojo. For example the following should print No module named '`my_uninstalled_module`':

```
fn main():
    try:
        let my_module = Python.import_module("my_uninstalled_module")
    except e:
        print(e)
```

- Error messages can now store dynamic messages. For example, the following should print "Failed on: Hello"

```
fn foo(x:String) raises:
    raise Error("Failed on: " + x)

fn main():
    try:
        foo("Hello")
    except e:
        print(e)
```

□ Changed

- We have improved and simplified the `parallelize` function. The function now elides some overhead by caching the Mojo parallel runtime.
- The Mojo REPL and Jupyter environments no longer implicitly expose `Python`, `PythonObject`, or `Pointer`. These symbols must now be imported explicitly, for example:

```
from python import Python
from python.object import PythonObject
from memory.unsafe import Pointer
```

- The syntax for specifying attributes with the `__mlir_op` prefix have changed to mimic Python's keyword argument passing syntax. That is, `=` should be used instead of `:`, e.g.:

```
# Old syntax, now fails.
__mlir_op.`index.bool.constant`[value : __mlir_attr.`false`]()
# New syntax.
__mlir_op.`index.bool.constant`[value=__mlir_attr.`false`()]
```

- You can now print the `Error` object directly. The `message()` method has been removed.

□ Fixed

- [#794](#) - Parser crash when using the `in` operator.
- [#936](#) - The `Int` constructor now accepts other `Int` instances.
- [#921](#) - Better error message when running `mojo` on a module with no `main` function.
- [#556](#) - `UInt64s` are now printed correctly.
- [#804](#) - Emit error instead of crashing when passing variadic arguments of unsupported types.
- [#833](#) - Parser crash when assigning module value.
- [#752](#) - Parser crash when calling `async def`.
- [#711](#) - The overload resolution logic now correctly prioritizes instance methods over static methods (if candidates are an equally good match otherwise), and no longer crashed if a static method has a `Self` type as its first argument.
- [#859](#) - Fix confusing error and documentation of the `rebind` builtin.
- [#753](#) - Direct use of LLVM dialect produces strange errors in the compiler.
- [#926](#) - Fixes an issue that occurred when a function with a return type of `StringRef` raised an error. When the function raised an error, it incorrectly returned the string value of that error.
- [#536](#) - Report More information on python exception.

v0.3.1 (2023-09-28)

Our first-ever patch release of the Mojo SDK is here! Release v0.3.1 includes primarily installation-related fixes. If you've had trouble installing the previous versions of the SDK, this release may be for you.

Fixed

- [#538](#) - Installation hangs during the testing phase. This issue occurs on machines with a low number of CPU cores, such as free AWS EC2 instances and GitHub Codespaces.
- [#590](#) - Installation fails with a “failed to run python” message.
- [#672](#) - Language server hangs on code completion. Related to #538, this occurs on machines with a low number of CPU cores.
- [#913](#) - In the REPL and Jupyter notebooks, inline comments were being parsed incorrectly.

v0.3.0 (2023-09-21)

There's more Mojo to love in this, the second release of the Mojo SDK! This release includes new features, an API change, and bug fixes.

There's also an updated version of the [Mojo extension for VS Code](#).

New

- Mojo now has partial support for passing keyword arguments to functions and methods. For example the following should work:

```
fn foo(a: Int, b: Int = 3) -> Int:  
    return a * b  
  
fn main():  
    print(foo(6, b=7))  # prints '42'  
    print(foo(a=6, b=7))  # prints '42'  
    print(foo(b=7, a=6))  # prints '42' □
```

Parameters can also be inferred from keyword arguments, for example:

```
fn bar[A: AnyType, B: AnyType](a: A, b: B):  
    print("Hello □")  
  
fn bar[B: AnyType](a: StringLiteral, b: B):  
    print(a)  
  
fn main():  
    bar(1, 2)  # prints `Hello □`  
    bar(b=2, a="Yay!")  # prints `Yay!` □
```

For the time being, the following notable limitations apply:

- Keyword-only arguments are not supported:

```
fn baz(*args: Int, b: Int): pass  # fails  
fn baz(a: Int, *, b: Int): pass  # fails□
```

(Keyword-only arguments are described in [PEP 3102](#).)
- Variadic keyword arguments are not supported:

```
fn baz(a: Int, **kwargs: Int): pass  # fails□
```
- Mojo now supports the `@nonmaterializable` decorator. The purpose is to mark data types that should only exist in the parameter domain. To use it, a struct is decorated with `@nonmaterializable(TargetType)`. Any time the nonmaterializable type is converted from the parameter domain, it is automatically converted to `TargetType`. A nonmaterializable struct should have all of its methods annotated as `@always_inline`, and must be computable in the parameter domain. In the following example, the `NmStruct` type can be added in the parameter domain, but are converted to `HasBool` when materialized.

```
@value  
@register_passable("trivial")  
struct HasBool:  
    var x: Bool  
    fn __init__(x: Bool) -> Self:  
        return Self {x: x}  
    @always_inline("nodebug")  
    fn __init__(nms: NmStruct) -> Self:  
        return Self {x: True if (nms.x == 77) else False}  
  
@value  
@nonmaterializable(HasBool)  
@register_passable("trivial")  
struct NmStruct:  
    var x: Int  
    @always_inline("nodebug")  
    fn __add__(self: Self, rhs: Self) -> Self:  
        return NmStruct(self.x + rhs.x)  
  
alias stillNmStruct = NmStruct(1) + NmStruct(2)  
# When materializing to a run-time variable, it is automatically converted,  
# even without a type annotation.  
let convertedToHasBool = stillNmStruct □
```

- Mojo integer literals now produce the `IntLiteral` infinite precision integer type when used in the parameter domain. `IntLiteral` is materialized to the `Int` type for runtime computation, but intermediate computations at compile time, using supported operators, can now exceed the bit width of the `Int` type.
- The Mojo Language Server now supports top-level code completions, enabling completion when typing a reference to a variable, type, etc. This resolves [#679](#).

- The Mojo REPL now colorizes the resultant variables to help distinguish input expressions from the output variables.

Changed

- Mojo allows types to implement two forms of move constructors, one that is invoked when the lifetime of one value ends, and one that is invoked if the compiler cannot prove that. These were previously both named `_moveinit_`, with the following two signatures:

```
fn __moveinit__(inout self, owned existing: Self): ...
fn __moveinit__(inout self, inout existing: Self): ...
```

We've changed the second form to get its own name to make it more clear that these are two separate operations: the second has been renamed to `_takeinit_`:

```
fn __moveinit__(inout self, owned existing: Self): ...
fn __takeinit__(inout self, inout existing: Self): ...
```

The name is intended to connote that the operation takes the conceptual value from the source (without destroying it) unlike the first one which "moves" a value from one location to another.

- The Error type in Mojo has changed. Instead of extracting the error message using `error.value` you will now extract the error message using `error.message()`.

For more information, see [Unique "move-only" types](#) in the Mojo docs.

Fixed

- [#503](#) - Improve error message for failure lowering `kgen.param.constant`.
- [#554](#) - Alias of static tuple fails to expand.
- [#500](#) - Call expansion failed due to verifier error.
- [#422](#) - Incorrect comment detection in multiline strings.
- [#729](#) - Improve messaging on how to exit the REPL.
- [#756](#) - Fix initialization errors of the VS Code extension.
- [#575](#) - Build LLDB/REPL with libedit for a nicer editing experience in the terminal.

v0.2.1 (2023-09-07)

The first versioned release of Mojo! ☺

All earlier releases were considered version 0.1.

Legendary

- First release of the Mojo SDK!

You can now develop with Mojo locally. The Mojo SDK is currently available for Ubuntu Linux systems, and support for Windows and macOS is coming soon. You can still develop from a Windows or Mac computer using a container or remote Linux system.

The Mojo SDK includes the Mojo standard library and the [Mojo command-line interface](#) (CLI), which allows you to run, compile, and package Mojo code. It also provides a REPL programming environment.

[Get the Mojo SDK!](#)

- First release of the [Mojo extension for VS Code](#).

This provides essential Mojo language features in Visual Studio Code, such as code completion, code quick fixes, docs tooltips, and more. Even when developing on a remote system, using VS Code with this extension provides a native-like IDE experience.

New

- A new `clobber_memory` function has been added to the [benchmark](#) module. The clobber memory function tells the system to flush all memory operations at the specified program point. This allows you to benchmark operations without the compiler reordering memory operations.
- A new `keep` function has been added to the [benchmark](#) module. The `keep` function tries to tell the compiler not to optimize the variable away if not used. This allows you to avoid compiler's dead code elimination mechanism, with a low footprint side effect.
- New `shift_right` and `shift_left` functions have been added to the [simd](#) module. They shift the elements in a SIMD vector right/left, filling elements with zeros as needed.
- A new `cumsum` function has been added to the [reduction](#) module that computes the cumulative sum (also known as scan) of input elements.
- Mojo Jupyter kernel now supports code completion.

Changed

- Extends `rotate_bits_left`, `rotate_left`, `rotate_bits_right`, and `rotate_right` to operate on `Int` values. The ordering of parameters has also been changed to enable type inference. Now it's possible to write `rotate_right[shift_val](simd_val)` and have the `dtype` and `simd_width` inferred from the argument. This addresses [Issue #528](#).

Fixed

- Fixed a bug causing the parser to crash when the `with` statement was written without a colon. This addresses [Issue #529](#).
- Incorrect imports no longer crash when there are other errors at the top level of a module. This fixes [Issue #531](#).

August 2023

2023-08-24

- Fixed issue where the `with expr as x` statement within `fn` behaved as if it were in a `def`, binding `x` with function scope instead of using lexical scope.

□ New

- Major refactoring of the standard library to enable packaging and better import ergonomics:
 - The packages are built as binaries to improve startup speed.
 - Package and module names are now lowercase to align with the Python style.
 - Modules have been moved to better reflect the purpose of the underlying functions (e.g. `Pointer` is now within the `unsafe` module in the `memory` package).
 - The following modules are now included as built-ins: `SIMD`, `DType`, `I0`, `Object`, and `String`. This means it's no longer necessary to explicitly import these modules. Instead, these modules will be implicitly imported for the user. Private methods within the module are still accessible using the `builtin.module_name._private_method` import syntax.
 - New `math` package has been added to contain the `bit`, `math`, `numerics`, and `polynomial` modules. The contents of the `math.math` module are re-exported into the `math` package.
- Mojo now supports using memory-only types in parameter expressions and as function or type parameters:

```
@value
struct IntPair:
    var first: Int
    var second: Int

fn add_them[value: IntPair]() -> Int:
    return value.first + value.second

fn main():
    print(add_them[IntPair(1, 2)]()) # prints '3'
```

- In addition, Mojo supports evaluating code that uses heap-allocated memory at compile-time and materializing compile-time values with heap-allocated memory into dynamic values:

```
fn fillVector(lowerBound: Int, upperBound: Int, step: Int) -> DynamicVector[Int]:
    var result = DynamicVector[Int]()
    for i in range(lowerBound, upperBound, step):
        result.push_back(i)
    return result

fn main():
    alias values = fillVector(5, 23, 7)
    for i in range(0, values._len_()):
        print(values[i]) # prints '5', '12', and then '19'
```

□ Changed

- `def main():`, without the explicit `None` type, can now be used to define the entry point to a Mojo program.
- The `assert_param` function has been renamed to `constrained` and is now a built-in function.
- The `print` function now works on Complex values.

□ Fixed

- Fixed issues with `print` formatting for `DType.uint16` and `DType.int16`.
- [Issue #499](#) - Two new `rotate_right` and `rotate_left` functions have been added to the `SIMD` module.
- [Issue #429](#) - You can now construct a `Bool` from a `SIMD` type whose element-type is `DType.bool`.
- [Issue #350](#) - Confusing Matrix implementation
- [Issue #349](#) - Missing `load_tr` in struct `Matrix`
- [Issue #501](#) - Missing syntax error messages in Python expressions.

2023-08-09

□ Changed

- The `ref` and `mutref` identifiers are now treated as keywords, which means they cannot be used as variable, attribute, or function names. These keywords are used by the “lifetimes” features, which is still in development. We can consider renaming these (as well as other related keywords) when the development work gels, support is enabled in public Mojo builds, and when we have experience using them.
- The argument handling in `def` functions has changed: previously, they had special behavior that involved mutable copies in the callee. Now, we have a simple rule, which is that `def` argument default to the `owned` convention (`fn` arguments still default to the `borrowed` convention).

This change is mostly an internal cleanup and simplification of the compiler and argument model, but does enable one niche use-case: you can now pass non-copyable types to `def` arguments by transferring ownership of a value into the `def` call. Before, that would not be possible because the copy was made on the callee side, not the caller's side. This also allows the explicit use of the `borrowed` keyword with a `def` that wants to opt-in to that behavior.

2023-08-03

□ New

- A new [Tensor](#) type has been introduced. This tensor type manages its own data (unlike `NDBuffer` and `Buffer` which are just views). Therefore, the tensor type performs its own allocation and free. Here is a simple example of using the tensor type to represent an RGB image and convert it to grayscale:

```
from tensor import Tensor, TensorShape
from utils.index import Index
from random import rand

let height = 256
let width = 256
let channels = 3

# Create the tensor of dimensions height, width, channels and fill with
# random value.
let image = rand[DTType.float32](height, width, channels)

# Declare the grayscale image.
var gray_scale_image = Tensor[DTType.float32](height, width)

# Perform the RGB to grayscale transform.
for y in range(height):
    for x in range(width):
        let r = image[y,x,0]
        let g = image[y,x,1]
        let b = image[y,x,2]
        gray_scale_image[Index(y,x)] = 0.299 * r + 0.587 * g + 0.114 * b
```

□ Fixed

- [Issue #53](#) - Int now implements true division with the `/` operator. Similar to Python, this returns a 64-bit floating point number. The corresponding in-place operator, `/=`, has the same semantics as `//=`.

July 2023

2023-07-26

□ New

- Types that define both `__getitem__` and `__setitem__` (i.e. where sub-scripting instances creates computed LValues) can now be indexed in parameter expressions.
- Unroll decorator for loops with constant bounds and steps:
 - `@unroll`: Fully unroll a loop.
 - `@unroll(n)`: Unroll a loop by factor of `n`, where `n` is a positive integer.
 - Unroll decorator requires loop bounds and iteration step to be compiler time constant value, otherwise unrolling will fail with compilation error. This also doesn't make loop induction variable a parameter.

```
# Fully unroll the loop.
@unroll
for i in range(5):
    print(i)

# Unroll the loop by a factor of 4 (with remainder iterations of 2).
@unroll(4)
for i in range(10):
    print(i)
```

- The Mojo REPL now prints the values of variables defined in the REPL. There is full support for scalars and structs. Non-scalar SIMD vectors are not supported at this time.

□ Fixed

- [Issue #437](#) - Range can now be instantiated with a `PyObject`.
- [Issue #288](#) - Python strings can now be safely copied.

2023-07-20

□ New

- Mojo now includes a `Limits` module, which contains functions to get the max and min values representable by a type, as requested in [Issue #51](#). The following functions moved from `Math` to `Limits`: `inf()`, `neginf()`, `isinf()`, `isfinite()`.
- Mojo decorators are now distinguished between “signature” and “body” decorators and are ordered. Signature decorators, like `@register_passable` and `@parameter`, modify the type of declaration before the body is parsed. Body decorators, like `@value`, modify the body of declaration after it is fully parsed. Due to ordering, a signature decorator cannot be applied after a body decorator. That means the following is now invalid:

```
@register_passable # error: cannot apply signature decorator after a body one!
@value
struct Foo:
    pass
```

- Global variables can now be exported in Mojo compiled archives, using the `@export` decorator. Exported global variables are public symbols in compiled archives and use the variable name as its linkage name, by default. A custom linkage name can be specified with `@export("new_name")`. This does not affect variable names in Mojo code.

- Mojo now supports packages! A Mojo package is defined by placing an `__init__.mojo` or `__init__.moj` within a directory. Other files in the same directory form modules within the package (this works exactly like it does [in Python](#)). Example:

```
main.□
my_package/
  __init__.□
  module.□
  my_other_package/
    __init__.□
    stuff.□
      □

# main.□
from my_package.module import some_function
from my_package.my_other_package.stuff import SomeType

fn main():
    var x: SomeType = some_function()□

import builtin.io as io
import SIMD

io.print("hello world")
var x: SIMD.Float32 = 1.2□
```

□ Changed

- Reverted the feature from 2023-02-13 that allowed unqualified struct members. Use the `self` keyword to conveniently access struct members with bound parameters instead. This was required to fix [Issue #260](#).
- Updated the RayTracing notebook: added step 5 to create specular lighting for more realistic images and step 6 to add a background image.

□ Fixed

- [Issue #260](#) - Definitions inside structs no longer shadow definitions outside of struct definitions.

2023-07-12

□ New

- Mojo now has support for global variables! This enables `var` and `let` declaration at the top-level scope in Mojo files. Global variable initializers are run when code modules are loaded by the platform according to the order of dependencies between global variables, and their destructors are called in the reverse order.
- The [Mojo programming manual](#) is now written as a Jupyter notebook, and available in its entirety in the Mojo Playground (`programming-manual.ipynb`). (Previously, `HelloMojo.ipynb` included most of the same material, but it was not up-to-date.)
- As a result, we've also re-written `HelloMojo.ipynb` to be much shorter and provide a more gentle first-user experience.
- [Coroutine module documentation](#) is now available. Coroutines form the basis of Mojo's support for asynchronous execution. Calls to `async fns` can be stored into a `Coroutine`, from which they can be resumed, awaited upon, and have their results retrieved upon completion.

□ Changed

- `simd_bit_width` in the `TargetInfo` module has been renamed to `simdbitwidth` to better align with `simdwidhtof`, `bitwidhtof`, etc.

□ Fixed

- The walrus operator now works in if/while statements without parentheses, e.g. `if x := function():`
- [Issue #428](#) - The `FloatLiteral` and `SIMD` types now support conversion to `Int` via the `to_int` or `__int__` method calls. The behavior matches that of Python, which rounds towards zero.

2023-07-05

□ New

- Tuple expressions now work without parentheses. For example, `a, b = b, a` works as you'd expect in Python.
- Chained assignments (e.g. `a = b = 42`) and the walrus operator (e.g. `some_function(b := 17)`) are now supported.

□ Changed

- The `simd_width` and `dtype_simd_width` functions in the `TargetInfo` module have been renamed to `simdwidhtof`.
- The `dtype_` prefix has been dropped from `alignof`, `sizeof`, and `bitwidhtof`. You can now use these functions (e.g. `alignof`) with any argument type, including `DType`.
- The `inf`, `neginf`, `nan`, `isinf`, `isfinite`, and `isnan` functions were moved from the `Numerics` module to the `Math` module, to better align with Python's library structure.

□ Fixed

- [Issue #253](#) - Issue when accessing a struct member alias without providing parameters.
- [Issue #404](#) - The docs now use snake_case for variable names, which more closely conforms to Python's style.
- [Issue #379](#) - Tuple limitations have been addressed and multiple return values are now supported, even without parentheses.
- [Issue #347](#) - Tuples no longer require parentheses.
- [Issue #320](#) - Python objects are now traversable via for loops.

June 2023

2023-06-29

□ New

- You can now share .ipynb notebook files in Mojo Playground. Just save a file in the shared directory, and then right-click the file and select **Copy Sharable link**. To open a shared notebook, you must already have [access to Mojo Playground](#); when you open a shared notebook, click **Import** at the top of the notebook to save your own copy. For more details about this feature, see the instructions inside the help directory, in the Mojo Playground file browser.

□ Changed

- The unroll2() and unroll3() functions in the [Functional](#) module have been renamed to overload the unroll() function. These functions unroll 2D and 3D loops and unroll() can determine the intent based on the number of input parameters.

□ Fixed

- [Issue #229](#) - Issue when throwing an exception from __init__ before all fields are initialized.
- [Issue #74](#) - Struct definition with recursive reference crashes.
- [Issue #285](#) - The [TargetInfo](#) module now includes is_little_endian() and is_big_endian() to check if the target host uses either little or big endian.
- [Issue #254](#) - Parameter name shadowing in nested scopes is now handled correctly.

2023-06-21

□ New

- Added support for overloading on parameter signature. For example, it is now possible to write the following:

```
fn foo[a: Int](x: Int):
    pass

fn foo[a: Int, b: Int](x: Int):
    pass
```

For details on the overload resolution logic, see the [programming manual](#).

- A new cost_of() function has been added to Autotune. This meta-function must be invoked at compile time, and it returns the number of MLIR operations in a function (at a certain stage in compilation), which can be used to build basic heuristics in higher-order generators.

```
from autotune import cost_of

fn generator[f: fn(Int) -> Int]() -> Int:
    @parameter
    if cost_of[fn(Int) -> Int, f]() < 10:
        return f()
    else:
        # Do something else for slower functions...
```

- Added a new example notebook with a basic Ray Tracing algorithm.

□ Changed

- The constrained_msg() in the Assert module has been renamed to constrained().

□ Fixed

- Overloads marked with @adaptive now correctly handle signatures that differ only in declared parameter names, e.g. the following now works correctly:

```
@adaptive
fn foobar[w: Int, T: DType]() -> SIMD[T, w]: ...

@adaptive
fn foobar[w: Int, S: DType]() -> SIMD[S, w]: ...
```

- [Issue #219](#) - Issue when redefining a function and a struct defined in the same cell.

- [Issue #355](#) - The loop order in the Matmul notebook for Python and naive mojo have been reordered for consistency. The loop order now follows (M, K, N) ordering.

- [Issue #309](#) - Use snake case naming within the testing package and move the asserts out of the TestSuite struct.

2023-06-14

□ New

- Tuple type syntax is now supported, e.g. the following works:

```
fn return_tuple() -> (Int, Int):
    return (1, 2)___
```

□ Changed

- The TupleLiteral type was renamed to just Tuple, e.g. Tuple[Int, Float].

□ Fixed

- [Issue #354](#) - Returning a tuple doesn't work even with parens.
- [Issue #365](#) - Copy-paste error in FloatLiteral docs.
- [Issue #357](#) - Crash when missing input parameter to variadic parameter struct member function.

2023-06-07

□ New

- Tuple syntax now works on the left-hand side of assignments (in "lvalue" positions), enabling things like `(a, b) = (b, a)`. There are several caveats: the element types must exactly match (no implicit conversions), this only works with values of TupleLiteral type (notably, it will not work with PythonObject yet) and parentheses are required for tuple syntax.

□ Removed

- Mojo Playground no longer includes the following Python packages (due to size, compute costs, and [environment complications](#)): torch, tensorflow, keras, transformers.

□ Changed

- The data types and scalar names now conform to the naming convention used by numpy. So we use Int32 instead of S132, similarly using Float32 instead of F32. Closes [Issue #152](#).

□ Fixed

- [Issue #287](#) - computed lvalues don't handle raising functions correctly
- [Issue #318](#) - Large integers are not being printed correctly
- [Issue #326](#) - Float modulo operator is not working as expected
- [Issue #282](#) - Default arguments are not working as expected
- [Issue #271](#) - Confusing error message when converting between function types with different result semantics

May 2023

2023-05-31

□ New

- Mojo Playground now includes the following Python packages (in response to [popular demand](#)): torch, tensorflow, polars, opencv-python, keras, Pillow, plotly, seaborn, sympy, transformers.
- A new optimization is applied to non-trivial copyable values that are passed as an owned value without using the transfer (^) operator. Consider code like this:

```
var someValue : T = ...
...
takeValueAsOwned(someValue)
...___
```

When `takeValueAsOwned()` takes its argument as an [owned](#) value (this is common in initializers for example), it is allowed to do whatever it wants with the value and destroy it when it is finished. In order to support this, the Mojo compiler is forced to make a temporary copy of the `someValue` value, and pass that value instead of `someValue`, because there may be other uses of `someValue` after the call.

The Mojo compiler is now smart enough to detect when there are no uses of `someValue` later, and it will elide the copy just as if you had manually specified the transfer operator like `takeValueAsOwned(someValue^)`. This provides a nice "it just works" behavior for non-trivial types without requiring manual management of transfers.

If you'd like to take full control and expose full ownership for your type, just don't make it copyable. Move-only types require the explicit transfer operator so you can see in your code where all ownership transfer happen.

- Similarly, the Mojo compiler now transforms calls to `__copyinit__` methods into calls to `__moveinit__` when that is the last use of the source value along a control flow path. This allows types which are both copyable and movable to get transparent move optimization. For example, the following code is compiled into moves instead of copies even without the use of the transfer operator:

```

var someValue = somethingCopyableAndMovable()
use(someValue)
...
let otherValue = someValue      # Last use of someValue
use(otherValue)
...
var yetAnother = otherValue    # Last use of otherValue
mutate(yetAnother)_

```

This is a significant performance optimization for things like `PythonObject` (and more complex value semantic types) that are commonly used in a fluid programming style. These don't want extraneous reference counting operations performed by its copy constructor.

If you want explicit control over copying, it is recommended to use a non-dunder `.copy()` method instead of `__copyinit__`, and recall that non-copyable types must always use the transfer operator for those that want fully explicit behavior.

Fixed

- [Issue #231](#) - Unexpected error when a Python expression raises an exception
- [Issue #119](#) - The REPL fails when a python variable is redefined

2023-05-24

New

- finally clauses are now supported on try statements. In addition, try statements no longer require `except` clauses, allowing try-finally blocks. finally clauses contain code that is always executed from control-flow leaves any of the other clauses of a try statement by any means.

Changed

- with statement emission changed to use the new finally logic so that

```
with ContextMgr():
    return_
```

Will correctly execute `ContextMgr.__exit__` before returning.

Fixed

- [Issue #204](#) - Mojo REPL crash when returning a String at compile-time
- [Issue #143](#) - synthesized init in `@register_passable` type doesn't get correct convention.
- [Issue #201](#) - String literal concatenation is too eager.
- [Issue #209](#) - [QoI] Terrible error message trying to convert a type to itself.
- [Issue #32](#) - Include struct fields in docgen
- [Issue #50](#) - Int to string conversion crashes due to buffer overflow
- [Issue #132](#) - `PythonObject to_int` method has a misleading name
- [Issue #189](#) - `PythonObject bool` conversion is incorrect
- [Issue #65](#) - Add SIMD constructor from `Bool`
- [Issue #153](#) - Meaning of `Time.now` function result is unclear
- [Issue #165](#) - Type in `Pointer.free` documentation
- [Issue #210](#) - Parameter results cannot be declared outside top-level in function
- [Issue #214](#) - Pointer offset calculations at compile-time are incorrect
- [Issue #115](#) - Float printing does not include the right number of digits
- [Issue #202](#) - `kgen.unreachable` inside nested functions is illegal
- [Issue #235](#) - Crash when register passable struct field is not register passable
- [Issue #237](#) - Parameter closure sharp edges are not documented

2023-05-16

New

- Added missing dunder methods to `PythonObject`, enabling the use of common arithmetic and logical operators on imported Python values.
- `PythonObject` is now [printable from Mojo](#), instead of requiring you to import Python's print function.

Fixed

- [Issue #98](#): Incorrect error with lifetime tracking in loop.
- [Issue #49](#): Type inference issue (?) in 'ternary assignment' operation (`FloatLiteral` vs. '`SIMD[f32, 1]`').
- [Issue #48](#): and/or don't work with memory-only types.
- [Issue #11](#): `setitem` Support for `PythonObject`.

2023-05-11

New

- `NDBuffer` and `Buffer` are now constructable via `Pointer` and `DTypewriter`.
- `String` now supports indexing with either integers or slices.

- Added factorial function to the Math module.

□ Changed

- The “byref” syntax with the & sigil has changed to use an `inout` keyword to be more similar to the borrowed and owned syntax in arguments. Please see [Issue #7](#) for more information.
- Optimized the Matrix multiplication implementation in the notebook. Initially we were optimizing for expandability rather than performance. We have found a way to get the best of both worlds and now the performance of the optimized Matmul implementation is 3x faster.
- Renamed the [^postfix operator](#) from “consume” to “transfer.”

□ Fixed

- Fixed missing overloads for `Testing.assertEqual` so that they work on `Integer` and `String` values.
- [Issue #6](#): Playground stops evaluating cells when a simple generic is defined.
- [Issue #18](#): Memory leak in Python interoperability was removed.

2023-05-02

□ Released

- Mojo publicly launched! This was epic, with lots of great coverage online including a [wonderful post by Jeremy Howard](#). The team is busy this week.

□ New

- Added a Base64 encoding function to perform base64 encoding on strings.

□ Changed

- Decreased memory usage of serialization of integers to strings.
- Speedup the sort function.

□ Fixed

- Fixed time unit in the `sleep` function.

April 2023

Week of 2023-04-24

- □ The default behavior of nested functions has been changed. Mojo nested functions that capture are by default are non-parametric, runtime closures, meaning that:

```
def foo(x):
    # This:
    def bar(y): return x*y
    # Is the same as:
    let bar = lambda y: x*y
```

These closures cannot have input or result parameters, because they are always materialized as runtime values. Values captured in the closure (x in the above example), are captured by copy: values with copy constructors cannot be copied and captures are immutable in the closure.

Nested functions that don’t capture anything are by default “parametric” closures: they can have parameters and they can be used as parameter values. To restore the previous behavior for capturing closures, “parametric, capture-by-unsafe-reference closures”, tag the nested function with the `@parameter` decorator.

- □ Mojo now has full support for “runtime” closures: nested functions that capture state materialized as runtime values. This includes taking the address of functions, indirect calls, and passing closures around through function arguments. Note that capture-by-reference is still unsafe!

You can also take references to member functions with instances of that class using `foo.member_function`, which creates a closure with `foo` bound to the `self` argument.

- □ Mojo now supports Python style `with` statements and context managers.

These things are very helpful for implementing things like our trace region support and things like Runtime support.

A context manager in Mojo implements three methods:

```
fn __enter__(self) -> T:
fn __exit__(self):
fn __exit__(self, err: Error) -> Bool:
```

The first is invoked when the context is entered, and returns a value that may optionally be bound to a target for use in the `with` body. If the `with` block exits normally, the second method is invoked to clean it up. If an error is raised, the third method is invoked with the `Error` value. If that method returns true, the error is considered handled, if it returns false, the error is re-thrown so propagation continues out of the ‘`with`’ block.

- Mojo functions now support variable scopes! Explicit `var` and `let` declarations inside functions can shadow declarations from higher “scopes”, where a scope is defined as any new indentation block. In addition, the `for` loop iteration variable is now scoped to the loop body, so it is finally possible to write

```
for i in range(1): pass
for i in range(2): pass
```

- Mojo now supports an `@value` decorator on structs to reduce boilerplate and encourage best practices in value semantics. The `@value` decorator looks to see the struct has a memberwise initializer (which has arguments for each field of the struct), a `__copyinit__` method, and a `__moveinit__` method, and synthesizes the missing ones if possible. For example, if you write:

```
@value
struct MyPet:
    var name: String
    var age: Int
```

The `@value` decorator will synthesize the following members for you:

```
fn __init__(inout self, owned name: String, age: Int):
    self.name = name^
    self.age = age
fn __copyinit__(inout self, existing: Self):
    self.name = existing.name
    self.age = existing.age
fn __moveinit__(inout self, owned existing: Self):
    self.name = existing.name^
    self.age = existing.age
```

This decorator can greatly reduce the boilerplate needed to define common aggregates, and gives you best practices in ownership management automatically. The `@value` decorator can be used with types that need custom copy constructors (your definition wins). We can explore having the decorator take arguments to further customize its behavior in the future.

- `Memcpy` and `memcmp` now consistently use `count` as the byte count.
- Add a variadic string join on strings.
- Introduce a `reduce_bit_count` method to count the number of 1 across all elements in a SIMD vector.
- Optimize the `pow` function if the exponent is integral.
- Add a `len` function which dispatches to `__len__` across the different structs that support it.

Week of 2023-04-17

- Error messages have been significantly improved, thanks to prettier printing for Mojo types in diagnostics.
- Variadic values can now be indexed directly without wrapping them in a `VariadicList`!
- `let` declarations in a function can now be lazily initialized, and `var` declarations that are never mutated get a warning suggesting they be converted to a `let` declaration. Lazy initialization allows more flexible patterns of initialization than requiring the initializer be inline, e.g.:

```
let x : Int
if cond:
    x = foo()
else:
    x = bar()
use(x)
```

- Functions defined with `def` now return `object` by default, instead of `None`. This means you can return values (convertible to `object`) inside `def` functions without specifying a return type.
- The `@raises` decorator has been removed. Raising `fn` should be declared by specifying `raises` after the function argument list. The rationale is that `raises` is part of the type system, instead of a function modifier.
- The `BoolLiteral` type has been removed. Mojo now emits `True` and `False` directly as `Bool`.
- Syntax for function types has been added. You can now write function types with `fn(Int) -> String` or `async def(&String, *Int) -> None`. No more writing `!kgen.signature` types by hand!
- Float literals are not emitted as `FloatLiteral` instead of an MLIR `f64` type!
- Automatic destructors are now supported by Mojo types, currently spelled `fn __del__(owned self):` (the extra underscore will be dropped shortly). These destructors work like Python object destructors and similar to C++ destructors, with the major difference being that they run “as soon as possible” after the last use of a value. This means they are not suitable for use in C++-style RAII patterns (use the `with` statement for that, which is currently unsupported).

These should be generally reliable for both memory-only and register-passable types, with the caveat that closures are known to *not* capture values correctly. Be very careful with interesting types in the vicinity of a closure!

- A new (extremely dangerous!) builtin function is available for low-level ownership muckery. The `__get_address_as_owned_value(x)` builtin takes a low-level address value (of `!kgen.pointer` type) and returns an `owned` value for the memory that is pointed to. This value is assumed live at the invocation of the builtin, but is “owned” so it needs to be consumed by the caller, otherwise it will be automatically destroyed. This is an effective way to do a “placement delete” on a pointer.

```
# "Placement delete": destroy the initialized object begin pointed to.
_ = __get_address_as_owned_value(somePointer.value)

# Result value can be consumed by anything that takes it as an 'owned'
# argument as well.
consume(__get_address_as_owned_value(somePointer.value))
```

- Another magic operator, named `__get_address_as_uninit_lvalue(x)` joins the magic LValue operator family. This operator projects a pointer to an LValue like `__get_address_as_lvalue(x)`. The difference is that `__get_address_as_uninit_lvalue(x)` tells the compiler that the pointee is uninitialized on entry and initialized on exit, which means that you can use it as a “placement new” in C++ sense. `__get_address_as_lvalue(x)` tells the compiler that the pointee is initialized already, so reassigning over it will run the destructor.

```
# /*Re*placement new": destroy the existing SomeHeavy value in the memory,
# then initialize a new value into the slot.
__get_address_as_lvalue(somePointer.value) = SomeHeavy(4, 5)

# Ok to use an lvalue, convert to borrow etc.
use(__get_address_as_lvalue(somePointer.value))

# "Placement new": Initialize a new value into uninitialized memory.
__get_address_as_uninit_lvalue(somePointer.value) = SomeHeavy(4, 5)

# Error, cannot read from uninitialized memory.
use(__get_address_as_uninit_lvalue(somePointer.value))
```

Note that `__get_address_as_lvalue` assumes that there is already a value at the specified address, so the assignment above will run the `SomeHeavy` destructor (if any) before reassigning over the value.

- Implement full support for `__moveinit_` (aka move constructors)

This implements the ability for memory-only types to define two different types of move ctors if they'd like:

1. fn `__moveinit_(inout self, owned existing: Self)`: Traditional Rust style moving constructors that shuffles data around while taking ownership of the source binding.
2. fn `__moveinit_(inout self, inout existing: Self)`: C++ style “stealing” move constructors that can be used to take from an arbitrary LValue.

This gives us great expressive capability (better than Rust/C++/Swift) and composes naturally into our lifetime tracking and value categorization system.

- The `__call_` method of a callable type has been relaxed to take `self` by borrow, allow non-copyable callees to be called.
- Implicit conversions are now invoked in `raise` statements properly, allowing converting strings to `Error` type.
- Automatic destructors are turned on for `__del_` instead of `__del__`.
- Add the builtin `FloatLiteral` type.
- Add integral `floordiv` and `mod` for the SIMD type that handle negative values.
- Add an `F64` to `String` converter.
- Make the `print` function take variadic inputs.

Week of 2023-04-10

- Introduce consume operator `x^`

This introduces the postfix consume operator, which produces an RValue given a lifetime tracked object (and, someday, a movable LValue).

- Mojo now automatically synthesizes empty destructor methods for certain types when needed.
- The `object` type has been built out into a fully-dynamic type, with dynamic function dispatch, with full error handling support.

```
def foo(a) -> object:
    return (a + 3.45) < [1, 2, 3] # raises a TypeError
```

- The `@always_inline` decorator is no longer required for passing capturing closures as parameters, for both the functions themselves as functions with capturing closures in their parameters. These functions are still inlined but it is an implementation detail of capturing parameter closures. Mojo now distinguishes between capturing and non-capturing closures. Nested functions are capturing by default and can be made non-capturing with the `@noncapturing` decorator. A top-level function can be passed as a capturing closure by marking it with the `@closure` decorator.

- Support for list literals has been added. List literals `[1, 2, 3]` generate a variadic heterogeneous list type.
- Variadics have been extended to work with memory-primary types.
- Slice syntax is now fully-supported with a new builtin `slice` object, added to the compiler builtins. Slice indexing with `a[1:2:3]` now emits calls to `__getitem_` and `__setitem_` with a slice object.
- Call syntax has been wired up to `__call_`. You can now `f()` on custom types!
- Closures are now explicitly typed as capturing or non-capturing. If a function intends to accept a capturing closure, it must specify the capturing function effect.
- Add a `Tile2D` function to enable generic 2D tiling optimizations.
- Add the `slice` struct to enable getting/setting spans of elements via `getitem/setitem`.
- Add syntax sugar to autotuning for both specifying the autotuned values, searching, and declaring the evaluation function.

Week of 2023-04-03

- The `AnyType` and `NoneType` aliases were added and auto-imported in all files.

- ☐ The Mojo VS Code extension has been improved with docstring validation. It will now warn when a function's docstring has a wrong argument name, for example.
 - ☐ A new built-in literal type `TupleLiteral` was added in `_CompilerBuiltins`. It represents literal tuple values such as `(1, 2.0)` or `()`.
 - ☐ The `Int` type has been moved to a new `Builtin` module and is auto-imported in all code. The type of integer literals has been changed from the MLIR index type to the `Int` type.
 - Mojo now has a powerful flow-sensitive uninitialized variable checker. This means that you need to initialize values before using them, even if you overwrite all subcomponents. This enables the compiler to reason about the true lifetime of values, which is an important stepping stone to getting automatic value destruction in place.
 - ☐ Call syntax support has been added. Now you can directly call an object that implements the `__call__` method, like `foo(5)`.
 - ☐ The name for copy constructors got renamed from `__copy__` to `__copyinit__`. Furthermore, non-`@register_passable` types now implement it like they do an init method where you fill in a by-reference self, for example:
- ```
fn __copyinit__(inout self, existing: Self):
 self.first = existing.first
 self.second = existing.second
```
- This makes copy construction work more similarly to initialization, and still keeps copies `x = y` distinct from initialization `x = T(y)`.
- ☐ Initializers for memory-primary types are now required to be in the form `__init__(inout self, ...)`: with a `None` result type, but for register primary types, it remains in the form `__init__(...) -> Self`. The `T{}` initializer syntax has been removed for memory-primary types.
  - Mojo String literals now emit a builtin `StringLiteral` type! One less MLIR type to worry about.
  - New `__getattr__` and `__setattr__` dunder methods were added. Mojo calls these methods on a type when attempting member lookup of a non-static member. This allows writing dynamic objects like `x.foo()` where `foo` is not a member of `x`.
  - Early destructor support has been added. Types can now define a special destructor method `__del__` (note three underscores). This is an early feature and it is still being built out. There are many caveats, bugs, and missing pieces. Stay tuned!
  - ☐ Integer division and mod have been corrected for rounding in the presence of negative numbers.
  - ☐ Add scalar types (`UI8`, `SI32`, `F32`, `F64`, etc.) which are aliases to `SIMD[1, type]`.

## March 2023

### Week of 2023-03-27

- ☐ Parameter names are no longer load-bearing in function signatures. This gives more flexibility in defining higher-order functions, because the functions passed as parameters do not need their parameter names to match.

```
Define a higher-order function...
fn generator[
 func: __mlir_type['!kgen.signature<', Int, '>() -> !kgen.none`]
]():
 pass

Int parameter is named "foo".
fn f0[foo: Int]() {
 pass

Int parameter is named "bar".
fn f1[bar: Int]() {
 pass

fn main():
 # Both can be used as `func`!
 generator[f0]()
 generator[f1()]
```

Stay tuned for improved function type syntax...

- ☐ Two magic operators, named `__get_lvalue_as_address(x)` and `__get_address_as_lvalue` convert stored LValues to and from `!kgen.pointer` types (respectively). This is most useful when using the `Pointer[T]` library type. The `Pointer.address_of(lvalue)` method uses the first one internally. The second one must currently be used explicitly, and can be used to project a pointer to a reference that you can pass around and use as a self value, for example:

```
"Replacement new" SomeHeavy value into the memory pointed to by a
Pointer[SomeHeavy].
__get_address_as_lvalue(somePointer.value) = SomeHeavy(4, 5)
```

Note that `__get_address_as_lvalue` assumes that there is already a value at the specified address, so the assignment above will run the `SomeHeavy` destructor (if any) before reassigning over the value.

- The `((x))` syntax is `__mlir_op` has been removed in favor of `__get_lvalue_as_address` which solves the same problem and is more general.
- ☐ When using a mutable `self` argument to a struct `__init__` method, it now must be declared with `&`, like any other mutable method. This clarifies the mutation model by making `__init__` consistent with other mutating methods.
- ☐ Add variadic string join function.
- ☐ Default initialize values with 0 or null if possible.
- ☐ Add compressed, aligned, and mask store intrinsics.

## Week of 2023-03-20

- Initial String type is added to the standard library with some very basic methods.
  - Add DimList to remove the need to use an MLIR list type throughout the standard library.
  - The \_\_clone\_\_ method for copying a value is now named \_\_copy\_\_ to better follow Python term of art.
  - The \_\_copy\_\_ method now takes its self argument as a “borrowed” value, instead of taking it by reference. This makes it easier to write, works for @register\_passable types, and exposes more optimization opportunities to the early optimizer and dataflow analysis passes.
- # Before:  
fn \_\_clone\_\_(inout self) -> Self: ...
- # After:  
fn \_\_copy\_\_(self) -> Self: ...
- A new @register\_passable("trivial") may be applied to structs that have no need for a custom \_\_copy\_\_ or \_\_del\_\_ method, and whose state is only made up of @register\_passable("trivial") types. This eliminates the need to define \_\_copy\_\_ boilerplate and reduces the amount of IR generated by the compiler for trivial types like Int.
  - You can now write back to attributes of structs that are produced by a computed lvalue expression. For example a[i].x = .. works when a[i] is produced with a \_\_getitem\_\_ / \_\_setitem\_\_ call. This is implemented by performing a read of a[i], updating the temporary, then doing a writeback.
  - The remaining hurdles to using non-parametric, @register\_passable types as parameter values have been cleared. Types like Int should enjoy full use as parameter values.
  - Parameter pack inference has been added to function calls. Calls to functions with parameter packs can now elide the pack types:

```
fn foo[*Ts: AnyType](args: *Ts): pass
foo(1, 1.2, True, "hello")
```

Note that the syntax for parameter packs has been changed as well.

- Add the runtime string type.
- Introduce the DimList struct to remove the need to use low-level MLIR operations.

## Week of 2023-03-13

- Initializers for structs now use \_\_init\_\_ instead of \_\_new\_\_, following standard practice in Python. You can write them in one of two styles, either traditional where you mutate self:

```
fn __init__(self, x: Int):
 self.x = x
```

or as a function that returns an instance:

```
fn __init__(x: Int) -> Self:
 return Self {x: x}
```

Note that @register\_passable types must use the later style.

- The default argument convention is now the borrowed convention. A “borrowed” argument is passed like a C++ const& so it doesn’t need to invoke the copy constructor (aka the \_\_clone\_\_ method) when passing a value to the function. There are two differences from C++ const&:

1. A future borrow checker will make sure there are no mutable aliases with an immutable borrow.
2. @register\_passable values are passed directly in an SSA register (and thus, usually in a machine register) instead of using an extra reference wrapper. This is more efficient and is the ‘right default’ for @register\_passable values like integers and pointers.

This also paves the way to remove the reference requirement from \_\_clone\_\_ method arguments, which will allow us to fill in more support for them.

- Support for variadic pack arguments has been added to Mojo. You can now write heterogeneous variadic packs like:

```
fn foo[*Ts: AnyType](args*: Ts): pass
foo[Int, F32, String, Bool](1, 1.5, "hello", True)
```

- The owned argument convention has been added. This argument convention indicates that the function takes ownership of the argument and is responsible for managing its lifetime.
- The borrowed argument convention has been added. This convention signifies the callee gets an immutable shared reference to a value in the caller’s context.
- Add the getenv function to the os module to enable getting environment variables.
- Enable the use of dynamic strides in NDBuffer.

## Week of 2023-03-06

- Support added for using capturing async functions as parameters.
- Returning result parameters has been moved from return statements to a new param\_return statement. This allows returning result parameters from throwing functions:

```
@raises
fn foo() -> out: Int():
 param_return[42]
 raise Error()
```

And returning different parameters along @parameter if branches:

```
fn bar[in: Bool -> out: Int]()@parameter:
 if in:
 param_return[1]
 else:
 param_return[2]
```

- Mojo now supports omitting returns at the end of functions when they would not be reachable. For instance,

```
fn foo(cond: Bool) -> Int:
 if cond:
 return 0
 else:
 return 1

fn bar() -> Int:
 while True:
 pass
```

- String literals now support concatenation, so "hello " "world" is treated the same as "hello world".
- Empty bodies on functions, structs, and control flow statements are no longer allowed. Please use `pass` in them to explicitly mark that they are empty, just like in Python.
- Structs in Mojo now default to living in memory instead of being passed around in registers. This is the right default for generality (large structures, structures whose pointer identity matters, etc) and is a key technology that enables the borrow model. For simple types like `Int` and `SIMD`, they can be marked as `@register_passable`.

Note that memory-only types currently have some limitations: they cannot be used in generic algorithms that take and return a `!mlir_type` argument, and they cannot be used in parameter expressions. Because of this, a lot of types have to be marked `@register_passable` just to work around the limitations. We expect to enable these use-cases over time.

- Mojo now supports computed lvalues, which means you can finally assign to subscript expressions instead of having to call `__setitem__` explicitly.

Some details on this: Mojo allows you to define multiple `__setitem__` overloads, but will pick the one that matches your `__getitem__` type if present. It allows you to pass computed lvalues into inout arguments by introducing a temporary copy of the value in question.

- Mojo now has much better support for using register-primary struct types in parameter expressions and as the types of parameter values. This will allow migration of many standard library types away from using bare MLIR types like `_mlir_type.index` and towards using `Int`. This moves us towards getting rid of MLIR types everywhere and makes struct types first-class citizens in the parameter system.
- Add a `sort` function.
- Add non-temporal store to enable cache bypass.

## February 2023

### Week of 2023-02-27

- The `@interface`, `@implements`, and `@evaluator` trio of decorators have been removed, replaced by the `@parameter if` and `@adaptive` features.
- Parameter inference can now infer the type of variadic lists.
- Memory primary types are now supported in function results. A result slot is allocated in the caller, and the callee writes the result of the function into that slot. This is more efficient for large types that don't fit into registers neatly! And initializers for memory-primary types now initialize the value in-place, instead of emitting a copy!
- Support for `let` decls of memory primary types has been implemented. These are constant, ready-only values of memory primary types but which are allocated on the function stack.
- Overload conversion resolution and parameter inference has been improved:
  - Inference now works with `let` decls in some scenarios that weren't working before.
  - Parameter bindings can now infer types into parameter expressions. This helps resolve higher-order functions in parameter expressions.
- Optimize `floor`, `ceil`, and `ldexp` on X86 hardware.
- Implement the log math function.

### Week of 2023-02-20

- A new `__memory_primary` struct decorator has been introduced. Memory primary types must always have an address. For instance, they are always stack-allocated when declared in a function and their values are passed into function calls by address instead of copy. This is in contrast with register primary types that may not have an address, and which are passed by value in function calls. Memory-primary fields are not allowed inside register-primary structs, because struct elements are stored in-line.

- ☐ A new `_CompilerBuiltin` module was added. This module defines core types and functions of the language that are referenced by the parser, and hence, is auto-imported by all other modules. For example new types for literal values like the boolean `True`/`False` will be included in `_CompilerBuiltin`.
- ☐ A special `__adaptive_set` property can be accessed on a function reference marked as `@adaptive`. The property returns the adaptive overload set of that function. The return type is a `!kgen.variadic`. This feature is useful to implement a generic `evaluate` function in the standard library.
- ☐ A new built-in literal type `BoolLiteral` was added in `_CompilerBuiltin`. It represents the literal boolean values `True` and `False`. This is the first Mojo literal to be emitted as a standard library type!
- ☐ Add the prefetch intrinsic to enable HW prefetching a cache line.
- ☐ Add the `InlinedFixedVector`, which is optimized for small vectors and stores values on both the stack and the heap.

## Week of 2023-02-13

- Unqualified lookups of struct members apply contextual parameters. This means for instance that you can refer to static methods without binding the struct parameters.

```
struct Foo[x: Int]:
 @staticmethod
 bar(): pass

 foo(self):
 bar() # implicitly binds to Foo[x].bar()
 Foo[2].bar() # explicitly bind to another parameter
```

- ☐ A new `Self` type refers to the enclosing type with all parameters bound to their current values. This is useful when working with complex parametric types, e.g.:

```
struct MyArray[size: Int, element_type: type]:
 fn __new__() -> Self:
 return Self {...}
```

which is a lot nicer than having to say `MyArray[size, element_type]` over and over again.

- ☐ Mojo now supports an `@adaptive` decorator. This decorator will supersede interfaces, and it represents an overloaded function that is allowed to resolve to multiple valid candidates. In that case, the call is emitted as a fork, resulting in multiple function candidates to search over.

```
@adaptive
fn sort(arr: ArraySlice[Int]):
 bubble_sort(arr)

@adaptive
fn sort(arr: ArraySlice[Int]):
 merge_sort(arr)

fn concat_and_sort(lhs: ArraySlice[Int], rhs: ArraySlice[Int]):
 let arr = lhs + rhs
 sort(arr) # this forks compilation, creating two instances
 # of the surrounding function
```

- ☐ Mojo now requires that types implement the `__clone__` special member in order to copy them. This allows the safe definition of non-copyable types like `Atomic`. Note that Mojo still doesn't implement destructors, and (due to the absence of non-mutable references) it doesn't actually invoke the `__clone__` member when copying a `let` value. As such, this forces to you as a Mojo user to write maximal boilerplate without getting much value out of it.

In the future, we will reduce the boilerplate with decorators, and we will actually start using it. This will take some time to build out though.

- ☐ A special `__mlir_region` statement was added to provide stronger invariants around defining MLIR operation regions in Mojo. It has similar syntax to function declarations, except it there are no results and no input conventions.
- ☐ Implement the log math function.
- ☐ Improve the `DType` struct to enable compile-time equality checks.
- ☐ Add the `Complex` struct class.

## Week of 2023-02-06

- ☐ The `if` statement now supports a `@parameter` decorator, which requires its condition to be a parameter expression, but which only emits the 'True' side of the condition to the binary, providing a "static if" functionality. This should eliminate many uses of `@interface` that are just used to provide different constraint on the implementations.
- ☐ `fn main()`: is now automatically exported and directly runnable by the command-line `mojo` tool. This is a stop-gap solution to enable script-like use cases until we have more of the language built out.
- ☐ The `@nodebug_inline` feature has been removed, please use `@alwaysinline("nodebug")` for methods that must be inlined and that we don't want to step into.
- ☐ Python chained comparisons, ex. `a < b < c`, are now supported in Mojo.
- ☐ Functions can now be defined with default argument values, such as `def f(x: Int, y: Int = 5):`. The default argument value is used when callers do not provide a value for that argument: `f(3)`, for example, uses the default argument value of `y = 5`.
- Unused coroutine results are now nicely diagnosed as "missing await" warnings.

- ☐ Introduce a vectorized reduction operations to the SIMD type.

## January 2023

### Week of 2023-01-30

- A basic Mojo language server has been added to the VS Code extension, which parses your code as you write it, and provides warnings, errors, and fix-it suggestions!
- ☐ The Mojo standard library is now implicitly imported by default.
- The coroutine lowering support was reworked and a new `Coroutine[T]` type was implemented. Now, the result of a call to an `async` function MUST be wrapped in a `Coroutine[T]`, or else memory will leak. In the future, when Mojo supports destructors and library types as literal types, the results of `async` function calls will automatically wrapped in a `Coroutine[T]`. But today, it must be done manually. This type implements all the expected hooks, such as `_await_`, and `get()` to retrieve the result. Typical usage:

```
async fn add_three(a: Int, b: Int, c: Int) -> Int:
 return a + b + c

async fn call_it():
 let task: Coroutine[Int] = add_three(1, 2, 3)
 print(await task)
```

- ☐ We now diagnose unused expression values at statement context in `fn` declarations (but not in `defs`). This catches bugs with unused values, e.g. when you forget the parens to call a function.
- ☐ An `@always_inline("nodebug")` function decorator can be used on functions that need to be force inlined, but when they should not have debug info in the result. This should be used on methods like `Int.__add__` which should be treated as builtin.
- ☐ The `@export` decorator now supports an explicit symbol name to export to, for example:

```
@export("baz") # exported as 'baz'
fn some_mojo_fn_name():
```

- ☐ Subscript syntax is now wired up to the `__getitem__` dunder method.

This allows type authors to implement the `__getitem__` method to enable values to be subscripted. This is an extended version of the Python semantics (given we support overloading) that allows you to define N indices instead of a single version that takes a tuple (also convenient because we don't have tuples yet).

Note that this has a very, very important limitation: subscripts are NOT wired up to `__setitem__` yet. This means that you can read values with `.. = v[i]` but you cannot store to them with `v[i] = ...` For this, please continue to call `__setitem__` directly.

- ☐ Function calls support parameter inference.

For calls to functions that have an insufficient number of parameters specified at the callsite, we can now infer them from the argument list. We do this by matching up the parallel type structure to infer what the parameters must be.

Note that this works left to right in the parameter list, applying explicitly specified parameters before trying to infer new ones. This is similar to how C++ does things, which means that you may want to reorder the list of parameters with this in mind. For example, a `dyn_cast`-like function will be more elegant when implemented as:

```
fn dyn_cast[DstType: type, SrcType: type](src: SrcType) -> DstType:
```

Than with the `SrcType/DstType` parameters flipped around.

- ☐ Add the growable Dynamic vector struct.

### Week of 2023-01-23

- Inplace operations like `+=/__iadd__` may now take `self` by-val if they want to, instead of requiring it to be by-ref.
- ☐ Inplace operations are no longer allowed to return a non-None value. The corresponding syntax is a statement, not an expression.
- A new `TaskGroup` type was added to the standard library. This type can be used to schedule multiple tasks on a multi-threaded workqueue to be executed in parallel. An `async` function can await all the tasks at once with the `taskgroup`.
- ☐ We now support for loops! A type that defines an `__iter__` method that returns a type that defines `__next__` and `__len__` methods is eligible to be used in the statement `for el in X()`. Control flow exits the loop when the length is zero.

This means things like this now work:

```
for item in range(start, end, step):
 print(item)
```

- Result parameters now have names. This is useful for referring to result parameters in the return types of a function:

```
fn return_simd() -> nelts: Int() -> SIMD[f32, nelts]:
```

- ☐ We now support homogeneous variadics in value argument lists, using the standard Python `fn thing(*args: Int):` syntax! Variadics also have support in parameter lists:

```
fn variadic_params_and_args[*a: Int](*b: Int):
 print(a[0])
 print(b[1])
```

- ☐ Add the range struct to enable for ... range(...) loops.

- ☐ Introduce the unroll generator to allow one to unroll loops via a library function.

## Week of 2023-01-16

- ☐ Struct field references are now supported in parameter context, so you can use `someInt.value` to get the underlying MLIR thing out of it. This should allow using first-class types in parameters more widely.
- ☐ We now support “pretty” initialization syntax for structs, e.g.:

```
struct Int:
 var value: __mlir_type.index
 fn __new__(value: __mlir_type.index) -> Int:
 return Int {value: value}()
```

This eliminates the need to directly use the MLIR `lit.struct.create` op in struct initializers. This syntax may change in the future when ownership comes in, because we will be able to support the standard `__init__` model then.

- ☐ It is now possible to attach regions to `__mlir_op` operations. This is done with a hack that allows an optional `_region` attribute that lists references to the region bodies (max 1 region right now due to lack of list [] literal).
- Nested functions now parse, e.g.:

```
fn foo():
 fn bar():
 pass
 bar()
```

- Python-style `async` functions should now work and the `await` expression prefix is now supported. This provides the joy of `async/await` syntactic sugar when working with asynchronous functions. This is still somewhat dangerous to use because we don’t have proper memory ownership support yet.
- String literals are now supported.
- Return processing is now handled by a dataflow pass inside the compiler, so it is possible to return early out of if statements.
- The parser now supports generating ‘fixit’ hints on diagnostics, and uses them when a dictionary literal uses a colon instead of equal, e.g.:

```
x.mojo:8:48: error: expected ':' in subscript slice, not '='
 return __mlir_op.'lit.struct.create'[value = 42]()
 ^
:
```

- ☐ Add reduction methods which operate on buffers.
- ☐ Add more math functions like sigmoid, sqrt, rsqrt, etc.
- ☐ Add partial load / store which enable loads and stores that are predicated on a condition.

## Week of 2023-01-09

- The `/` and `*` markers in function signatures are now parsed and their invariants are checked. We do not yet support keyword arguments yet though, so they aren’t very useful.
- Functions now support a new `@nodebug_inline` decorator. (Historical note: this was later replaced with `@alwaysinline("nodebug")`).

Many of the things at the bottom level of the Mojo stack are trivial zero-abstraction wrappers around MLIR things, for example, the `+` operator on `Int` or the `__bool__` method on `Bool` itself. These operators need to be force inlined even at `-O0`, but they have some additional things that we need to wrestle with:

1. In no case would a user actually want to step into the `__bool__` method on `Bool` or the `+` method on `Int`. This would be terrible debugger QoL for unless you’re debugging `Int` itself. We need something like `__always_inline__`, `__nodebug__` attributes that clang uses in headers like `xmmmintrin.h`.
2. Similarly, these “operators” should be treated by users as primitives: they don’t want to know about MLIR or internal implementation details of `Int`.
3. These trivial zero abstraction things should be eliminated early in the compiler pipeline so they don’t slow down the compiler, bloating out the call graph with trivial leaves. Such thing slows down the elaborator, interferes with basic MLIR things like `fold()`, bloats out the IR, or bloats out generated debug info.
4. In a parameter context, we want some of these things to get inlined so they can be simplified by the attribute logic and play more nicely with canonical types. This is just a nice to have thing those of us who have to stare at generated IR.

The solution to this is a new `@nodebug_inline` decorator. This decorator causes the parser to force-inline the callee instead of generating a call to it. While doing so, it gives the operations the location of the call itself (that’s the “nodebug” part) and strips out let decls that were part of the internal implementation details.

This is a super-power-user-feature intended for those building the standard library itself, so it is intentionally limited in power and scope: It can only be used on small functions, it doesn’t support regions, by-ref, throws, async, etc.

- Separately, we now support an `@alwaysInline` decorator on functions. This is a general decorator that works on any function, and indicates that the function must be inlined. Unlike `@nodebug_inline`, this kind of inlining is performed later in the compilation pipeline.
- The `__include` hack has been removed now that we have proper import support.
- `__mlir_op` can now get address of l-value:

You can use magic (((x))) syntax in `_mlir_op` that forces the `x` expression to be an lvalue, and yields its address. This provides an escape hatch (isolated off in `_mlir_op` land) that allows unsafe access to lvalue addresses.

- We now support `_rlshift_` and `_rtruediv_`.
- □ The parser now resolves scoped alias references. This allows us to support things like `SomeType.someAlias`, forward substituting the value. This unblocks use of aliases in types like `DTyep`. We'd like to eventually preserve the reference in the AST, but this unblocks library development.
- □ Add a `now` function and `Benchmark` struct to enable timing and benchmarking.
- □ Move more of the computation in `NDBuffer` from runtime to compile time if possible (e.g. when the dimensions are known at compile time).

## Week of 2023-01-02

- □ Added the `print` function which works on Integers and SIMD values.
- The frontend now has a new diagnostic subsystem used by the `kgen` tool (but not by `kgen-translate` for tests) that supports source ranges on diagnostics. Before we'd emit an error like:

```
x.mojo:13:3: error: invalid call to 'callee': in argument #0, value of type '$F32::F32' cannot be converted to expected type '$int::Int'
 callee(1.0+F32(2.0))
 ^
x.lit:4:1: note: function declared here
fn callee(a: Int):
^
```

now we produce:

```
x.mojo:13:3: error: invalid call to 'callee': in argument #0, value of type '$F32::F32' cannot be converted to expected type '$int::Int'
 callee(1.0+F32(2.0))
 ^
  ~~~~~
x.lit:4:1: note: function declared here
fn callee(a: Int):
^
```

- □ Parameter results are now supported in a proper way. They are now forward declared with an alias declaration and then bound in a call with an arrow, e.g.:

```
alias a : __mlir_type.index
alias b : __mlir_type.index
idx_result_params[xyz*2 -> a, b]()
```

- Various minor issues with implicit conversions are fixed. For instances, implicit conversions are now supported in parameter binding contexts and alias declarations with explicit types.
- Doc strings are allowed on functions and structs, but they are currently discarded by the parser.
- □ Add a `print` method!!!
- □ Demonstrate a naive matmul in Mojo.
- □ Initial work on functions that depend on types (e.g. `FPUtols`, `nan`, `inf`, etc.)
- □ Allow one to query hardware properties such as `simd_width`, `os`, etc. via `TargetInfo` at compile time.

## December 2022

### Week of 2022-12-26

- □ You can now call functions in a parameter context! Calling a function in a parameter context will evaluate the function at compile time. The result can then be used as parameter values. For example,

```
fn fma(x: Int, y: Int, z: Int) -> Int:
  return a + b * c

fn parameter_call():
  alias nelts = fma(32, 2, 16)
  var x: SIMD[f32, nelts]
```

- You can now disable printing of types in an `_mlir_attr` substitution by using unary + expression.
- □ `let` declarations are now supported in functions. `let` declarations are local run-time constant values, which are always rvalues. They complement 'var' decls (which are mutable lvalues) and are the normal thing to use in most cases. They also generate less IR and are always in SSA form when initialized.

We will want to extend this to support 'let' decls in structs at some point and support lazy initialized 'let' declarations (using dataflow analysis) but that isn't supported yet.

- □ Add the `NDBuffer` struct.
- Happy new year.

### Week of 2022-12-19

- □ Start of the Standard library:
  1. Added Integer and SIMD structs to bootstrap the standard library.
  2. Added very basic buffer data structure.

- We have basic support for parsing parameter results in function calls! Result parameters are an important Mojo metaprogramming feature. They allow functions to return compile-time constants.

```
fn get_preferred_simdwidthof() -> nelts: Int():
    return[2]
```

```
fn vectorized_function():
    get_preferred_simdwidthof() -> nelts]()
    var x: SIMD[f32, nelts] =
```

- Types can now be used as parameters of `!kgen.mlirtype` in many more cases.
- MLIR operations with zero results don't need to specify `_type: []` anymore.
- We support parsing triple quoted strings, for writing docstrings for your functions and structs!
- A new `_mlir_type[a,b,c]` syntax is available for substituting into MLIR types and attributes is available, and the old placeholder approach is removed. This approach has a few advantages beyond what placeholders do:

1. It's simpler.
2. It doesn't form the intermediate result with placeholders, which gets rejected by MLIR's semantic analysis, e.g. the complex case couldn't be expressed before.
3. It provides a simple way to break long attrs/types across multiple lines.

- We now support an `@evaluator` decorator on functions for KGEN evaluators. This enables specifying user-defined interface evaluators when performing search during compilation.
- `import` syntax is now supported!

This handles packaging imported modules into file ops, enables effective isolation from the other decls. "import" into the desired context is just aliasing decls, with the proper symbols references handle automatically during IR generation. As a starting point, this doesn't handle any notion of packages (as those haven't been sketched out enough).

- Reversed binary operators (like `_radd_`) are now looked up and used if the forward version (like `_add_`) doesn't work for some reason.
- Implicit conversions are now generally available, e.g. in assign statements, variable initializers etc. There are probably a few more places they should work, but we can start eliminating all the extraneous explicit casts from literals now.
- Happy Holidays

## Week of 2022-12-12

- Function overloading now works. Call resolution filters candidate list according to the actual parameter and value argument specified at the site of the call, diagnosing an error if none of the candidates are viable or if multiple are viable and ambiguous. We also consider implicit conversions in overload look:

```
fn foo(x: Int): pass
fn foo(x: F64): pass

foo(Int(1)) # resolves to the first overload
foo(1.0)    # resolves to the second overload
foo(1)      # error: both candidates viable with 1 implicit conversion! =
```

- The short circuiting binary `and` and `or` expressions are now supported.
- Unary operator processing is a lot more robust, now handling the `not` expression and `~x` on `Bool`.
- The compiler now generates debug information for use with GDB/LLDB that describes variables and functions.
- The first version of the Mojo Visual Studio Code extension has been released! It supports syntax highlighting for Mojo files.
- The first version of the `Bool` type has landed in the new Mojo standard library!
- Implicit conversions are now supported in return statements.

## Week of 2022-12-05

- "Discard" patterns are now supported, e.g. `_ = foo()`
- We now support implicit conversions in function call arguments, e.g. converting an `index` value to `Int` automatically. This eliminates a bunch of casts, e.g. the need to say `F32(1.0)` everywhere.

This is limited for a few reasons that will be improved later:

1. We don't support overloading, so lots of types aren't convertible from all the things they should be, e.g. you can't pass "1" to something that expects `F32`, because `F32` can't be created from `index`.
2. This doesn't "check to see if we can invoke `_new_`" it force applies it on a mismatch, which leads to poor QoL.
3. This doesn't fix things that need `radd`.

## November 2022

### Week of 2022-11-28

- We support the `True` and `False` keywords as expressions.
- A new `alias` declaration is supported which allows defining local parameter values. This will eventually subsume type aliases and other things as it gets built out.

- We now have end-to-end execution of Mojo files using the `kgen` tool! Functions exported with `@export` can be executed.
- We have `try-except-else` and `raise` statements and implicit error propagation! The error semantics are that `def` can raise by default, but `fn` must explicitly declare raising with a `@raises` decorator. Stub out basic `Error` type.
- The & sigil for by-ref arguments is now specified after the identifier. Postfix works better for ref and move operators on the expression side because it chains and mentally associates correctly: `thing.method().result^`. We don't do that yet, but align param decl syntax to it so that things won't be odd looking when we do. In practice this looks like:

```
def mutate_argument(a&: index):
    a = 25
```

## Week of 2022-11-21

- The magic `index` type is gone. Long live `_mlir_type.index`.
- Implement parameter substitution into parametric `_mlir_type` decls. This allows us to define parametric opaque MLIR types with exposed parameters using a new "placeholder" attribute. This allows us to expose the power of the KGEN type parametric system directly into Mojo.
- Fully-parametric custom types can now be defined and work in Mojo, bringing together a lot of the recent work. We can write the SIMD type directly as a wrapper around the KGEN type, for example:

```
struct SIMD[dt: _mlir_type.`!kgen.dtype`, nelts: _mlir_type.index]:
    var value:
        _mlir_type.`!pop.simd<#lit<placeholder index>, #lit<placeholder !kgen.dtype>>`[nelts, dt]

fn __add__(self, rhs: SIMD[dt, nelts]) -> SIMD[dt, nelts]:
    return __mlir_op.`pop.add`(self.value, rhs.value)
```

## Week of 2022-11-14

- Implement a magic `_mlir_type` declaration that can be used to access any MLIR type. E.g. `_mlir_type.f64`.
- Add an `fn` declaration. These are like `def` declarations, but are more strict in a few ways: they require type annotations on arguments, don't allow implicit variable declarations in their body, and make their arguments `rvalues` instead of `lvalues`.
- Implemented Swift-style backtick identifiers, which are useful for code migration where names may collide with new keywords.
- A new `_include` directive has been added that performs source-level textual includes. This is temporary until we have an `import` model.
- Implement IR generation for arithmetic operators like `+` and `*` in terms of the `_add_` and `_mul_` methods.
- Added support for `break` and `continue` statements, as well as early returns inside loops and conditionals!
- Implemented augmented assignment operators, like `+=` and `@=`.
- Mojo now has access to generating any MLIR operations (without regions) with a new `_mlir_op` magic declaration. We can start to build out the language's builtin types with this:

```
struct Int:
    var value: _mlir_type.index

fn __add__(self, rhs: Int) -> Int:
    return __mlir_op.`index.add`(self.value, rhs.value)
```

Attributes can be attached to the declaration with subscript `[]` syntax, and an explicit result type can be specified with a special `_type` attribute if it cannot be inferred. Attributes can be accessed via the `_mlir_attr` magic decl:

```
_mlir_op.`index.cmp`[
    _type: _mlir_type.i1,
    pred: _mlir_attr.`#index<cmp_predicate slt>`
](lhs, rhs)
```

- Improved diagnostics emissions with ranges! Now errors highlight the whole section of code and not just the first character.

## Week of 2022-11-07

- Implemented the `@interface` and `@implements` decorators, which provide access to KGEN generator interfaces. A function marked as an `@interface` has no body, but it can be implemented by multiple other functions.

```
@interface
def add(lhs: index, rhs: index):

@implements(add)
def normal_add(lhs: index, rhs: index) -> index:
    return lhs + rhs

@implements(add)
def slow_add(lhs: index, rhs: index) -> index:
    wait(1000)
    return normal_add(lhs, rhs)
```

- Support for static struct methods and initializer syntax has been added. Initializing a struct with `Foo()` calls an implicitly static `_new_` method. This method should be used instead of `_init_` inside structs.

```
struct Foo:
    var value: index

def __new__() -> Foo:
```

```

var result: Foo
result.value = Foo.return_a_number() # static method!
return result

@staticmethod
def return_a_number() -> index:
    return 42

```

- Full by-ref argument support. It's now possible to define in-place operators like `__iadd__` and functions like `swap(x, y)` correctly.
- Implemented support for field extract from rvalues, like `x.value` where `x` is not an lvalue (var declaration or by-ref function argument).

## October 2022

### Week of 2022-10-31

- Revised return handling so that a return statement with no expression is syntax sugar for `return None`. This enables early exits in functions that implicitly return `None` to be cleaner:

```
def just_return():
    return
```

- Added support for parsing more expressions: if-else, bitwise operators, shift operators, comparisons, floor division, remainder, and matmul.
- The type of the `self` argument can now be omitted on member methods.

### Week of 2022-10-24

- Added parser support for right-associativity and unary ops, like the power operator `a ** b ** c` and negation operator `-a`.
- Add support for `&expr` in Mojo, which allows denoting a by-ref argument in functions. This is required because the `self` type of a struct method is implicitly a pointer.
- Implemented support for parametric function declarations, such as:

```

struct SIMD[dt: DType, width: index]:
    fn struct_method(self: &SIMD[dt, width]):
        pass

def fancy_add[dt: DType, width: index](
    lhs: SIMD[dt, width], rhs: SIMD[dt, width]) -> index:
    return width

```

### Week of 2022-10-17

- Added explicit variable declarations with `var`, for declaring variables both inside functions and structs, with support for type references. Added `index` as a temporary built-in type.

```
def foo(lhs: index, rhs: index) -> index:
    var result: index = lhs + rhs
    return result
```

- Implemented support for parsing struct declarations and references to type declarations in functions! In `def`, the type can be omitted to signal an object type.

```

struct Foo:
    var member: index

def bar(x: Foo, obj) -> index:
    return x.member

```

- Implemented parser support for `if` statements and `while` loops!

```

def if_stmt(c: index, a: index, b: index) -> index:
    var result: index = 0
    if c:
        result = a
    else:
        result = b
    return result

def while_stmt(init: index):
    while init > 1:
        init = init - 1

```

- Significantly improved error emission and handling, allowing the parser to emit multiple errors while parsing a file.

### Week of 2022-10-10

- Added support for parsing integer, float, and string literals.
- Implemented parser support for function input parameters and results. You can now write parametric functions like,

```
def foo[param: Int](arg: Int) -> Int:
    result = param + arg
    return result
```

### Week of 2022-10-03

- Added some basic parser scaffolding and initial parser productions, including trivial expressions and assignment parser productions.

- Implemented basic scope handling and function IR generation, with support for forward declarations. Simple functions like,

```
def foo(x: Int):
```

Now parse! But all argument types are hard-coded to the MLIR `index` type.

- Added IR emission for simple arithmetic expressions on builtin types, like `x + y`.

## September 2022

### Week of 2022-09-26

- Mojo's first patch to add a lexer was Sep 27, 2022.
- Settled on `[]` for Mojo generics instead of `<>`. Square brackets are consistent with Python generics and don't have the less than ambiguity other languages have.

# MojoFAQ

Answers to questions we expect about Mojo.

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!

# Mojo community

Resources to share feedback, report issues, and chat.

Mojo is still very young, but we believe an active community and a strong feedback pipeline is key to its success.

We'd love to hear from you through the following community channels.

## [Ask a question](#)

[See existing GitHub Discussion posts, ask new questions, and share your ideas.](#)

This is a forum for ideas and questions, moderated by the Modular team.

## [Report an issue](#)

[Report bugs or other issues with the Mojo SDK or Mojo Playground.](#)

[Before reporting an issue, see the Mojo roadmap & sharp edges.](#)

## [Chat on Discord](#)

[Join our realtime chat on Discord to discuss the Mojo language and tools with the community.](#)

This is a community space where you can chat with other Mojo developers in realtime.

# Mojo community

Resources to share feedback, report issues, and chat.

Mojo is still very young, but we believe an active community and a strong feedback pipeline is key to its success.

We'd love to hear from you through the following community channels.

## [Ask a question](#)

[See existing GitHub Discussion posts, ask new questions, and share your ideas.](#)

This is a forum for ideas and questions, moderated by the Modular team.

## [Report an issue](#)

[Report bugs or other issues with the Mojo SDK or Mojo Playground.](#)

[Before reporting an issue, see the Mojo roadmap & sharp edges.](#)

## [Chat on Discord](#)

[Join our realtime chat on Discord to discuss the Mojo language and tools with the community.](#)

This is a community space where you can chat with other Mojo developers in realtime.

# Get started with Mojo

Get the Mojo SDK or try coding in the Mojo Playground.

Mojo is now available for local development!

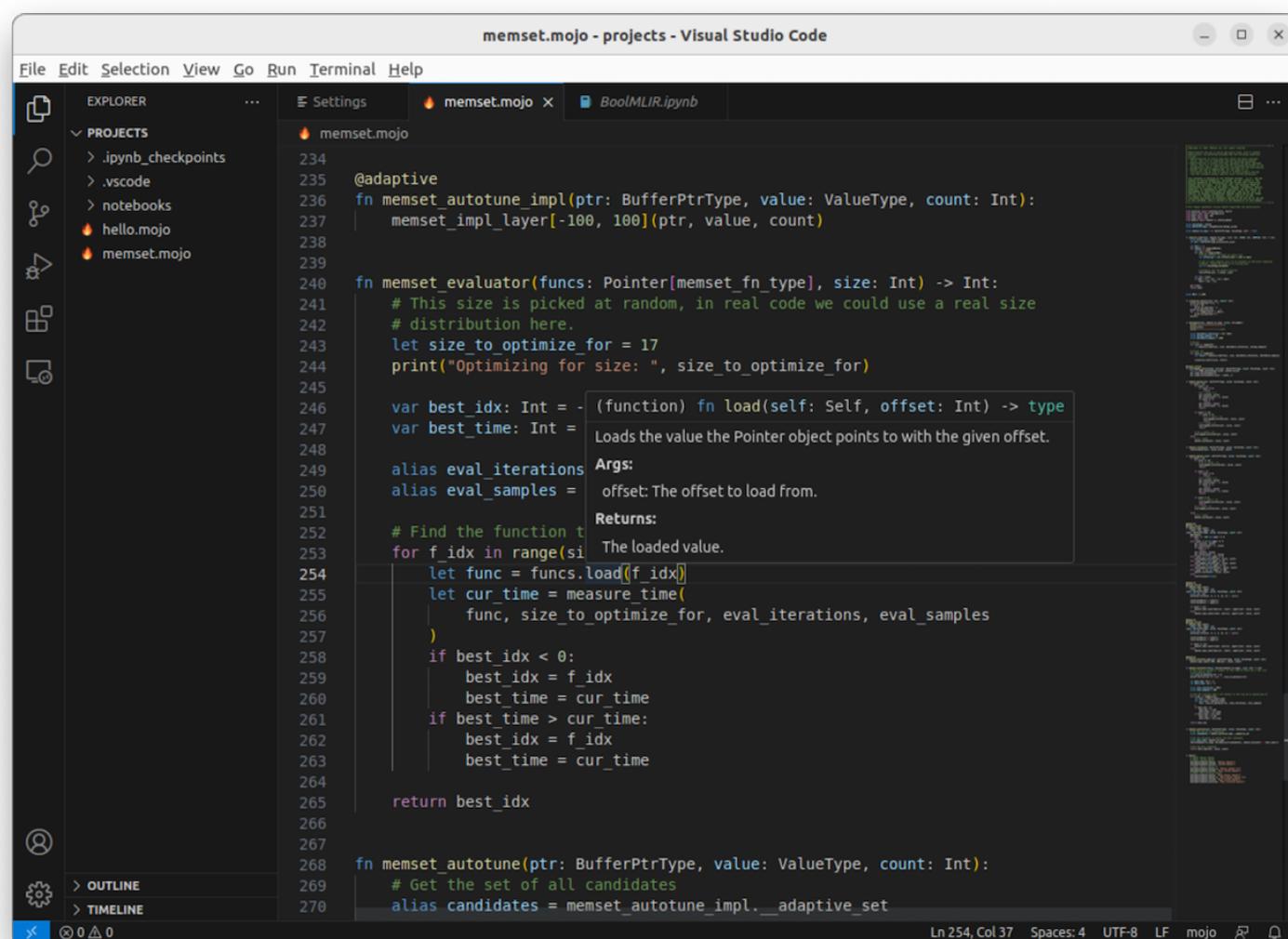
[Download Now](#) 

The Mojo SDK is currently available for Ubuntu Linux systems and macOS systems running on Apple silicon. Support for Windows is coming soon. Our setup guide also includes instructions about how to develop from Windows or Intel macOS using a container or remote Linux system. Alternatively, you can also experiment with Mojo using our web-based [Mojo Playground](#).

## Get the Mojo SDK

The Mojo SDK includes everything you need for local Mojo development, including the Mojo standard library and the [Mojo command-line interface](#) (CLI). The Mojo CLI can start a REPL programming environment, compile and run Mojo source files, format source files, and more.

We've also published a [Mojo extension for Visual Studio Code](#) to provide a first-class developer experience with features like code completion, quick fixes, and hover help for Mojo APIs.



```
memset.mojo - projects - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER ... Settings memset.mojo x BoolMLIR.ipynb
memset.mojo
234
235 @adaptive
236 fn memset_autotune_impl(ptr: BufferPtrType, value: ValueType, count: Int):
237     memset_impl_layer[-100, 100](ptr, value, count)
238
239
240 fn memset_evaluator(funcs: Pointer[memset_fn_type], size: Int) -> Int:
241     # This size is picked at random, in real code we could use a real size
242     # distribution here.
243     let size_to_optimize_for = 17
244     print("Optimizing for size: ", size_to_optimize_for)
245
246     var best_idx: Int = -
247     var best_time: Int =
248
249     alias eval_iterations
250     alias eval_samples =
251
252     # Find the function to use
253     for f_idx in range(size):
254         let func = funcs.load(f_idx)
255         let cur_time = measure_time(
256             func, size_to_optimize_for, eval_iterations, eval_samples
257         )
258         if best_idx < 0:
259             best_idx = f_idx
260             best_time = cur_time
261         if best_time > cur_time:
262             best_idx = f_idx
263             best_time = cur_time
264
265     return best_idx
266
267
268 fn memset_autotune(ptr: BufferPtrType, value: ValueType, count: Int):
269     # Get the set of all candidates
270     alias candidates = memset_autotune_impl.__adaptive_set
```

## System requirements

To use the Mojo SDK, you need a system that meets these specifications:

Linux:

- Ubuntu 20.04/22.04 LTS
- x86-64 CPU (with [SSE4.2 or newer](#)) and a minimum of 8 GiB memory
- Python 3.8 - 3.11
- g++ or clang++ C++ compiler

Mac:

- Apple silicon (M1 or M2 processor)
- macOS Monterey (12) or later
- Python 3.8 - 3.11
- Command-line tools for Xcode, or Xcode

Support for Windows will be added in a future release.

## Install Mojo

The Mojo SDK is available through the [Modular CLI tool](#), which works like a package manager to install and update Mojo. Use the following link to log into the Modular developer console, where you can get the Modular CLI and then install Mojo:

[Download Now](#) 

Then get started with [Hello, world!](#)

**Note:** To help us improve Mojo, we collect some basic system information and crash reports. [Learn more.](#)

## Update Mojo

Mojo is a work in progress and we will release regular updates to the Mojo language and SDK tools. For information about each release, see the [Mojo changelog](#).

To check your current Mojo version, use the `--version` option:

```
mojo --version
```

To update to the latest Mojo version, use the `modular update` command:

```
modular update mojo
```

## Update the Modular CLI

We may also release updates to the `modular` tool. Run the following commands to update the CLI on your system.

Linux:

```
sudo apt update  
sudo apt install modular
```

Mac:

```
brew update  
brew upgrade modular
```

# Develop in the Mojo Playground

Instead of downloading the Mojo SDK, you can also experiment with Mojo in our hosted Jupyter notebook environment called Mojo Playground. This is a hosted version of [JupyterLab](#) that's running our latest Mojo kernel.

To get access, just [log in to the Mojo Playground here](#).

The screenshot shows the JupyterLab interface for the Mojo Playground. On the left, there is a file browser window titled "HelloMojo.ipynb" showing files like "Mandelbrot.ipynb", "HelloMojo.ipynb", and "BoolMLIR.ipynb". The main area is a code editor titled "Parallelizing Mandelbrot" containing the following Mojo code:

```
[10]: from Functional import parallelize

def compute_mandelbrot_simd_parallel() -> Matrix:
    # create a matrix. Each element of the matrix corresponds to a pixel
    var result = Matrix(xn, yn)

    let dx = (xmax - xmin) / xn
    let dy = (ymax - ymin) / yn

    alias SIMD_width = dtype SIMD_width[DType.f32]()

    @parameter
    fn _process_row(row:Int):
        var y = ymin + dy*row
        var x = xmin
    @parameter
    fn _process_simd_element[simd_width:Int](col: Int):
        let c = ComplexGenericSIMD[DType.f32, SIMD_width](dx*ioata[simd_width, DType.f32]() + x,
                                                          SIMD[DType.f32, SIMD_width](y))
        result.store[simd_width](col, row, mandelbrot_kernel_simd[simd_width](c))
        x += SIMD_width*dx

    vectorize[simd_width, _process_simd_element](xn)

    parallelize[_process_row](yn)
    return result

make_plot(compute_mandelbrot_simd_parallel())
print("finished")
```

## What to expect

- The Mojo Playground is a [JupyterHub](#) environment in which you get a private volume associated with your account, so you can create your own notebooks and they'll be saved across sessions.
- We've included a handful of notebooks to show you Mojo basics and demonstrate its capabilities.
- The number of vCPU cores available in your cloud instance may vary, so baseline performance is not representative of the language. However, as you will see in the included Matmul.ipynb notebook, Mojo's relative performance over Python is significant.
- There might be some bugs. Please [report issues and feedback on GitHub](#).

## Tips

- If you want to keep any edits to the included notebooks, **rename the notebook files**. These files will reset upon any server refresh or update, sorry. So if you rename the files, your changes will be safe.
- You can use `%%python` at the top of a notebook cell and write normal Python code. Variables, functions, and imports defined in a Python cell are available for access in subsequent Mojo cells.

## Caveats

- Did we mention that the included notebooks will lose your changes?  
**Rename the files if you want to save your changes.**
- The Mojo environment does not have network access, so you cannot install other tools or Python packages. However, we've included a variety of popular Python packages, such as `numpy`, `pandas`, and `matplotlib` (see how to [import Python modules](#)).
- Redefining implicit variables is not supported (variables without a `let` or `var` in front). If you'd like to redefine a variable across notebook cells, you must introduce the variable with `var` (`let` variables are immutable).
- You can't use global variables inside functions—they're only visible to other global variables.
- For a longer list of things that don't work yet or have pain-points, see the [Mojo roadmap and sharp edges](#).

# Why Mojo

A backstory and rationale for why we created the Mojo language.

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.

To see all the features we've built with Mojo so far, see the [Mojo programming manual](#).

# Mojo programming manual

A tour of major Mojo language features with code examples.

Mojo is a programming language that is as easy to use as Python but with the performance of C++ and Rust. Furthermore, Mojo provides the ability to leverage the entire Python library ecosystem.

Mojo achieves this feat by utilizing next-generation compiler technologies with integrated caching, multithreading, and cloud distribution technologies. Furthermore, Mojo's autotuning and compile-time metaprogramming features allow you to write code that is portable to even the most exotic hardware.

More importantly, **Mojo allows you to leverage the entire Python ecosystem** so you can continue to use tools you are familiar with. Mojo is designed to become a **superset** of Python over time by preserving Python's dynamic features while adding new primitives for [systems programming](#). These new system programming primitives will allow Mojo developers to build high-performance libraries that currently require C, C++, Rust, CUDA, and other accelerator systems. By bringing together the best of dynamic languages and systems languages, we hope to provide a **unified** programming model that works across levels of abstraction, is friendly for novice programmers, and scales across many use cases from accelerators through to application programming and scripting.

This document is an introduction to the Mojo programming language, not a complete language guide. It assumes knowledge of Python and systems programming concepts. At the moment, Mojo is still a work in progress and the documentation is targeted to developers with systems programming experience. As the language grows and becomes more broadly available, we intend for it to be friendly and accessible to everyone, including beginner programmers. It's just not there today.

## Using the Mojo compiler

With the [Mojo SDK](#), you can run a Mojo program from a terminal just like you can with Python. So if you have a file named `hello.mojo` (or `hello.□`—yes, the file extension can be an emoji!), just type `mojo hello.mojo`:

```
$ cat hello.□
def main():
    print("hello world")
    for x in range(9, 0, -3):
        print(x)
$ mojo hello.□
hello world
9
6
3
$ □
```

Again, you can use either the `.□` or `.mojo` suffix.

For more details about the Mojo compiler tools, see the [mojo CLI docs](#).

## Basic systems programming extensions

Given our goal of compatibility and Python's strength with high-level applications and dynamic APIs, we don't have to spend much time explaining how those portions of the language work. On the other hand, Python's support for systems programming is mainly delegated to C, and we want to provide a single system that is great in that world. As such, this section breaks down each major component and feature and describes how to use them with examples.

## let and var declarations

Inside a `def` in Mojo, you may assign a value to a name and it implicitly creates a function scope variable just like in Python. This provides a very dynamic and low-ceremony way to write code, but it is a challenge for two reasons:

1. Systems programmers often want to declare that a value is immutable for type-safety and performance.
2. They may want to get an error if they mistype a variable name in an assignment.

To support this, Mojo provides scoped runtime value declarations: `let` is immutable, and `var` is mutable. These values use lexical scoping and support name shadowing:

```
def your_function(a, b):
    let c = a
    # Uncomment to see an error:
    # c = b # error: c is immutable

    if c != b:
        let d = b
        print(d)
```

```
your_function(2, 3) ↴
```

```
3
```

`let` and `var` declarations support type specifiers as well as patterns, and late initialization:

```
def your_function():
    let x: Int = 42
    let y: Float64 = 17.0

    let z: Float32
    if x != 0:
        z = 1.0
    else:
        z = foo()
    print(z)

def foo() -> Float32:
    return 3.14
```

```
your_function() ↴
```

```
1.0
```

Note that `let` and `var` are completely optional when in a `def` function (you can instead use implicitly declared values, just like Python), but they're required for all variables in an `fn` function.

Also beware that when using Mojo in a REPL environment (such as this notebook), top-level variables (variables that live outside a function or struct) are treated like variables in a `def`, so they allow implicit value type declarations (they do not require `var` or `let` declarations, nor type declarations). This matches the Python REPL behavior.

## struct types

Mojo is based on MLIR and LLVM, which offer a cutting-edge compiler and code generation system used in many programming languages. This lets us have better control over data organization, direct access to data fields, and other ways to improve performance. An important feature of modern systems programming languages is the ability to build high-level and safe abstractions on top of these complex, low-level operations without any performance loss. In Mojo, this is provided by the `struct` type.

A `struct` in Mojo is similar to a Python `class`: they both support methods, fields, operator overloading, decorators for metaprogramming, etc. Their differences are as follows:

- Python classes are dynamic: they allow for dynamic dispatch, monkey-patching (or “swizzling”), and dynamically binding instance properties at runtime.
- Mojo structs are static: they are bound at compile-time (you cannot add methods at runtime). Structs allow you to trade flexibility for performance while being safe and easy to use.

Here's a simple definition of a struct:

```
struct MyPair:
    var first: Int
    var second: Int

    # We use 'fn' instead of 'def' here - we'll explain that soon
    fn __init__(inout self, first: Int, second: Int):
        self.first = first
        self.second = second

    fn __lt__(self, rhs: MyPair) -> Bool:
        return self.first < rhs.first or
            (self.first == rhs.first and
             self.second < rhs.second)
```

Syntactically, the biggest difference compared to a Python class is that all instance properties in a struct **must** be explicitly declared with a var or let declaration.

In Mojo, the structure and contents of a “struct” are set in advance and can't be changed while the program is running. Unlike in Python, where you can add, remove, or change attributes of an object on the fly, Mojo doesn't allow that for structs. This means you can't use del to remove a method or change its value in the middle of running the program.

However, the static nature of struct has some great benefits! It helps Mojo run your code faster. The program knows exactly where to find the struct's information and how to use it without any extra steps or delays.

Mojo's structs also work really well with features you might already know from Python, like operator overloading (which lets you change how math symbols like + and - work with your own data). Furthermore, *all* the “standard types” (like Int, Bool, String and even Tuple) are made using structs. This means they're part of the standard set of tools you can use, rather than being hardwired into the language itself. This gives you more flexibility and control when writing your code.

If you're wondering what the inout means on the self argument: this indicates that the argument is mutable and changes made inside the function are visible to the caller. For details, see below about [inout arguments](#).

## Int vs int

In Mojo, you might notice that we use Int (with a capital “I”), which is different from Python's int (with a lowercase “i”). This difference is on purpose, and it's actually a good thing!

In Python, the int type can handle really big numbers and has some extra features, like checking if two numbers are the same object. But this comes with some extra baggage that can slow things down. Mojo's Int is different. It's designed to be simple, fast, and tuned for your computer's hardware to handle quickly.

We made this choice for two main reasons:

1. We want to give programmers who need to work closely with computer hardware (systems programmers) a transparent and reliable way to interact with hardware. We don't want to rely on fancy tricks (like JIT compilers) to make things faster.

2. We want Mojo to work well with Python without causing any issues. By using a different name (Int instead of int), we can keep both types in Mojo without changing how Python's int works.

As a bonus, Int follows the same naming style as other custom data types you might create in Mojo. Additionally, Int is a struct that's included in Mojo's standard set of tools.

## Strong type checking

Even though you can still use flexible types like in Python, Mojo lets you use strict type checking. Type-checking can make your code more predictable, manageable, and secure.

One of the primary ways to employ strong type checking is with Mojo's struct type. A struct definition in Mojo defines a compile-time-bound name, and references to that name in a type context are treated as a strong specification for the value being defined. For example, consider the following code that uses the MyPair struct shown above:

```
def pair_test() -> Bool:  
    let p = MyPair(1, 2)  
    # Uncomment to see an error:  
    # return p < 4 # gives a compile time error  
    return True
```

If you uncomment the first return statement and run it, you'll get a compile-time error telling you that 4 cannot be converted to MyPair, which is what the right-hand-side of `_lt_()` requires (in the MyPair definition).

This is a familiar experience when working with systems programming languages, but it's not how Python works. Python has syntactically identical features for [MyPy](#) type annotations, but they are not enforced by the compiler: instead, they are hints that inform static analysis. By tying types to specific declarations, Mojo can handle both the classical type annotation hints and strong type specifications without breaking compatibility.

Type checking isn't the only use-case for strong types. Since we know the types are accurate, we can optimize the code based on those types, pass values in registers, and be as efficient as C for argument passing and other low-level details. This is the foundation of the safety and predictability guarantees Mojo provides to systems programmers.

## Overloaded functions and methods

Like Python, you can define functions in Mojo without specifying argument data types and Mojo will handle them dynamically. This is nice when you want expressive APIs that just work by accepting arbitrary inputs and let dynamic dispatch decide how to handle the data. However, when you want to ensure type safety, as discussed above, Mojo also offers full support for overloaded functions and methods.

This allows you to define multiple functions with the same name but with different arguments. This is a common feature seen in many languages, such as C++, Java, and Swift.

When resolving a function call, Mojo tries each candidate and uses the one that works (if only one works), or it picks the closest match (if it can determine a close match), or it reports that the call is ambiguous if it can't figure out which one to pick. In the latter case, you can resolve the ambiguity by adding an explicit cast on the call site.

Let's look at an example:

```
struct Complex:  
    var re: Float32  
    var im: Float32  
  
fn __init__(inout self, x: Float32):  
    """Construct a complex number given a real number."""  
    self.re = x
```

```

self.im = 0.0

fn __init__(inout self, r: Float32, i: Float32):
    """Construct a complex number given its real and imaginary components."""
    self.re = r
    self.im = i

```

You can overload methods in structs and classes and overload module-level functions.

Mojo doesn't support overloading solely on result type, and doesn't use result type or contextual type information for type inference, keeping things simple, fast, and predictable. Mojo will never produce an "expression too complex" error, because its type-checker is simple and fast by definition.

Again, if you leave your argument names without type definitions, then the function behaves just like Python with dynamic types. As soon as you define a single argument type, Mojo will look for overload candidates and resolve function calls as described above.

Although we haven't discussed parameters yet (they're different from function arguments), you can also [overload functions and methods based on parameters](#).

## fn definitions

The extensions above are the cornerstone that provides low-level programming and provide abstraction capabilities, but many systems programmers prefer more control and predictability than what def in Mojo provides. To recap, def is defined by necessity to be very dynamic, flexible and generally compatible with Python: arguments are mutable, local variables are implicitly declared on first use, and scoping isn't enforced. This is great for high level programming and scripting, but is not always great for systems programming. To complement this, Mojo provides an fn declaration which is like a "strict mode" for def.

Alternative: instead of using a new keyword like fn, we could instead add a modifier or decorator like @strict def. However, we need to take new keywords anyway and there is little cost to doing so. Also, in practice in systems programming domains, fn is used all the time so it probably makes sense to make it first class.

As far as a caller is concerned, fn and def are interchangeable: there is nothing a def can provide that a fn cannot (and vice versa). The difference is that a fn is more limited and controlled on the *inside* of its body (alternatively: pedantic and strict). Specifically, fns have a number of limitations compared to def functions:

1. Argument values default to being immutable in the body of the function (like a let), instead of mutable (like a var). This catches accidental mutations, and permits the use of non-copyable types as arguments.
2. Argument values require a type specification (except for self in a method), catching accidental omission of type specifications. Similarly, a missing return type specifier is interpreted as returning None instead of an unknown return type. Note that both can be explicitly declared to return object, which allows one to opt-in to the behavior of a def if desired.
3. Implicit declaration of local variables is disabled, so all locals must be declared. This catches name typos and dovetails with the scoping provided by let and var.
4. Both support raising exceptions, but this must be explicitly declared on a fn with the raises keyword.

Programming patterns will vary widely across teams, and this level of strictness will not be for everyone. We expect that folks who are used to C++ and already use MyPy-style type annotations in Python to prefer the use of fns, but higher level programmers and ML researchers to continue to use def. Mojo allows you to freely intermix def and fn declarations, e.g.

implementing some methods with one and others with the other, and allows each team or programmer to decide what is best for their use-case.

For more about argument behavior in Mojo functions, see the section below about [Argument passing control and memory ownership](#).

## The `__copyinit__`, `__moveinit__`, and `__takeinit__` special methods

Mojo supports full “value semantics” as seen in languages like C++ and Swift, and it makes defining simple aggregates of fields very easy with the [@value decorator](#).

For advanced use cases, Mojo allows you to define custom constructors (using Python’s existing `__init__` special method), custom destructors (using the existing `__del__` special method) and custom copy and move constructors using the `__copyinit__`, `__moveinit__` and `__takeinit__` special methods.

These low-level customization hooks can be useful when doing low level systems programming, e.g. with manual memory management. For example, consider a dynamic string type that needs to allocate memory for the string data when constructed and destroy it when the value is destroyed:

```
from memory.unsafe import Pointer

struct HeapArray:
    var data: Pointer[Int]
    var size: Int

    fn __init__(inout self, size: Int, val: Int):
        self.size = size
        self.data = Pointer[Int].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, val)

    fn __del__(owned self):
        self.data.free()

    fn dump(self):
        print_no_newline("[")
        for i in range(self.size):
            if i > 0:
                print_no_newline(", ")
            print_no_newline(self.data.load(i))
        print("]")
```

This array type is implemented using low level functions to show a simple example of how this works. However, if you try to copy an instance of `HeapArray` with the `=` operator, you might be surprised:

```
var a = HeapArray(3, 1)
a.dump() # Should print [1, 1, 1]
# Uncomment to see an error:
# var b = a # ERROR: Vector doesn't implement __copyinit__

var b = HeapArray(4, 2)
b.dump() # Should print [2, 2, 2, 2]
a.dump() # Should print [1, 1, 1]

[1, 1, 1]
[2, 2, 2, 2]
[1, 1, 1]
```

If you uncomment the line to copy `a` into `b`, you’ll see that Mojo doesn’t allow you to make a copy of our array: `HeapArray` contains an instance of `Pointer` (which is equivalent to a low-level C pointer), and Mojo doesn’t know what kind of data it points to or how to copy it. More generally, some types (like atomic numbers) cannot be copied or moved around because their address provides an **identity** just like a class instance does.

In this case, we do want our array to be copyable. To enable this, we have to implement the `__copyinit__` special method, which is conventionally implemented like this:

```
struct HeapArray:  
    var data: Pointer[Int]  
    var size: Int  
  
    fn __init__(inout self, size: Int, val: Int):  
        self.size = size  
        self.data = Pointer[Int].alloc(self.size)  
        for i in range(self.size):  
            self.data.store(i, val)  
  
    fn __copyinit__(inout self, other: Self):  
        self.size = other.size  
        self.data = Pointer[Int].alloc(self.size)  
        for i in range(self.size):  
            self.data.store(i, other.data.load(i))  
  
    fn __del__(owned self):  
        self.data.free()  
  
    fn dump(self):  
        print_no_newline("[")  
        for i in range(self.size):  
            if i > 0:  
                print_no_newline(", ")  
            print_no_newline(self.data.load(i))  
        print("]")
```

With this implementation, our code above works correctly and the `b = a` copy produces a logically distinct instance of the array with its own lifetime and data:

```
var a = HeapArray(3, 1)  
a.dump()    # Should print [1, 1, 1]  
# This is no longer an error:  
var b = a  
  
b.dump()    # Should print [1, 1, 1]  
a.dump()    # Should print [1, 1, 1]
```

```
[1, 1, 1]  
[1, 1, 1]  
[1, 1, 1]
```

Mojo also supports the `__moveinit__` method which allows both Rust-style moves (which transfers a value from one place to another when the source lifetime ends) and the `__takeinit__` method for C++-style moves (where the contents of a value is logically transferred out of the source, but its destructor is still run), and allows defining custom move logic. For more detail, see the [Value Lifecycle](#) section below.

Mojo provides full control over the lifetime of a value, including the ability to make types copyable, move-only, and not-movable. This is more control than languages like Swift and Rust offer, which require values to at least be movable. If you are curious how existing can be passed into the `__copyinit__` method without itself creating a copy, check out the section on [Borrowed arguments](#) below.

## Argument passing control and memory ownership

In both Python and Mojo, much of the language revolves around function calls: a lot of the (apparently) built-in behaviors are implemented in the standard library with “dunder” (double-underscore) methods. Inside these magic functions is where a lot of memory ownership is determined through argument passing.

Let's review some details about how Python and Mojo pass arguments:

- All values passed into a *Python* `def` function use reference semantics. This means the function can modify mutable objects passed into it and those changes are visible outside the function. However, the behavior is sometimes surprising for the uninitiated, because you can change the object that an argument points to and that change is not visible outside the function.
- All values passed into a *Mojo* `def` function use value semantics by default. Compared to Python, this is an important difference: A *Mojo* `def` function receives a copy of all arguments—it can modify arguments inside the function, but the changes are **not** visible outside the function.
- All values passed into a *Mojo* [fn function](#) are immutable references by default. This means the function can read the original object (it is *not* a copy), but it cannot modify the object at all.

This convention for immutable argument passing in a *Mojo* `fn` is called “borrowing.” In the following sections, we’ll explain how you can change the argument passing behavior in *Mojo*, for both `def` and `fn` functions.

## Why argument conventions are important

In *Python*, all fundamental values are references to objects—as described above, a *Python* function can modify the original object. Thus, *Python* developers are used to thinking about everything as reference semantic. However, at the *CPython* or machine level, you can see that the references themselves are actually passed *by-copy*—*Python* copies a pointer and adjusts reference counts.

This *Python* approach provides a comfortable programming model for most people, but it requires all values to be heap-allocated (and results are occasionally surprising results due to reference sharing). *Mojo* classes (TODO: will) follow the same reference-semantic approach for most objects, but this isn’t practical for simple types like integers in a systems programming context. In these scenarios, we want the values to live on the stack or even in hardware registers. As such, *Mojo* structs are always inlined into their container, whether that be as the field of another type or into the stack frame of the containing function.

This raises some interesting questions: How do you implement methods that need to mutate `self` of a structure type, such as `__iadd__`? How does `let` work, and how does it prevent mutation? How are the lifetimes of these values controlled to keep *Mojo* a memory-safe language?

The answer is that the *Mojo* compiler uses dataflow analysis and type annotations to provide full control over value copies, aliasing of references, and mutation control. These features are similar in many ways to features in the *Rust* language, but they work somewhat differently in order to make *Mojo* easier to learn, and they integrate better into the *Python* ecosystem without requiring a massive annotation burden.

In the following sections, you’ll learn about how you can control memory ownership for objects passed into *Mojo* `fn` functions.

## Immutable arguments (borrowed)

A borrowed object is an **immutable reference** to an object that a function receives, instead of receiving a copy of the object. So the callee function has full read-and-execute access to the object, but it cannot modify it (the caller still has exclusive “ownership” of the object).

For example, consider this struct that we don’t want to copy when passing around instances of it:

```
# Don't worry about this code yet. It's just needed for the function below.
# It's a type so expensive to copy around so it does not have a
# __copyinit__ method.
struct SomethingBig:
    var id_number: Int
```

```

var huge: HeapArray
fn __init__(inout self, id: Int):
    self.huge = HeapArray(1000, 0)
    self.id_number = id

# self is passed by-reference for mutation as described above.
fn set_id(inout self, number: Int):
    self.id_number = number

# Arguments like self are passed as borrowed by default.
fn print_id(self): # Same as: fn print_id(borrowed self):
    print(self.id_number)_

```

When passing an instance of `SomethingBig` to a function, it's necessary to pass a reference because `SomethingBig` cannot be copied (it has no `_copyinit_` method). And, as mentioned above, `fn` arguments are immutable references by default, but you can explicitly define it with the `borrowed` keyword as shown in the `use_something_big()` function here:

```

fn use_something_big(borrowed a: SomethingBig, b: SomethingBig):
    """'a' and 'b' are both immutable, because 'borrowed' is the default."""
    a.print_id()
    b.print_id()

let a = SomethingBig(10)
let b = SomethingBig(20)
use_something_big(a, b)_

```

10

20

This default applies to all arguments uniformly, including the `self` argument of methods. This is much more efficient when passing large values or when passing expensive values like a reference-counted pointer (which is the default for Python/Mojo classes), because the copy constructor and destructor don't have to be invoked when passing the argument.

Because the default argument convention for `fn` functions is `borrowed`, Mojo has simple and logical code that does the right thing by default. For example, we don't want to copy or move all of `SomethingBig` just to invoke the `print_id()` method, or when calling `use_something_big()`.

This borrowed argument convention is similar in some ways to passing an argument by `const&` in C++, which avoids a copy of the value and disables mutability in the callee. However, the borrowed convention differs from `const&` in C++ in two important ways:

1. The Mojo compiler implements a borrow checker (similar to Rust) that prevents code from dynamically forming mutable references to a value when there are immutable references outstanding, and it prevents multiple mutable references to the same value. You are allowed to have multiple borrows (as the call to `use_something_big` does above) but you cannot pass something by mutable reference and borrow at the same time. (TODO: Not currently enabled).
2. Small values like `Int`, `Float`, and `SIMD` are passed directly in machine registers instead of through an extra indirection (this is because they are declared with the [@register\\_passable\\_decorator](#)). This is a [significant performance enhancement](#) when compared to languages like C++ and Rust, and moves this optimization from every call site to being declarative on a type.

Similar to Rust, Mojo's borrow checker enforces the exclusivity of invariants. The major difference between Rust and Mojo is that Mojo does not require a sigil on the caller side to pass by borrow. Also, Mojo is more efficient when passing small values, and Rust defaults to moving values instead of passing them around by borrow. These policy and syntax decisions allow Mojo to provide an easier-to-use programming model.

## Mutable arguments (`inout`)

On the other hand, if you define an `fn` function and want an argument to be mutable, you must declare the argument as mutable with the `inout` keyword.

**Tip:** When you see `inout`, it means any changes made to the argument **inside** the function are visible **outside** the function.

Consider the following example, in which the `_iadd_` function (which implements the in-place add operation such as `x += 2`) tries to modify `self`:

```
struct MyInt:  
    var value: Int  
  
    fn __init__(inout self, v: Int):  
        self.value = v  
  
    fn __copyinit__(inout self, other: MyInt):  
        self.value = other.value  
  
    # self and rhs are both immutable in __add__.  
    fn __add__(self, rhs: MyInt) -> MyInt:  
        return MyInt(self.value + rhs.value)  
  
    # ... but this cannot work for __iadd__  
    # Uncomment to see the error:  
    #fn __iadd__(self, rhs: Int):  
    #    self = self + rhs # ERROR: cannot assign to self! □
```

If you uncomment the `__iadd__()` method, you'll get a compiler error.

The problem here is that `self` is immutable because this is a Mojo `fn` function, so it can't change the internal state of the argument (the default argument convention is borrowed). The solution is to declare that the argument is mutable by adding the `inout` keyword on the `self` argument name:

```
struct MyInt:  
    var value: Int  
  
    fn __init__(inout self, v: Int):  
        self.value = v  
  
    fn __copyinit__(inout self, other: MyInt):  
        self.value = other.value  
  
    # self and rhs are both immutable in __add__.  
    fn __add__(self, rhs: MyInt) -> MyInt:  
        return MyInt(self.value + rhs.value)  
  
    # ... now this works:  
    fn __iadd__(inout self, rhs: Int):  
        self = self + rhs □
```

Now the `self` argument is mutable in the function and any changes are visible in the caller, so we can perform in-place addition with `MyInt`:

```
var x: MyInt = 42  
x += 1  
print(x.value) # prints 43 as expected  
  
# However...  
let y = x  
# Uncomment to see the error:  
# y += 1 # ERROR: Cannot mutate 'let' value □
```

If you uncomment the last line above, mutation of the `let` value fails because it isn't possible to form a mutable reference to an immutable value (`let` makes the variable immutable).

Of course, you can declare multiple `inout` arguments. For example, you can define and use a swap function like this:

```
fn swap(inout lhs: Int, inout rhs: Int):
    let tmp = lhs
    lhs = rhs
    rhs = tmp

var x = 42
var y = 12
print(x, y) # Prints 42, 12
swap(x, y)
print(x, y) # Prints 12, 42
```

```
42 12
12 42
```

A very important aspect of this system is that it all composes correctly.

Notice that we don't call this argument passing "by reference." Although the `inout` convention is conceptually the same, we don't call it by-reference passing because the implementation may actually pass values using pointers.

## Transfer arguments (`owned` and `^`)

The final argument convention that Mojo supports is the `owned` argument convention. This convention is used for functions that want to take exclusive ownership over a value, and it is often used with the postfix `^` operator.

For example, imagine you're working with a move-only type like a unique pointer:

```
# This is not really a unique pointer, we just model its behavior here
# to serve the examples below.
struct UniquePointer:
    var ptr: Int

    fn __init__(inout self, ptr: Int):
        self.ptr = ptr

    fn __moveinit__(inout self, owned existing: Self):
        self.ptr = existing.ptr

    fn __del__(owned self):
        self.ptr = 0
```

While the `borrow` convention makes it easy to work with this unique pointer without ceremony, at some point you might want to transfer ownership to some other function. This is a situation where you want to use the `^` "transfer" operator with your movable type.

The `^` operator ends the lifetime of a value binding and transfers the value ownership to something else (in the following example, ownership is transferred to the `take_ptr()` function). To support this, you can define functions as taking `owned` arguments. For example, you can define `take_ptr()` to take ownership of an argument as follows:

```
fn take_ptr(owned p: UniquePointer):
    print("take_ptr")
    print(p.ptr)

fn use_ptr(borrowed p: UniquePointer):
    print("use_ptr")
    print(p.ptr)

fn work_with_unique_ptrs():
    let p = UniquePointer(100)
    use_ptr(p)    # Pass to borrowing function.
    take_ptr(p^)  # Pass ownership of the `p` value to another function.

    # Uncomment to see an error:
    # use_ptr(p) # ERROR: p is no longer valid here!
```

```
work_with_unique_ptrs()
```

```
use_ptr  
100  
take_ptr  
100
```

Notice that if you uncomment the second call to `use_ptr()`, you get an error because the `p` value has been transferred to the `take_ptr()` function and, thus, the `p` value is destroyed.

Because it is declared `owned`, the `take_ptr()` function knows it has unique access to the value. This is very important for things like unique pointers, and it's useful when you want to avoid copies.

For example, you will notably see the `owned` convention on destructors and on consuming move initializers. For example, our `HeapArray` struct defined earlier uses `owned` in its `__del__()` method, because you need to own a value to destroy it (or to steal its parts, in the case of a move constructor).

## Comparing def and fn argument passing

Mojo's `def` function is essentially just sugar for the `fn` function:

- A `def` argument without an explicit type annotation defaults to `Object`.
- A `def` argument without a convention keyword (such as `inout` or `owned`) is passed by implicit copy into a mutable var with the same name as the argument. (This requires that the type have a `__copyinit__` method.)

For example, these two functions have the same behavior:

```
def example(inout a: Int, b: Int, c):  
    # b and c use value semantics so they're mutable in the function  
    ...  
  
fn example(inout a: Int, b_in: Int, c_in: Object):  
    # b_in and c_in are immutable references, so we make mutable shadow copies  
    var b = b_in  
    var c = c_in  
    ...
```

The shadow copies typically add no overhead, because references for small types like `Object` are cheap to copy. The expensive part is adjusting the reference count, but that's eliminated by a move optimization.

## Python integration

It's easy to use Python modules you know and love in Mojo. You can import any Python module into your Mojo program and create Python types from Mojo types.

### Importing Python modules

To import a Python module in Mojo, just call `Python.import_module()` with the module name:

```
from python import Python  
  
# This is equivalent to Python's `import numpy as np`  
let np = Python.import_module("numpy")  
  
# Now use numpy as if writing in Python  
array = np.array([1, 2, 3])  
print(array)  
[1 2 3]
```

Yes, this imports Python NumPy, and you can import *any other Python module*.

Currently, you cannot import individual members (such as a single Python class or function)—you must import the whole Python module and then access members through the module name.

## Mojo types in Python

Mojo primitive types implicitly convert into Python objects. Today we support lists, tuples, integers, floats, booleans, and strings.

For example, given this Python function that prints Python types:

```
%%python
def type_printer(my_list, my_tuple, my_int, my_string, my_float):
    print(type(my_list))
    print(type(my_tuple))
    print(type(my_int))
    print(type(my_string))
    print(type(my_float))
```

You can pass the Python function Mojo types with no problem:

```
type_printer([0, 3], (False, True), 4, "orange", 3.4)
<class 'list'>
<class 'tuple'>
<class 'int'>
<class 'str'>
<class 'float'>
```

Notice that in a Jupyter notebook, the Python function declared above is automatically available to any Mojo code in following code cells.

Mojo doesn't have a standard Dictionary yet, so it is not yet possible to create a Python dictionary from a Mojo dictionary. You can work with Python dictionaries in Mojo though! To create a Python dictionary, use the `dict` method:

```
from python import Python
from python.object import PythonObject

var dictionary = Python.dict()
dictionary["fruit"] = "apple"
dictionary["starch"] = "potato"

var keys: PythonObject = ["fruit", "starch", "protein"]
var N: Int = keys.__len__().__index__()
print(N, "items")

for i in range(N):
    if Python.is_type(dictionary.get(keys[i]), Python.none()):
        print(keys[i], "is not in dictionary")
    else:
        print(keys[i], "is included")
```

3 items  
fruit is included  
starch is included  
protein is not in dictionary

## Importing local Python modules

If you have some local Python code you want to use in Mojo, just add the directory to the Python path and then import the module.

For example, suppose you have a Python file named `mypython.py`:

```
import numpy as np
```

```
def my_algorithm(a, b):
    array_a = np.random.rand(a, a)
    return array_a + b
```

Here's how you can import it and use it in a Mojo file:

```
from python import Python
Python.add_to_path("path/to/module")
let mypython = Python.import_module("mypython")

let c = mypython.my_algorithm(2, 3)
print(c)
```

There's no need to worry about memory management when using Python in Mojo. Everything just works because Mojo was designed for Python from the beginning.

## Parameterization: compile-time metaprogramming

One of Python's most amazing features is its extensible runtime metaprogramming features. This has enabled a wide range of libraries and provides a flexible and extensible programming model that Python programmers everywhere benefit from. Unfortunately, these features also come at a cost: because they are evaluated at runtime, they directly impact run-time efficiency of the underlying code. Because they are not known to the IDE, it is difficult for IDE features like code completion to understand them and use them to improve the developer experience.

Outside the Python ecosystem, static metaprogramming is also an important part of development, enabling the development of new programming paradigms and advanced libraries. There are many examples of prior art in this space, with different tradeoffs, for example:

1. Preprocessors (e.g. C preprocessor, Lex/YACC, etc) are perhaps the heaviest handed. They are fully general but the worst in terms of developer experience and tools integration.
2. Some languages (like Lisp and Rust) support (sometimes “hygienic”) macro expansion features, enabling syntactic extension and boilerplate reduction with somewhat better tooling integration.
3. Some older languages like C++ have very large and complex metaprogramming languages (templates) that are a dual to the *runtime* language. These are notably difficult to learn and have poor compile times and error messages.
4. Some languages (like Swift) build many features into the core language in a first-class way to provide good ergonomics for common cases at the expense of generality.
5. Some newer languages like Zig integrate a language interpreter into the compilation flow, and allow the interpreter to reflect over the AST as it is compiled. This allows many of the same features as a macro system with better extensibility and generality.

For Modular's work in AI, high-performance machine learning kernels, and accelerators, we need high abstraction capabilities provided by advanced metaprogramming systems. We needed high-level zero-cost abstractions, expressive libraries, and large-scale integration of multiple variants of algorithms. We want library developers to be able to extend the system, just like they do in Python, providing an extensible developer platform.

That said, we are not willing to sacrifice developer experience (including compile times and error messages) nor are we interested in building a parallel language ecosystem that is difficult to teach. We can learn from these previous systems but also have new technologies to build on top of, including MLIR and fine-grained language-integrated caching technologies.

As such, Mojo supports compile-time metaprogramming built into the compiler as a separate stage of compilation—after parsing, semantic analysis, and IR generation, but before lowering to

target-specific code. It uses the same host language for runtime programs as it does for metaprograms, and leverages MLIR to represent and evaluate these programs predictably.

Let's take a look at some simple examples.

**About “parameters”:** Python developers use the words “arguments” and “parameters” fairly interchangeably for “things that are passed into functions.” We decided to reclaim “parameter” and “parameter expression” to represent a compile-time value in Mojo, and continue to use “argument” and “expression” to refer to runtime values. This allows us to align around words like “parameterized” and “parametric” for compile-time metaprogramming.

## Defining parameterized types and functions

You can parameterize structs and functions by specifying parameter names and types in square brackets (using an extended version of the [PEP695 syntax](#)). Unlike argument values, parameter values are known at compile-time, which enables an additional level of abstraction and code reuse, plus compiler optimizations such as [autotuning](#).

For instance, let's look at a [SIMD](#) type, which represents a low-level vector register in hardware that holds multiple instances of a scalar data-type. Hardware accelerators are constantly introducing new vector data types, and even CPUs may have 512-bit or longer SIMD vectors. In order to access the SIMD instructions on these processors, the data must be shaped into the proper SIMD width (data type) and length (vector size).

However, it's not feasible to define all the different SIMD variations with Mojo's built-in types. So, Mojo's SIMD type (defined as a struct) exposes the common SIMD operations in its methods, and makes the SIMD data type and size values parametric. This allows you to directly map your data to the SIMD vectors on any hardware.

Here is a cut-down (non-functional) version of Mojo's SIMD type definition:

```
struct SIMD[type: DType, size: Int]:  
    var value: ... # Some low-level MLIR stuff here  
  
    # Create a new SIMD from a number of scalars  
    fn __init__(inout self, *elems: SIMD[type, 1]): ...  
  
    # Fill a SIMD with a duplicated scalar value.  
    @staticmethod  
    fn splat(x: SIMD[type, 1]) -> SIMD[type, size]: ...  
  
    # Cast the elements of the SIMD to a different elt type.  
    fn cast[target: DType](self) -> SIMD[target, size]: ...  
  
    # Many standard operators are supported.  
    fn __add__(self, rhs: Self) -> Self: ...
```

Defining each SIMD variant with parameters is great for code reuse because the SIMD type can express all the different vector variants statically, instead of requiring the language to pre-define every variant.

Because SIMD is a parameterized type, the self argument in its functions carries those parameters—the full type name is SIMD[type, size]. Although it's valid to write this out (as shown in the return type of `splat()`), this can be verbose, so we recommend using the `Self` type (from [PEP673](#)) like the `__add__` example does.

## Overloading on parameters

Functions and methods can be overloaded on their parameter signatures. The overload resolution logic filters for candidates according to the following rules, in order of precedence:

1. Candidates with the minimal number of implicit conversions (in both arguments and parameters).

2. Candidates without variadic arguments.
3. Candidates without variadic parameters.
4. Candidates with the shortest parameter signature.
5. Non-@staticmethod candidates (over @staticmethod ones, if available).

If there is more than one candidate after applying these rules, the overload resolution fails. For example:

```
@register_passable("trivial")
struct MyInt:
    """A type that is implicitly convertible to `Int`."""
    var value: Int

    @always_inline("nodebug")
    fn __init__(_a: Int) -> Self:
        return Self {value: _a}

fn foo[x: MyInt, a: Int]():
    print("foo[x: MyInt, a: Int]()")

fn foo[x: MyInt, y: MyInt]():
    print("foo[x: MyInt, y: MyInt]()")

fn bar[a: Int](b: Int):
    print("bar[a: Int](b: Int)")

fn bar[a: Int](*b: Int):
    print("bar[a: Int](*b: Int)")

fn bar[*a: Int](b: Int):
    print("bar[*a: Int](b: Int)")

fn parameter_overloads[a: Int, b: Int, x: MyInt]():
    # `foo[x: MyInt, a: Int]()` is called because it requires no implicit
    # conversions, whereas `foo[x: MyInt, y: MyInt]()` requires one.
    foo[x, a]()

    # `bar[a: Int](b: Int)` is called because it does not have variadic
    # arguments or parameters.
    bar[a](b)

    # `bar[*a: Int](b: Int)` is called because it has variadic parameters.
    bar[a, a, a](b)

parameter_overloads[1, 2, MyInt(3)]()

struct MyStruct:
    fn __init__(inout self):
        pass

    fn foo(inout self):
        print("calling instance method")

    @staticmethod
    fn foo():
        print("calling static method")

fn test_static_overload():
    var a = MyStruct()
    # `foo(inout self)` takes precedence over a static method.
    a.foo()
```

```
foo[x: MyInt, a: Int]()
bar[a: Int](b: Int)
bar[*a: Int](b: Int)
```

## Using parameterized types and functions

You can instantiate parametric types and functions by passing values to the parameters in square brackets. For example, for the SIMD type above, `type` specifies the data type and `size` specifies the length of the SIMD vector (it must be a power of 2):

```

# Make a vector of 4 floats.
let small_vec = SIMD[DType.float32, 4](1.0, 2.0, 3.0, 4.0)

# Make a big vector containing 1.0 in float16 format.
let big_vec = SIMD[DType.float16, 32].splat(1.0)

# Do some math and convert the elements to float32.
let bigger_vec = (big_vec+big_vec).cast[DType.float32]()

# You can write types out explicitly if you want of course.
let bigger_vec2 : SIMD[DType.float32, 32] = bigger_vec

print('small_vec type:', small_vec.element_type, 'length:', len(small_vec))
print('bigger_vec2 type:', bigger_vec2.element_type, 'length:', len(bigger_vec2))
```

small\_vec type: float32 length: 4  
bigger\_vec2 type: float32 length: 32

Note that the `cast()` method also needs a parameter to specify the type you want from the `cast` (the method definition above expects a target parametric value). Thus, just like how the `SIMD` struct is a generic type definition, the `cast()` method is a generic method definition that gets instantiated at compile-time instead of runtime, based on the parameter value.

The code above shows the use of concrete types (that is, it instantiates `SIMD` using known type values), but the major power of parameters comes from the ability to define parametric algorithms and types (code that uses the parameter values). For example, here's how to define a parametric algorithm with `SIMD` that is type- and width-agnostic:

```

from math import sqrt

fn rsqrt[dt: DType, width: Int](x: SIMD[dt, width]) -> SIMD[dt, width]:
    return 1 / sqrt(x)

print(rsqrt[DType.float16, 4](42))
```

[0.154296875, 0.154296875, 0.154296875, 0.154296875]

Notice that the `x` argument is actually a `SIMD` type based on the function parameters. The runtime program can use the value of parameters, because the parameters are resolved at compile-time before they are needed by the runtime program (but compile-time parameter expressions cannot use runtime values).

The Mojo compiler is also smart about type inference with parameters. Note that the above function is able to call the parametric `sqrt[1]()` function without specifying the parameters—the compiler infers its parameters based on the parametric `x` value passed into it, as if you wrote `sqrt[dt, width](x)` explicitly. Also note that `rsqrt()` chose to define its first parameter named `width` even though the `SIMD` type names it `size`, and there is no problem.

## Using default parameter values

Just like how you can specify [default argument values](#) (in function arguments), you can also specify default values for parameters (in both function and struct parameters).

For example, here's a function with two parameters, each with a default value:

```

fn foo[a: Int = 3, msg: StringLiteral = "woof"]():
    print(msg, a)

fn use_defaults():
    foo()           # prints 'woof 3'
    foo[5]()        # prints 'woof 5'
    foo[7, "meow"]() # prints 'meow 7'
```

Recall that Mojo can infer parameter values in a parametric function, based on the parametric values attached to an argument value (see the `rsqrt[]()` example above). If the parametric function also has a default value defined, then the inferred parameter type takes precedence.

For example, in the following code, we update the parametric `foo[]()` function to take an argument with a parametric type. Although the function has a default parameter value for `a`, Mojo instead uses the inferred a parameter value from the `bar` argument (as written, the default a value can never be used, but this is just for demonstration purposes):

```
@value
struct Bar[v: Int]:
    pass

fn foo[a: Int = 3, msg: StringLiteral = "woof"](bar: Bar[a]):
    print(msg, a)

fn use_inferred():
    foo(Bar[9]()) # prints 'woof 9' □
```

And here's an example of default parameters in a struct:

```
@value
struct DefaultParams[msg: StringLiteral = "woof"]:
    alias message = msg

fn use_struct_default():
    print(DefaultParams[]().message)      # prints 'woof'
    print(DefaultParams["meow"]().message) # prints 'meow' □
```

## Parameter expressions are just Mojo code

A parameter expression is any code expression (such as `a+b`) that occurs where a parameter is expected. Parameter expressions support operators and function calls, just like runtime code, and all parameter types use the same type system as the runtime program (such as `Int` and `DType`).

Because parameter expressions use the same grammar and types as runtime Mojo code, you can use many “dependent type” features. For example, you might want to define a helper function to concatenate two SIMD vectors:

```
fn concat[ty: DType, len1: Int, len2: Int](
    lhs: SIMD[ty, len1], rhs: SIMD[ty, len2]) -> SIMD[ty, len1+len2]:

    var result = SIMD[ty, len1 + len2]()
    for i in range(len1):
        result[i] = SIMD[ty, 1](lhs[i])
    for j in range(len2):
        result[len1 + j] = SIMD[ty, 1](rhs[j])
    return result

let a = SIMD[DType.float32, 2](1, 2)
let x = concat[DType.float32, 2, 2](a, a)

print('result type:', x.element_type, 'length:', len(x)) □
result type: float32 length: 4
```

Note how the resulting length is the sum of the input vector lengths, and you can express that with a simple `+` operation. For a more complex example, take a look at the [SIMD.shuffle\(\)](#) method in the standard library: it takes two input SIMD values, a vector shuffle mask as a list, and returns a SIMD that matches the length of the shuffle mask.

## Powerful compile-time programming

While simple expressions are useful, sometimes you want to write imperative compile-time logic with control flow. For example, the `isclose()` function in the Mojo Math module uses exact equality for integers but “close” comparison for floating-point. You can even do compile-time recursion. For instance, here is an example “tree reduction” algorithm that sums all elements of a vector recursively into a scalar:

```

fn slice[ty: DType, new_size: Int, size: Int]{
    x: SIMD[ty, size], offset: Int) -> SIMD[ty, new_size]:
    var result = SIMD[ty, new_size]()
    for i in range(new_size):
        result[i] = SIMD[ty, 1](x[i + offset])
    return result

fn reduce_add[ty: DType, size: Int](x: SIMD[ty, size]) -> Int:
    @parameter
    if size == 1:
        return x[0].to_int()
    elif size == 2:
        return x[0].to_int() + x[1].to_int()

    # Extract the top/bottom halves, add them, sum the elements.
    alias half_size = size // 2
    let lhs = slice[ty, half_size, size](x, 0)
    let rhs = slice[ty, half_size, size](x, half_size)
    return reduce_add[ty, half_size](lhs + rhs)

let x = SIMD[DType.index, 4](1, 2, 3, 4)
print(x)
print("Elements sum:", reduce_add[DType.index, 4](x))
```

[1, 2, 3, 4]  
Elements sum: 10

This makes use of the `@parameter if` feature, which is an `if` statement that runs at compile-time. It requires that its condition be a valid parameter expression, and ensures that only the live branch of the `if` statement is compiled into the program.

## Mojo types are just parameter expressions

While we've shown how you can use parameter expressions within types, type annotations can themselves be arbitrary expressions (just like in Python). Types in Mojo have a special metatype type, allowing type-parametric algorithms and functions to be defined.

For example, we can create a simplified `Array` that supports arbitrary types of the elements (via the `AnyType` parameter):

```

struct Array[T: AnyType]:
    var data: Pointer[T]
    var size: Int

    fn __init__(inout self, size: Int, value: T):
        self.size = size
        self.data = Pointer[T].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, value)

    fn __getitem__(self, i: Int) -> T:
        return self.data.load(i)

    fn __del__(owned self):
        self.data.free()

var v = Array[Float32](4, 3.14)
print(v[0], v[1], v[2], v[3])
```

3.1400001049041748 3.1400001049041748 3.1400001049041748 3.1400001049041748

Notice that the `T` parameter is being used as the formal type for the `value` arguments and the return type of the `__getitem__` function. Parameters allow the `Array` type to provide different APIs based on the different use-cases.

There are many other cases that benefit from more advanced use of parameters. For example, you can execute a closure `N` times in parallel, feeding in a value from the context, like this:

```

fn parallelize[func: fn (Int) -> None](num_work_items: Int):
    # Not actually parallel: see the 'algorithm' module for real implementation.
```

```
for i in range(num_work_items):
    func(i)
```

Another example where this is important is with variadic generics, where an algorithm or data structure may need to be defined over a list of heterogeneous types such as for a tuple:

```
struct Tuple[*Ts: AnyType]:
    var _storage : *Ts
```

And although we don't have enough metatype helpers in place yet, we should be able to write something like this in the future (though overloading is still a better way to handle this):

```
struct Array[T: AnyType]:
    fn __getitem__[IndexType: AnyType](self, idx: IndexType)
        -> (ArraySlice[T] if issubclass(IndexType, Range) else T):
    ...
```

## alias: named parameter expressions

It is very common to want to *name* compile-time values. Whereas `var` defines a runtime value, and `let` defines a runtime constant, we need a way to define a compile-time temporary value. For this, Mojo uses an `alias` declaration.

For example, the `DTType` struct implements a simple enum using aliases for the enumerators like this (the actual `DTType` implementation details vary a bit):

```
struct DTType:
    var value : UI8
    alias invalid = DTType(0)
    alias bool = DTType(1)
    alias int8 = DTType(2)
    alias uint8 = DTType(3)
    alias int16 = DTType(4)
    alias int16 = DTType(5)
    ...
    alias float32 = DTType(15)
```

This allows clients to use `DTType.float32` as a parameter expression (which also works as a runtime value) naturally. Note that this is invoking the runtime constructor for `DTType` at compile-time.

Types are another common use for `alias`. Because types are compile-time expressions, it is handy to be able to do things like this:

```
alias Float16 = SIMD[DTType.float16, 1]
alias UInt8 = SIMD[DTType.uint8, 1]

var x : Float16 # FLoat16 works like a "typedef"
```

Like `var` and `let`, aliases obey scope, and you can use local aliases within functions as you'd expect.

By the way, both `None` and `AnyType` are defined as [type aliases](#).

## Autotuning / Adaptive compilation

Mojo parameter expressions allow you to write portable parametric algorithms like you can do in other languages, but when writing high-performance code you still have to pick concrete values to use for the parameters. For example, when writing high-performance numeric algorithms, you might want to use memory tiling to accelerate the algorithm, but the dimensions to use depend highly on the available hardware features, the sizes of the cache, what gets fused into the kernel, and many other fiddly details.

Even vector length can be difficult to manage, because the vector length of a typical machine depends on the datatype, and some datatypes like `bfloat16` don't have full support on all implementations. Mojo helps by providing an `autotune` function in the standard library. For

example if you want to write a vector-length-agnostic algorithm to a buffer of data, you might write it like this:

```
from autotune import autotune, search
from benchmark import Benchmark
from memory.unsafe import DTypePointer
from algorithm import vectorize

fn buffer_elementwise_add_impl[
    dt: DType
](lhs: DTypePointer[dt], rhs: DTypePointer[dt], result: DTypePointer[dt], N: Int):
    """Perform elementwise addition of N elements in RHS and LHS and store
    the result in RESULT.
    """
    @parameter
    fn add_simd[size: Int](idx: Int):
        let lhs_simd = lhssimd_load[size](idx)
        let rhs_simd = rhssimd_load[size](idx)
        resultsimd_store[size](idx, lhs_simd + rhs_simd)

    # Pick vector length for this dtype and hardware
    alias vector_len = autotune(1, 4, 8, 16, 32)

    # Use it as the vectorization length
    vectorize[vector_len, add_simd](N)

fn elementwise_evaluator[dt: DType](
    fns: Pointer[fn (DTypePointer[dt], DTypePointer[dt], DTypePointer[dt], Int) -> None],
    num: Int,
) -> Int:
    # Benchmark the implementations on N = 64.
    alias N = 64
    let lhs = DTypePointer[dt].alloc(N)
    let rhs = DTypePointer[dt].alloc(N)
    let result = DTypePointer[dt].alloc(N)

    # Fill with ones.
    for i in range(N):
        lhs.store(i, 1)
        rhs.store(i, 1)

    # Find the fastest implementation.
    var best_idx: Int = -1
    var best_time: Int = -1
    for i in range(num):
        @parameter
        fn wrapper():
            fns.load(i)(lhs, rhs, result, N)
        let cur_time = Benchmark(1).run[wrapper]()
        if best_idx < 0 or best_time > cur_time:
            best_idx = i
            best_time = cur_time
        print("time[", i, "] =", cur_time)
    print("selected:", best_idx)
    return best_idx

fn buffer_elementwise_add[
    dt: DType
](lhs: DTypePointer[dt], rhs: DTypePointer[dt], result: DTypePointer[dt], N: Int):
    # Forward declare the result parameter.
    alias best_impl: fn(DTypePointer[dt], DTypePointer[dt], DTypePointer[dt], Int) -> None

    # Perform search!
    search[
        fn(DTypePointer[dt], DTypePointer[dt], DTypePointer[dt], Int) -> None,
        buffer_elementwise_add_impl[dt],
        elementwise_evaluator[dt] -> best_impl
    ]()

    # Call the select implementation
    best_impl(lhs, rhs, result, N)
```

We can now call our function as usual:

```

let N = 32
let a = DTypePointer[DType.float32].alloc(N)
let b = DTypePointer[DType.float32].alloc(N)
let res = DTypePointer[DType.float32].alloc(N)
# Initialize arrays with some values
for i in range(N):
    a.store(i, 2.0)
    b.store(i, 40.0)
    res.store(i, -1)

buffer_elementwise_add[DType.float32](a, b, res, N)
print(a.load(10), b.load(10), res.load(10))
```

```

time[ 0 ] = 23
time[ 1 ] = 6
time[ 2 ] = 4
time[ 3 ] = 3
time[ 4 ] = 4
selected: 3
2.0 40.0 42.0
```

When compiling instantiations of this code, Mojo forks compilation of this algorithm and decides which value to use by measuring what works best in practice for the target hardware. It evaluates the different values of the `vector_len` expression and picks the fastest one according to a user-defined performance evaluator. Because it measures and evaluates each option individually, it might pick a different vector length for `float32` than for `int8`, for example. This simple feature is pretty powerful—going beyond simple integer constants—because functions and types are also parameter expressions.

Notice that the search for the best vector length is performed by the [search\(\)](#) function. `search()` takes an evaluator and a forked function and returns the fastest implementation selected by the evaluator as a parameter result. For a deeper dive on this topic, check out the notebooks about [Matrix Multiplication](#) and [Fast Memset in Mojo](#).

Autotuning is an inherently exponential technique that benefits from internal implementation details of the Mojo compiler stack (particularly MLIR, integrated caching, and distribution of compilation). This is also a power-user feature and needs continued development and iteration over time.

## “Value Lifecycle”: Birth, life and death of a value

At this point, you should understand the core semantics and features for Mojo functions and types, so we can now discuss how they fit together to express new types in Mojo.

Many existing languages express design points with different tradeoffs: C++, for example, is very powerful but often accused of “getting the defaults wrong” which leads to bugs and mis-features. Swift is easy to work with, but has a less predictable model that copies values a lot and is dependent on an “ARC optimizer” for performance. Rust started with strong value ownership goals to satisfy its borrow checker, but relies on values being movable, which makes it challenging to express custom move constructors and can put a lot of stress on `memcpy` performance. In Python, everything is a reference to a class, so it never really faces issues with types.

For Mojo, we’ve learned from these existing systems, and we aim to provide a model that’s very powerful while still easy to learn and understand. We also don’t want to require “best effort” and difficult-to-predict optimization passes built into a “sufficiently smart” compiler.

To explore these issues, we look at different value classifications and the relevant Mojo features that go into expressing them, and build from the bottom-up. We use C++ as the primary comparison point in examples because it is widely known, but we occasionally reference other languages if they provide a better comparison point.

## Types that cannot be instantiated

The most bare-bones type in Mojo is one that doesn't allow you to create instances of it: these types have no initializer at all, and if they have a destructor, it will never be invoked (because there cannot be instances to destroy):

```
struct NoInstances:  
    var state: Int # Pretty useless  
  
    alias my_int = Int  
  
    @staticmethod  
    fn print_hello():  
        print("hello world")
```

Mojo types do not get default constructors, move constructors, member-wise initializers or anything else by default, so it is impossible to create an instance of this `NoInstances` type. In order to get them, you need to define an `__init__` method or use a decorator that synthesizes an initializer. As shown, these types can be useful as "namespaces" because you can refer to static members like `NoInstances.my_int` or `NoInstances.print_hello()` even though you cannot instantiate an instance of the type.

## Non-movable and non-copyable types

If we take a step up the ladder of sophistication, we'll get to types that can be instantiated, but once they are pinned to an address in memory, they cannot be implicitly moved or copied. This can be useful to implement types like atomic operations (such as `std::atomic` in C++) or other types where the memory address of the value is its identity and is critical to its purpose:

```
struct Atomic:  
    var state: Int  
  
    fn __init__(inout self, state: Int = 0):  
        self.state = state  
  
    fn __iadd__(inout self, rhs: Int):  
        #...atomic magic...  
  
    fn get_value(self) -> Int:  
        return atomic_load_int(self.state)
```

This class defines an initializer but no copy or move constructors, so once it is initialized it can never be moved or copied. This is safe and useful because Mojo's ownership system is fully "address correct" - when this is initialized onto the stack or in the field of some other type, it never needs to move.

Note that Mojo's approach controls only the built-in move operations, such as `a = b` copies and the [^transfer operator](#). One useful pattern you can use for your own types (like `Atomic` above) is to add an explicit `copy()` method (a non-“dunder” method). This can be useful to make explicit copies of an instance when it is known safe to the programmer.

## Unique “move-only” types

If we take one more step up the ladder of capabilities, we will encounter types that are "unique" - there are many examples of this in C++, such as types like `std::unique_ptr` or even a `FileDescriptor` type that owns an underlying POSIX file descriptor. These types are pervasive in languages like Rust, where copying is discouraged, but "move" is free. In Mojo, you can implement these kinds of moves by defining the `__moveinit__` method to take ownership of a unique type. For example:

```
# This is a simple wrapper around POSIX-style fcntl.h functions.  
struct FileDescriptor:  
    var fd: Int  
  
    # This is how we move our unique type.  
    fn __moveinit__(inout self, owned existing: Self):  
        self.fd = existing.fd
```

```

# This takes ownership of a POSIX file descriptor.
fn __init__(inout self, fd: Int):
    self.fd = fd

fn __init__(inout self, path: String):
    # Error handling omitted, call the open(2) syscall.
    self = FileDescriptor(open(path, ...))

fn __del__(owned self):
    close(self.fd)  # pseudo code, call close(2)

fn dup(self) -> Self:
    # Invoke the dup(2) system call.
    return Self(dup(self.fd))

fn read(...): ...
fn write(...): ..._
```

The consuming move constructor (`__moveinit__`) takes ownership of an existing `FileDescriptor`, and moves its internal implementation details over to a new instance. This is because instances of `FileDescriptor` may exist at different locations, and they can be logically moved around—stealing the body of one value and moving it into another.

Here is an egregious example that will invoke `__moveinit__` multiple times:

```

fn egregious_moves(owned fd1: FileDescriptor):
    # fd1 and fd2 have different addresses in memory, but the
    # transfer operator moves unique ownership from fd1 to fd2.
    let fd2 = fd1^

    # Do it again, a use of fd2 after this point will produce an error.
    let fd3 = fd2^

    # We can do this all day...
    let fd4 = fd3^
    fd4.read(...)
    # fd4.__del__() runs here_
```

Note how ownership of the value is transferred between various values that own it, using the postfix-`^` “transfer” operator, which destroys a previous binding and transfer ownership to a new constant. If you are familiar with C++, the simple way to think about the transfer operator is like `std::move`, but in this case, we can see that it is able to move things without resetting them to a state that can be destroyed: in C++, if your move operator failed to change the old value’s `fd` instance, it would get closed twice.

Mojo tracks the liveness of values and allows you to define custom move constructors. This is rarely needed, but extremely powerful when it is. For example, some types like the [llvm::SmallVector\\_type](#) use the “inline storage” optimization technique, and they may want to be implemented with an “inner pointer” into their instance. This is a well-known trick to reduce pressure on the malloc memory allocator, but it means that a “move” operation needs custom logic to update the pointer when that happens.

With Mojo, this is as simple as implementing a custom `__moveinit__` method. This is something that is also easy to implement in C++ (though, with boilerplate in the cases where you don’t need custom logic) but is difficult to implement in other popular memory-safe languages.

One additional note is that while the Mojo compiler provides good predictability and control, it is also very sophisticated. It reserves the right to eliminate temporaries and the corresponding copy/move operations. If this is inappropriate for your type, you should use explicit methods like `copy()` instead of the dunder methods.

## Types that support a “taking move”

One challenge with memory-safe languages is that they need to provide a predictable programming model around what the compiler is able to track, and static analysis in a compiler is inherently limited. For example, while it is possible for a compiler to understand that the two

array accesses in the first example below are to different array elements, it is (in general) impossible to reason about the second example (this is C++ code):

```
std::pair<T, T> getValues1(MutableArray<T> &array) {
    return { std::move(array[0]), std::move(array[1]) };
}
std::pair<T, T> getValues2(MutableArray<T> &array, size_t i, size_t j) {
    return { std::move(array[i]), std::move(array[j]) };
}
```

The problem here is that there is simply no way (looking at just the function body above) to know or prove that the dynamic values of *i* and *j* are not the same. While it is possible to maintain dynamic state to track whether individual elements of the array are live, this often causes significant runtime expense (even when move/transfers are not used), which is something that Mojo and other systems programming languages are not keen to do. There are a variety of ways to deal with this, including some pretty complicated solutions that aren't always easy to learn.

Mojo takes a pragmatic approach to let Mojo programmers get their job done without having to work around its type system. As seen above, it doesn't force types to be copyable, movable, or even constructable, but it does want types to express their full contract, and it wants to enable fluent design patterns that programmers expect from languages like C++. The (well known) observation here is that many objects have contents that can be "taken away" without needing to disable their destructor, either because they have a "null state" (like an optional type or nullable pointer) or because they have a null value that is efficient to create and a no-op to destroy (e.g. `std::vector` can have a null pointer for its data).

To support these use-cases, the [^transfer operator](#) supports arbitrary LValues, and when applied to one, it invokes the "taking move constructor," which is spelled `_takeinit_`. This constructor must set up the new value to be in a live state, and it can mutate the old value, but it must put the old value into a state where its destructor still works. For example, if we want to put our `FileDescriptor` into a vector and move out of it, we might choose to extend it to know that -1 is a sentinel which means that it is "null". We can implement this like so:

```
# This is a simple wrapper around POSIX-style fcntl.h functions.
struct FileDescriptor:
    var fd: Int

    # This is the new key capability.
    fn __takeinit__(inout self, inout existing: Self):
        self.fd = existing.fd
        existing.fd = -1 # neutralize 'existing'.

    fn __moveinit__(inout self, owned existing: Self): # as above
    fn __init__(inout self, fd: Int): # as above
    fn __init__(inout self, path: String): # as above

    fn __del__(owned self):
        if self.fd != -1:
            close(self.fd) # pseudo code, call close(2)
```

Notice how the "stealing move" constructor takes the file descriptor from an existing value and mutates that value so that its destructor won't do anything. This technique has tradeoffs and is not the best for every type. We can see that it adds one (inexpensive) branch to the destructor because it has to check for the sentinel case. It is also generally considered bad form to make types like this nullable because a more general feature like an `Optional[T]` type is a better way to handle this.

Furthermore, we plan to implement `Optional[T]` in Mojo itself, and `Optional` needs this functionality. We also believe that the library authors understand their domain problem better than language designers do, and generally prefer to give library authors full power over that domain. As such you can choose (but don't have to) to make your types participate in this behavior in an opt-in way.

## Copyable types

The next step up from movable types are copyable types. Copyable types are also very common - programmers generally expect things like strings and arrays to be copyable, and every Python Object reference is copyable - by copying the pointer and adjusting the reference count.

There are many ways to implement copyable types. One can implement reference semantic types like Python or Java, where you propagate shared pointers around, one can use immutable data structures that are easily shareable because they are never mutated once created, and one can implement deep value semantics through lazy copy-on-write as Swift does. Each of these approaches has different tradeoffs, and Mojo takes the opinion that while we want a few common sets of collection types, we can also support a wide range of specialized ones that focus on particular use cases.

In Mojo, you can do this by implementing the `__copyinit__` method. Here is an example of that using a simple `String` (in pseudo-code):

```
struct MyString:  
    var data: Pointer[UI8]  
  
    # StringRef is a pointer + length and works with StringLiteral.  
    def __init__(inout self, input: StringRef):  
        self.data = ...  
  
    # Copy the string by deep copying the underlying malloc'd data.  
    def __copyinit__(inout self, existing: Self):  
        self.data = strdup(existing.data)  
  
    # This isn't required, but optimizes unneeded copies.  
    def __moveinit__(inout self, owned existing: Self):  
        self.data = existing.data  
  
    def __del__(owned self):  
        free(self.data.address)  
  
    def __add__(self, rhs: MyString) -> MyString: ...
```

This simple type is a pointer to a “null-terminated” string data allocated with `malloc`, using old-school C APIs for clarity. It implements the `__copyinit__`, which maintains the invariant that each instance of `MyString` owns its underlying pointer and frees it upon destruction. This implementation builds on tricks we’ve seen above, and implements a `__moveinit__` constructor, which allows it to completely eliminate temporary copies in some common cases. You can see this behavior in this code sequence:

```
fn test_my_string():  
    var s1 = MyString("hello ")  
  
    var s2 = s1      # s2.__copyinit__(s1) runs here  
  
    print(s1)  
  
    var s3 = s1^    # s3.__moveinit__(s1) runs here  
  
    print(s2)  
    # s2.__del__() runs here  
    print(s3)  
    # s3.__del__() runs here
```

In this case, you can see both why a copy constructor is needed: without one, the duplication of the `s1` value into `s2` would be an error - because you cannot have two live instances of the same non-copyable type. The move constructor is optional but helps the assignment into `s3`: without it, the compiler would invoke the copy constructor from `s1`, then destroy the old `s1` instance. This is logically correct but introduces extra runtime overhead.

Mojo destroys values eagerly, which allows it to transform copy+destroy pairs into single move operations, which can lead to much better performance than C++ without requiring the need for pervasive micromanagement of `std::move`.

# Trivial types

The most flexible types are ones that are just “bags of bits”. These types are “trivial” because they can be copied, moved, and destroyed without invoking custom code. Types like these are arguably the most common basic type that surrounds us: things like integers and floating point values are all trivial. From a language perspective, Mojo doesn’t need special support for these, it would be perfectly fine for type authors to implement these things as no-ops, and allow the inliner to just make them go away.

There are two reasons that approach would be suboptimal: one is that we don’t want the boilerplate of having to define a bunch of methods on trivial types, and second, we don’t want the compile-time overhead of generating and pushing around a bunch of function calls, only to have them inline away to nothing. Furthermore, there is an orthogonal concern, which is that many of these types are trivial in another way: they are tiny, and should be passed around in the registers of a CPU, not indirectly in memory.

As such, Mojo provides a struct decorator that solves all of these problems. You can implement a type with the `@register_passable("trivial")` decorator, and this tells Mojo that the type should be copyable and movable but that it has no user-defined logic for doing this. It also tells Mojo to prefer to pass the value in CPU registers, which can lead to efficiency benefits.

TODO: This decorator is due for reconsideration. Lack of custom logic copy/move/destroy logic and “passability in a register” are orthogonal concerns and should be split. This former logic should be subsumed into a more general `@value("trivial")` decorator, which is orthogonal from `@register_passable`.

## `@value` decorator

Mojo’s [value lifecycle](#) provides simple and predictable hooks that give you the ability to express exotic low-level things like Atomic correctly. This is great for control and for a simple programming model, but most structs are simple aggregations of other types, and we don’t want to write a lot of boilerplate for them. To solve this, Mojo provides a `@value` decorator for structs that synthesizes a lot of boilerplate for you.

You can think of `@value` as an extension of Python’s [@dataclass](#) that also handles Mojo’s `__moveinit__` and `__copyinit__` methods.

The `@value` decorator takes a look at the fields of your type, and generates some members that are missing. For example, consider a simple struct like this:

```
@value
struct MyPet:
    var name: String
    var age: Int
```

Mojo will notice that you do not have a member-wise initializer, a move constructor, or a copy constructor, and it will synthesize these for you as if you had written:

```
struct MyPet:
    var name: String
    var age: Int

fn __init__(inout self, owned name: String, age: Int):
    self.name = name^
    self.age = age

fn __copyinit__(inout self, existing: Self):
    self.name = existing.name
    self.age = existing.age

fn __moveinit__(inout self, owned existing: Self):
    self.name = existing.name^
    self.age = existing.age
```

When you add the `@value` decorator, Mojo synthesizes each of these special methods only when it doesn't exist. You can override the behavior of one or more by defining your own version. For example, it is fairly common to want a custom copy constructor but use the default member-wise and move constructor.

The arguments to `__init__` are all passed as owned arguments since the struct takes ownership and stores the value. This is a useful micro-optimization and enables the use of move-only types. Trivial types like `Int` are also passed as owned values, but since that doesn't mean anything for them, we elide the marker and the transfer operator (^) for clarity.

**Note:** If your type contains any [move-only](#) fields, Mojo will not generate a copy constructor because it cannot copy those fields. Further, the `@value` decorator only works on types whose members are copyable and/or movable. If you have something like `Atomic` in your struct, then it probably isn't a value type, and you don't want these members anyway.

Also notice that the `MyPet` struct above doesn't include the `__del__()` destructor—Mojo also synthesizes this, but it doesn't require the `@value` decorator (see the section below about [destructors](#)).

There is no way to suppress the generation of specific methods or customize generation at this time, but we can add arguments to the `@value` generator to do this if there is demand.

## Behavior of destructors

Any struct in Mojo can have a destructor (a `__del__()` method), which is automatically run when the value's lifetime ends (typically the point at which the value is last used). For example, a simple string might look like this (in pseudo code):

```
@value
struct MyString:
    var data: Pointer[UInt8]

    def __init__(inout self, input: StringRef): ...
    def __add__(self, rhs: String) -> MyString: ...
    def __del__(owned self):
        free(self.data.address)
```

Mojo destroys values like `MyString` (it calls the `__del__()` destructor) using an **“As Soon As Possible”** (ASAP) policy that runs after every call. Mojo does *not* wait until the end of the code block to destroy unused values. Even in an expression like `a+b+c+d`, Mojo destroys the intermediate expressions eagerly, as soon as they are no longer needed—it does not wait until the end of the statement.

The Mojo compiler automatically invokes the destructor when a value is dead and provides strong guarantees about when the destructor is run. Mojo uses static compiler analysis to reason about your code and decide when to insert calls to the destructor. For example:

```
fn use_strings():
    var a = String("hello a")
    let b = String("hello b")
    print(a)
    # a.__del__() runs here for "hello a"

    print(b)
    # b.__del__() runs here

    a = String("temporary a")
    # a.__del__() runs here because "temporary a" is never used

    # Other stuff happens here

    a = String("final a")
    print(a)
    # a.__del__() runs again here for "final a"
```

```
use_strings() →
```

```
hello a  
hello b  
final a
```

In the code above, you'll see that the `a` and `b` values are created early on, and each initialization of a value is matched with a call to a destructor. Notice that `a` is destroyed multiple times—once for each time it receives a new value.

Now, this might be surprising to a C++ programmer, because it's different from the [RAII pattern](#) in which C++ destroys values at the end of a scope. Mojo also follows the principle that values acquire resources in a constructor and release resources in a destructor, but eager destruction in Mojo has a number of strong advantages over scope-based destruction in C++:

- The Mojo approach eliminates the need for types to implement re-assignment operators, like `operator=(const T&)` and `operator=(T&&)` in C++, making it easier to define types and eliminating a concept.
- Mojo does not allow mutable references to overlap with other mutable references or with immutable borrows. One major way that it provides a predictable programming model is by making sure that references to objects die as soon as possible, avoiding confusing situations where the compiler thinks a value could still be alive and interfere with another value, but that isn't clear to the user.
- Destroying values at last-use composes nicely with “move” optimization, which transforms a “copy+del” pair into a “move” operation, a generalization of C++ move optimizations like NRVO (named return value optimization).
- Destroying values at end-of-scope in C++ is problematic for some common patterns like tail recursion because the destructor calls happen after the tail call. This can be a significant performance and memory problem for certain functional programming patterns.

Importantly, Mojo's eager destruction also works well within Python-style `def` functions to provide destruction guarantees (without a garbage collector) at a fine-grain level—recall that Python doesn't really provide scopes beyond a function, so C++-style destruction in Mojo would be a lot less useful.

**Note:** Mojo also supports the Python-style [with statement](#), which provides more deliberately-scoped access to resources.

The Mojo approach is more similar to how Rust and Swift work, because they both have strong value ownership tracking and provide memory safety. One difference is that their implementations require the use of a [dynamic “drop flag”](#)—they maintain hidden shadow variables to keep track of the state of your values to provide safety. These are often optimized away, but the Mojo approach eliminates this overhead entirely, making the generated code faster and avoiding ambiguity.

## Field-sensitive lifetime management

In addition to Mojo's lifetime analysis being fully control-flow aware, it is also fully field-sensitive (each field of a structure is tracked independently). That is, Mojo separately keeps track of whether a “whole object” is fully or only partially initialized/destroyed.

For example, consider this code:

```
@value  
struct TwoStrings:  
    var str1: String  
    var str2: String  
  
fn use_two_strings():
```

```
var ts = TwoStrings("foo", "bar")
print(ts.str1)
# ts.str1.__del__() runs here

# Other stuff happens here

ts.str1 = String("hello") # Overwrite ts.str1
print(ts.str1)
# ts.__del__() runs here
```

use\_two\_strings() 

```
foo
hello
```

Note that the `ts.str1` field is destroyed almost immediately, because Mojo knows that it will be overwritten down below. You can also see this when using the [transfer operator](#), for example:

```
fn consume(owned arg: String):
    pass

fn use(arg: TwoStrings):
    print(arg.str1)

fn consume_and_use_two_strings():
    var ts = TwoStrings("foo", "bar")
    consume(ts.str1^)
    # ts.str1.__moveinit__() runs here

    # ts is now only partially initialized here!

    ts.str1 = String("hello") # All together now
    use(ts)                 # This is ok
    # ts.__del__() runs here
```

consume\_and\_use\_two\_strings() 

```
hello
```

Notice that the code transfers ownership of the `str1` field: for the duration of `other_stuff()`, the `str1` field is completely uninitialized because ownership was transferred to `consume()`. Then `str1` is reinitialized before it is used by the `use()` function (if it weren't, Mojo would reject the code with an uninitialized field error).

Mojo's rule on this is powerful and intentionally straight-forward: fields can be temporarily transferred, but the "whole object" must be constructed with the aggregate type's initializer and destroyed with the aggregate destructor. This means that it isn't possible to create an object by initializing only its fields, nor is it possible to tear down an object by destroying only its fields. For example, this code does not compile:

```
fn consume_and_use_two_strings():
    let ts = TwoStrings("foo", "bar") # ts is initialized
    # Uncomment to see an error:
    # consume(ts.str1^)
    # Because `ts` is not used anymore, it should be destroyed here, but
    # the object is not whole, preventing the overall value from being destroyed

    let ts2 : TwoStrings # ts2 type is declared but not initialized
    ts2.str1 = String("foo")
    ts2.str2 = String("bar") # Both the member are initialized
    # Uncomment to see an error:
    # use(ts2) # Error: 'ts2' isn't fully initialized 
```

While we could allow patterns like this to happen, we reject this because a value is more than a sum of its parts. Consider a `FileDescriptor` that contains a POSIX file descriptor as an integer value: there is a big difference between destroying the integer (a no-op) and destroying the `FileDescriptor` (it might call the `close()` system call). Because of this, we require all full-value initialization to go through initializers and be destroyed with their full-value destructor.

For what it's worth, Mojo does internally have an equivalent of the Rust [mem::forget](#) function, which explicitly disables a destructor and has a corresponding internal feature for "blessing" an object, but they aren't exposed for user consumption at this point.

## Field lifetimes in `_init_`

The behavior of an `_init_` method works almost like any other method—there is a small bit of magic: it knows that the fields of an object are uninitialized, but it believes the full object is initialized. This means that you can use `self` as a whole object as soon as all the fields are initialized:

```
fn use(arg: TwoStrings2):
    pass

struct TwoStrings2:
    var str1: String
    var str2: String

    fn __init__(inout self, cond: Bool, other: String):
        self.str1 = String()
        if cond:
            self.str2 = other
            use(self) # Safe to use immediately!
            # self.str2.__del__(): destroyed because overwritten below.

        self.str2 = self.str1
        use(self) # Safe to use immediately! □
```

Similarly, it's safe for initializers in Mojo to completely overwrite `self`, such as by delegating to other initializers:

```
struct TwoStrings3:
    var str1: String
    var str2: String

    fn __init__(inout self):
        self.str1 = String()
        self.str2 = String()

    fn __init__(inout self, one: String):
        self = TwoStrings3() # Delegate to the basic init
        self.str1 = one □
```

## Field lifetimes of owned arguments in `_moveinit_` and `_del_`

A final bit of magic exists for the `owned` arguments of a `_moveinit_()` move initializer and a `_del_()` destructor. To recap, these method signatures look like this:

```
struct TwoStrings:
    ...
    fn __moveinit__(inout self, owned existing: Self):
        # Initializes a new `self` by consuming the contents of `existing`
    fn __del__(owned self):
        # Destroys all resources in `self` □
```

These methods face an interesting but obscure problem: both methods are in charge of dismantling the `owned` `existing`/`self` value. That is, `_moveinit_()` destroys sub-elements of `existing` in order to transfer ownership to a new instance, while `_del_()` implements the deletion logic for its `self`. As such, they both want to own and transform elements of the `owned` value, and they definitely don't want the `owned` value's destructor to also run (in the case of the `_del_()` method, that would turn into an infinite loop).

To solve this problem, Mojo handles these two methods specially by assuming that their whole values are destroyed upon reaching any return from the method. This means that the whole object may be used before the field values are transferred. For example, this works as you expect:

```

fn consume(owned str: String):
    print('Consumed', str)

struct TwoStrings4:
    var str1: String
    var str2: String

    fn __init__(inout self, one: String):
        self.str1 = one
        self.str2 = String("bar")

    fn __moveinit__(inout self, owned existing: Self):
        self.str1 = existing.str1
        self.str2 = existing.str2

    fn __del__(owned self):
        self.dump() # Self is still whole here
        # Mojo calls self.str2.__del__() since str2 isn't used anymore
        consume(self.str1^)
        # str1 has now been transferred;
        # `self.__del__()` is not called (avoiding an infinite loop).

    fn dump(inout self):
        print('str1:', self.str1)
        print('str2:', self.str2)

fn use_two_strings():
    let two_strings = TwoStrings4("foo")

# We use a function call to ensure the `two_strings` ownership is enforced
# (Currently, ownership is not enforced for top-level code in notebooks)
use_two_strings()_

```

str1: foo  
str2: bar  
Consumed foo

You should not generally have to think about this, but if you have logic with inner pointers into members, you may need to keep them alive for some logic within the destructor or move initializer itself. You can do this by assigning to the `_discard` pattern:

```

fn __del__(owned self):
    self.dump() # Self is still whole here

    consume(self.str1^)
    self.str2 =
    # self.str2.__del__(): Mojo destroys str2 after its last use.__

```

In this case, if `consume()` implicitly refers to some value in `str2` somehow, this will ensure that `str2` isn't destroyed until the last use when it is accessed by the `_discard` pattern.

## Defining the `__del__` destructor

You should define the `__del__()` method to perform any kind of cleanup the type requires. Usually, that includes freeing memory for any fields that are not trivial or destructible—Mojo automatically destroys any trivial and destructible types as soon as they're not used anymore.

For example, consider this struct:

```

struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, owned name: String, age: Int):
        self.name = name^
        self.age = age_

```

There's no need to define the `__del__()` method because `String` is a destructible (it has its own `__del__()` method) and Mojo destroys it as soon as it's no longer used (which is exactly when the

MyPet instance is no longer used), and `Int` is a [trivial type](#) and Mojo reclaims this memory also as soon as possible (although a little differently, without need for a `__del__()` method).

Whereas, the following struct must define the `__del__()` method to free the memory allocated for its `Pointer`:

```
struct Array[Type: AnyType]:  
    var data: Pointer[Type]  
    var size: Int  
  
    fn __init__(inout self, size: Int, value: Type):  
        self.size = size  
        self.data = Pointer[Type].alloc(self.size)  
        for i in range(self.size):  
            self.data.store(i, value)  
  
    fn __del__(owned self):  
        self.data.free()□
```

## Lifetimes

TODO: Explain how returning references work, tied into lifetimes which dovetail with parameters. This is not enabled yet.

## Type traits

This is a feature very much like Rust traits or Swift protocols or Haskell type classes. Note, this is not implemented yet.

## Advanced/Obscure Mojo features

This section describes power-user features that are important for building the bottom-est level of the standard library. This level of the stack is inhabited by narrow features that require experience with compiler internals to understand and utilize effectively.

### `@register_passable` struct decorator

The default model for working with values is they live in memory, so they have an identity, which means they are passed indirectly to and from functions (equivalently, they are passed “by reference” at the machine level). This is great for types that cannot be moved, and is a safe default for large objects or things with expensive copy operations. However, it is inefficient for tiny things like a single integer or floating point number.

To solve this, Mojo allows structs to opt-in to being passed in a register instead of passing through memory with the `@register_passable` decorator. You’ll see this decorator on types like `Int` in the standard library:

```
@register_passable("trivial")  
struct Int:  
    var value: __mlir_type.`!pop.scalar<index>`  
  
    fn __init__(value: __mlir_type.`!pop.scalar<index>`) -> Self:  
        return Self {value: value}  
    ...□
```

The basic `@register_passable` decorator does not change the fundamental behavior of a type: it still needs to have a `__copyinit__` method to be copyable, may still have a `__init__` and `__del__` methods, etc. The major effect of this decorator is on internal implementation details: `@register_passable` types are typically passed in machine registers (subject to the details of the underlying architecture).

There are only a few observable effects of this decorator to the typical Mojo programmer:

1. `@register_passable` types are not able to hold instances of types that are not themselves `@register_passable`.
2. Instances of `@register_passable` types do not have predictable identity, and so the `self` pointer is not stable/predictable (e.g. in hash tables).
3. `@register_passable` arguments and result are exposed to C and C++ directly, instead of being passed by-pointer.
4. The `__init__` and `__copyinit__` methods of this type are implicitly static (like `__new__` in Python) and return their results by-value instead of taking `inout self`.

We expect that this decorator will be used pervasively on core standard library types, but is safe to ignore for general application level code.

The `Int` example above actually uses the “trivial” variant of this decorator. It changes the passing convention as described above but also disallows copy and move constructors and destructors (synthesizing them all trivially).

TODO: Trivial needs to be decoupled to its own decorator since it applies to memory types as well.

## **@always\_inline decorator**

`@always_inline("nodebug")`: same thing but without debug information so you don't step into the + method on `Int`.

## **@parameter decorator**

The `@parameter` decorator can be placed on nested functions that capture runtime values to create “parametric” capturing closures. This is an unsafe feature in Mojo, because we do not currently model the lifetimes of capture-by-reference. A particular aspect of this feature is that it allows closures that capture runtime values to be passed as parameter values.

## **Magic operators**

C++ code has a number of magic operators that intersect with value lifecycle, things like “placement new”, “placement delete” and “operator=” that reassign over an existing value. Mojo is a safe language when you use all its language features and compose on top of safe constructs, but of any stack is a world of C-style pointers and rampant unsafety. Mojo is a pragmatic language, and since we are interested in both interoperating with C/C++ and in implementing safe constructs like `String` directly in Mojo itself, we need a way to express unsafe things.

The Mojo standard library `Pointer[element_type]` type is implemented with an underlying `!kgen.pointer<element_type>` type in MLIR, and we desire a way to implement these C++-equivalent unsafe constructs in Mojo. Eventually, these will migrate to all being methods on the `Pointer` type, but until then, some need to be exposed as built-in operators.

## **Direct access to MLIR**

Mojo provides full access to the MLIR dialects and ecosystem. Please take a look at the [Low level IR in Mojo](#) to learn how to use the `__mlir_type`, `__mlir_op`, and `__mlir_type` constructs. All of the built-in and standard library APIs are implemented by just calling the underlying MLIR constructs, and in doing so, Mojo effectively serves as syntax sugar on top of MLIR.

# Mojo[] modules

A list of all modules in the current standard library.

Mojo is a very young language, so these are the only modules we're making available at this time. There is much more to come!

## Standard library modules

### arg

[Implements functions and variables for interacting with execution and system environment.](#)

### atomic

[Implements the Atomic class.](#)

### autotuning

[Implements the autotune functionality.](#)

### base64

[Provides functions for base64 encoding strings.](#)

### benchmark

[Implements the Benchmark class for runtime benchmarking.](#)

### bit

[Provides functions for bit manipulation.](#)

### bool

[Implements the Bool class.](#)

### buffer

[Implements the Buffer class.](#)

### builtin\_list

[Implements the ListLiteral class.](#)

### builtin\_slice

[Implements slice.](#)

### complex

[Implements the Complex type.](#)

### constrained

[Implements compile time constraints.](#)

**coroutine**

[Implements classes and methods for coroutines.](#)

**debug\_assert**

[Implements a debug assert.](#)

**dtype**

[Implements the DTyPe class.](#)

**env**

[Implements basic routines for working with the OS.](#)

**error**

[Implements the Error class.](#)

**file**

[Implements the file based methods.](#)

**float literal**

[Implements the FloatLiteral class.](#)

**functional**

[Implements higher-order functions.](#)

**index**

[Implements StaticIntTuple which is commonly used to represent N-D indices.](#)

**info**

[Implements methods for querying the host target info.](#)

**int**

[Implements the Int class.](#)

**intrinsics**

[Defines intrinsics.](#)

**io**

[Provides utilities for working with input/output.](#)

**len**

[Provides the len function.](#)

## limit

[Provides interfaces to query numeric various numeric properties of types.](#)

## list

[Provides utilities for working with static and variadic lists.](#)

## math

[Defines math utilities.](#)

## memory

[Defines functions for memory manipulations.](#)

## numerics

[Defines utilities to work with numeric types.](#)

## object

[Defines the object type, which is used to represent untyped values.](#)

## object

[Implements PyObject.](#)

## param\_env

[Implements functions for retrieving compile-time defines.](#)

## path

## Aliases:

## polynomial

[Provides two implementations for evaluating polynomials.](#)

## python

[Implements Python interoperability.](#)

## random

[Provides functions for random numbers.](#)

## range

[Implements a 'range' call.](#)

## rebind

[Implements type rebind.](#)

## reduction

[Implements SIMD reductions.](#)

**simd**

[Implements SIMD struct.](#)

**sort**

[Implements sorting functions.](#)

**static\_tuple**

[Implements StaticTuple, a statically-sized uniform container.](#)

**string**

[Implements basic object methods for working with strings.](#)

**string\_literal**

[Implements the StringLiteral class.](#)

**StringRef**

[Implements the StringRef class.](#)

**tensor**

[Implements the Tensor type.](#)

**tensor\_shape**

[Implements the TensorShape type.](#)

**tensor\_spec**

[Implements the TensorSpec type.](#)

**testing**

[Implements various testing utils.](#)

**time**

[Implements basic utils for working with time.](#)

**tuple**

[Implements the Tuple type.](#)

**type\_aliases**

[Defines some type aliases.](#)

**unsafe**

[Implements classes for working with unsafe pointers.](#)

## [vector](#)

[Defines several vector-like classes.](#)

No matching items

# Mojo[] notebooks

All the Jupyter notebooks we've created for the Mojo Playground.

The following pages are rendered from the Jupyter notebooks that are [available on GitHub](#) and in the [Mojo Playground](#).

## [Low-level IR in Mojo](#)

[Learn how to use low-level primitives to define your own boolean type in Mojo.](#)

## [Mandelbrot in Mojo with Python plots](#)

[Learn how to write high-performance Mojo code and import Python packages.](#)

## [Matrix multiplication in Mojo](#)

[Learn how to leverage Mojo's various functions to write a high-performance matmul.](#)

## [Fast memset in Mojo](#)

[Learn how to use Mojo's autotuning to quickly write a memset function.](#)

## [Ray tracing in Mojo](#)

[Learn how to draw 3D graphics with ray-traced lighting using Mojo.](#)

No matching items

# Mojo[] roadmap & sharp edges

A summary of our Mojo plans, including upcoming features and things we need to fix.

This document captures the broad plan about how we plan to implement things in Mojo, and some early thoughts about key design decisions. This is not a full design spec for any of these features, but it can provide a “big picture” view of what to expect over time. It is also an acknowledgement of major missing components that we plan to add.

## Overall priorities

Mojo is still in early development and many language features will arrive in the coming months. We are highly focused on building Mojo the right way (for the long-term), so we want to fully build-out the core Mojo language features before we work on other dependent features and enhancements.

Currently, that means we are focused on the core system programming features that are essential to [Mojo’s mission](#), and as outlined in the following sections of this roadmap.

In the near-term, we will **not** prioritize “general goodness” work such as:

- Adding syntactic sugar and short-hands for Python.
- Adding features from other languages that are missing from Python (such as public/private declarations).
- Tackling broad Python ecosystem challenges like packaging.

If you have encountered any bugs with current Mojo behavior, please [submit an issue on GitHub](#).

If you have ideas about how to improve the core Mojo features, we prefer that you first look for similar topics or start a new conversation about it in our [GitHub Discussions](#).

We also consider Mojo to be a new member of the Python family, so if you have suggestions to improve the experience with Python, we encourage you to propose these “general goodness” enhancements through the formal [PEP process](#).

## Why not add syntactic sugar or other minor new features?

We are frequently asked whether Mojo will add minor features that people love in other languages but that are missing in Python, such as “implicit return” at the end of a function, public/private access control, fixing Python packaging, and various syntactic shorthands. As mentioned above, we are intentionally *not* adding these kinds of features to Mojo right now. There are three major reasons for this:

- First, Mojo is still young: we are still “building a house” by laying down major bricks in the type system and adding system programming features that Python lacks. We know we need to implement support for many existing Python features (compatibility a massive and important goal of Mojo) and this work is not done yet. We have limited engineering bandwidth and want focus on building essential functionality, and we will not debate whether certain syntactic sugar is important or not.
- Second, syntactic sugar is like mortar in a building—its best use is to hold the building together by filling in usability gaps. Sugar (and mortar) is problematic to add early into a system: you can run into problems with laying the next bricks because the sugar gets in the way. We have experience building other languages (such as Swift) that added sugar early, which could have been subsumed by more general features if time and care were given to broader evaluation.

- Third, the Python community should tackle some of these ideas first. It is important to us that Mojo be a good member of the Python family (a “Python++”), not just a language with Pythonic syntax. As such, we don’t want to needlessly diverge from Python evolution: adding a bunch of features could lead to problems down the road if Python makes incompatible decisions. Such a future would fracture the community which would cause massively more harm than any minor language feature could offset.

For all these reasons, “nice to have” syntactic sugar is not a priority, and we will quickly close such proposals to avoid cluttering the issue tracker. If you’d like to propose a “general goodness” syntactic feature, please do so with the existing [Python PEP process](#). If/when Python adopts a feature, Mojo will also add it, because Mojo’s goal is to be a superset. We are happy with this approach because the Python community is better equipped to evaluate these features, they have mature code bases to evaluate them with, and they have processes and infrastructure for making structured language evolution features.

## Mojo SDK known issues

The [Mojo SDK](#) is still in early development and currently only available for Ubuntu Linux and macOS (Apple silicon) systems. Here are some of the notable issues that we plan to fix:

- Missing native support for Windows, Intel Macs, and Linux distributions other than Ubuntu. Currently, we support Ubuntu systems with x86-64 processors only. Support for more Linux distributions (including Debian and RHEL) and Windows is in progress.
- Python interoperability might fail when running a compiled Mojo program, with the message `Unable to locate a suitable libpython, please set MOJO_PYTHON_LIBRARY`. This is because we currently do not embed the Python version into the Mojo binary. For details and the workaround, see [issue #551](#).
- Modular CLI install might fail and require `modular clean` before you re-install.

If it asks you to perform auth, run `modular auth <MODULAR_AUTH>` and use the `MODULAR_AUTH` value shown for the `curl` command on [the download page](#).

- `modular install mojo` is slow and might appear unresponsive (as the installer is downloading packages in the background). We will add a progress bar in a future release.
- If you attempt to uninstall Mojo with `modular uninstall`, your subsequent attempt to install Mojo might fail with an HTTP 500 error code. If so, run `modular clean` and try again.
- Mojo REPL might hang (become unresponsive for more than 10 seconds) when interpreting an expression if your system has 4 GiB or less RAM. If you encounter this issue, please report it with your system specs.

Additionally, we’re aware of some issues that we might not be able to solve, but we mention them here with some more information:

- When installing Mojo, if you receive the error, `failed to reach URL https://cas.modular.com`, it could be because your network connection is behind a firewall. Try updating your firewall settings to allow access to these end points: `https://packages.modular.com` and `https://cas.modular.com`. Then retry with `modular clean` and `modular install mojo`.
- When installing Mojo, if you receive the error, `gpg: no valid OpenGPG data found`, this is likely because you are located outside our supported geographies. Due to US export control restrictions, we are unable to provide access to Mojo to users situated in specific countries.
- If using Windows Subsystem for Linux (WSL), you might face issues with WSL1. We recommend you upgrade to WSL2. To check the version, run `wsl -l -v`.

You can see other [reported issues on GitHub](#).

# Small independent features

There are a number of features that are missing that are important to round out the language fully, but which don't depend strongly on other features. These include things like:

- Improved package management support.
- Many standard library features, including canonical arrays and dictionary types, copy-on-write data structures, etc.
- Support for “top level code” at file scope.
- Algebraic data types like `enum` in Swift/Rust, and pattern matching.
- Many standard library types, including `Optional[T]` and `Result[T, Error]` types when we have algebraic datatypes and basic traits.

## Ownership and Lifetimes

The ownership system is partially implemented, and is expected to get built out in the next couple of months. The basic support for ownership includes features like:

- Capture declarations in closures.
- Borrow checker: complain about invalid mutable references.

The next step in this is to bring proper lifetime support in. This will add the ability to return references and store references in structures safely. In the immediate future, one can use the unsafe `Pointer` struct to do this like in C++.

## Protocols / Traits

Unlike C++, Mojo does not “instantiate templates” in its parser. Instead, it has a separate phase that works later in the compilation pipeline (the “Elaborator”) that instantiates parametric code, which is aware of autotuning and caching. This means that the parser has to perform full type checking and IR generation without instantiating algorithms.

The planned solution is to implement language support for Protocols - variants of this feature exist in many languages (e.g. Swift protocols, Rust traits, Haskell typeclasses, C++ concepts) all with different details. This feature allows defining requirements for types that conform to them, and dovetails into static and dynamic metaprogramming features.

## Classes

Mojo still doesn’t support classes, the primary thing Python programmers use pervasively! This isn’t because we hate dynamism - quite the opposite. It is because we need to get the core language semantics nailed down before adding them. We expect to provide full support for all the dynamic features in Python classes, and want the right framework to hang that off of.

When we get here, we will discuss what the right default is: for example, is full Python hash-table dynamism the default? Or do we use a more efficient model by default (e.g. vtable-based dispatch and explicitly declared stored properties) and allow opt’ing into dynamism with a `@dynamic` decorator on the class. The latter approach worked well for Swift (its [@objc attribute](#)), but we’ll have to prototype to better understand the tradeoffs.

## C/C++ Interop

Integration to transparently import Clang C/C++ modules. Mojo’s type system and C++’s are pretty compatible, so we should be able to have something pretty nice here. Mojo can leverage Clang to transparently generate a foreign function interface between C/C++ and Mojo, with the ability to directly import functions:

```
from "math.h" import cos  
print(cos(0))
```

## Full MLIR decorator reflection

All decorators in Mojo have hard-coded behavior in the parser. In time, we will move these decorators to being compile-time metaprograms that use MLIR integration. This may depend on C++ interop for talking to MLIR. This completely opens up the compiler to programmers. Static decorators are functions executed at compile-time with the capability to inspect and modify the IR of functions and types.

```
fn value(t: TypeSpec):  
    t.__copyinit__ = # synthesize dunder copyinit automatically  
  
@value  
struct TrivialType: pass  
  
fn full_unroll(loop: mlir.Operation):  
    # unrolling of structured loop  
  
fn main():  
    @full_unroll  
    for i in range(10):  
        print(i)
```

## Sharp Edges

The entire Modular kernel library is written in Mojo, and its development has been prioritized based on the internal needs of those users. Given that Mojo is still a young language, there are a litany of missing small features that many Python and systems programmers may expect from their language, as well as features that don't quite work the way we want to yet, and in ways that can be surprising or unexpected. This section of the document describes a variety of "sharp edges" in Mojo, and potentially how to work around them if needed. We expect all of these to be resolved in time, but in the meantime, they are documented here.

### No list or dict comprehensions

Mojo does not yet support Python list or dictionary comprehension expressions, like `[x for x in range(10)]`, because Mojo's standard library has not yet grown a standard list or dictionary type.

### No lambda syntax

Mojo does not yet support defining anonymous functions with the `lambda` keyword.

### No parametric aliases

Mojo aliases can refer to parametric values but cannot themselves be a parameter. We would like this example to work, however:

```
alias Scalar[dt: DType] = SIMD[dt, 1]  
alias mul2[x: Int] = x * 2
```

### Exception is actually called Error

In Python, programmers expect that exceptions all subclass the `Exception` builtin class. The only available type for Mojo "exceptions" is `Error`:

```
fn raise_an_error() raises:  
    raise Error("I'm an error!")
```

The reason we call this type `Error` instead of `Exception` is because it's not really an exception. It's not an exception, because raising an error does not cause stack unwinding, but most importantly it does not have a stack trace. And without polymorphism, the `Error` type is the only kind of error that can be raised in Mojo right now.

## No Python-style generator functions

Mojo does not yet support Python-style generator functions (`yield` syntax). These are “synchronous co-routines” – functions with multiple suspend points.

## No `async for` or `async with`

Although Mojo has support for `async` functions with `async fn` and `async def`, Mojo does not yet support the `async for` and `async with` statements.

## The `rebind` builtin

One of the consequences of Mojo not performing function instantiation in the parser like C++ is that Mojo cannot always figure out whether some parametric types are equal and complain about an invalid conversion. This typically occurs in static dispatch patterns, like:

```
fn take_simd8(x: SIMD[DType.float32, 8]): pass

fn generic_simd[nelts: Int](x: SIMD[DType.float32, nelts]):
    @parameter
    if nelts == 8:
        take_simd8(x)___
```

The parser will complain,

```
error: invalid call to 'take_simd8': argument #0 cannot be converted from
'SIMD[f32, nelts]' to 'SIMD[f32, 8]'
    take_simd8(x)
~~~~~^~~
```

This is because the parser fully type-checks the function without instantiation, and the type of `x` is still `SIMD[f32, nelts]`, and not `SIMD[f32, 8]`, despite the static conditional. The remedy is to manually “rebind” the type of `x`, using the `rebind` builtin, which inserts a compile-time assert that the input and result types resolve to the same type after function instantiation.

```
fn generic_simd[nelts: Int](x: SIMD[DType.float32, nelts]):
 @parameter
 if nelts == 8:
 take_simd8(rebind[SIMD[DType.float32, 8]](x))___
```

## Scoping and mutability of statement variables

Python programmers understand that local variables are implicitly declared and scoped at the function level. As the programming manual explains, this feature is supported in Mojo only inside `def` functions. However, there are some nuances to Python’s implicit declaration rules that Mojo does not match 1-to-1.

For example, the scope of `for` loop iteration variables and caught exceptions in `except` statements is limited to the next indentation block, for both `def` and `fn` functions. Python programmers will expect the following program to print “2”:

```
for i in range(3): pass
print(i)___
```

However, Mojo will complain that `print(i)` is a use of an unknown declaration. This is because whether `i` is defined at this line is dynamic in Python. For instance the following Python program will fail:

```
for i range(0): pass
print(i)___
```

With `NameError: name 'i' is not defined`, because the definition of `i` is a dynamic characteristic of the function. Mojo's lifetime tracker is intentionally simple (so lifetimes are easy to use!), and cannot reason that `i` would be defined even when the loop bounds are constant.

Also stated in the programming manual: in `def` functions, the function arguments are mutable and re-assignable, whereas in `fn`, function arguments are rvalues and cannot be re-assigned. The same logic extends to statement variables, like `for` loop iteration variables or caught exceptions:

```
def foo():
 try:
 bad_function()
 except e:
 e = Error() # ok: we can overwrite 'e'

fn bar():
 try:
 bad_function()
 except e:
 e = Error() # error: 'e' is not mutable___
```

## Name scoping of nested function declarations

In Python, nested function declarations produce dynamic values. They are essentially syntax sugar for `bar = lambda ...`.

```
def foo():
 def bar(): # creates a function bound to the dynamic value 'bar'
 pass
 bar() # indirect call___
```

In Mojo, nested function declarations are static, so calls to them are direct unless made otherwise.

```
fn foo():
 fn bar(): # static function definition bound to 'bar'
 pass
 bar() # direct call
 let f = bar # materialize 'bar' as a dynamic value
 f() # indirect call___
```

Currently, this means you cannot declare two nested functions with the same name. For instance, the following example does not work in Mojo:

```
def pick_func(cond):
 if cond:
 def bar(): return 42
 else:
 def bar(): return 3 # error: redeclaration of 'bar'
 return bar___
```

The functions in each conditional must be explicitly materialized as dynamic values.

```
def pick_func(cond):
 let result: def() capturing # Mojo function type
 if cond:
 def bar0(): return 42
 result = bar0
 else:
 def bar1(): return 3 # error: redeclaration of 'bar'
 result = bar1
 return result___
```

We hope to sort out these oddities with nested function naming as our model of closures in Mojo develops further.

# No polymorphism

Mojo will implement static polymorphism through traits/protocols in the near future and dynamic polymorphism through classes and MLIR reflection. None of those things exist today, which presents several limitations to the language.

Python programmers are used to implementing special dunder methods on their classes to interface with generic methods like `print` and `len`. For instance, one expects that implementing `__repr__` or `__str__` on a class will enable that class to be printed via `print`.

```
class One:
 def __init__(self): pass
 def __repr__(self): return '1'

print(One()) # prints '1'
```

This is not currently possible in Mojo. Overloads of `print` are provided for common types, like `Int` and `SIMD` and `String`, but otherwise the builtin is not extensible. Overloads also have the limitation that they must be defined within the same module, so you cannot add a new overload of `print` for your struct types.

The same extends to `range`, `len`, and other builtins.

## No lifetime tracking inside collections

Due to the aforementioned lack of polymorphism, collections like lists, maps, and sets are unable to invoke element destructors. For collections of trivial types, like `DynamicVector[Int]`, this is no problem, but for collections of types with lifetimes, like `DynamicVector[String]`, the elements have to be manually destructed. Doing so requires quite an ugly pattern, shown in the next section.

## No safe value references

Mojo does not have proper lifetime marker support yet, and that means it cannot reason about returned references, so Mojo doesn't support them. You can return or keep unsafe references by passing explicit pointers around.

```
struct StringRef:
 var ref: Pointer[SI8]
 var size: Int
 # ...

fn bar(x: StringRef): pass

fn foo():
 let s: String = "1234"
 let ref: StringRef = s # unsafe reference
 bar(ref)
 _ = s # keep the backing memory alive!
```

Mojo will destruct objects as soon as it thinks it can. That means the lifetime of objects to which there are unsafe references must be manually extended. See the [lifetime document](#) for more details. This disables the RAI pattern in Mojo. Context managers and `with` statements are your friends in Mojo.

No lvalue returns also mean that implementing certain patterns require magic keywords until proper lifetime support is built. One such pattern is retrieving an unsafe reference from an object.

```
struct UnsafeIntRef:
 var ptr: Pointer[Int]

fn printIntRef(x: UnsafeIntRef):
 # "dereference" operator
 print(__get_address_as_lvalue(x.ptr)) # Pointer[Int] -> &Int
```

```
var c: Int = 10
"reference" operator
let ref = UnsafeIntRef(__get_lvalue_as_address(c)) # &Int -> Pointer[Int]
```

## Parameter closure captures are unsafe references

You may have seen nested functions, or “closures”, annotated with the `@parameter` decorator. This creates a “parameter closure”, which behaves differently than a normal “stateful” closure. A parameter closure declares a compile-time value, similar to an alias declaration. That means parameter closures can be passed as parameters:

```
fn take_func[f: fn() capturing -> Int]():
 pass

fn call_it(a: Int):
 @parameter
 fn inner() -> Int:
 return a # capture 'a'

 take_func[inner]() # pass 'inner' as a parameter
```

Parameter closures can even be parametric and capturing:

```
fn take_func[f: fn[a: Int]() capturing -> Int]():
 pass

fn call_it(a: Int):
 @parameter
 fn inner[b: Int]() -> Int:
 return a + b # capture 'a'

 take_func[inner]() # pass 'inner' as a parameter
```

However, note that parameter closures are always capture by *unsafe* reference. Mojo’s lifetime tracking is not yet sophisticated enough to form safe references to objects (see above section). This means that variable lifetimes need to be manually extended according to the lifetime of the parametric closure:

```
fn print_it[f: fn() capturing -> String]():
 print(f())

fn call_it():
 let s: String = "hello world"
 @parameter
 fn inner() -> String:
 return s # 's' captured by reference, so a copy is made here
 # lifetime tracker destroys 's' here

 print_it[inner]() # crash! 's' has been destroyed
```

The lifetime of the variable can be manually extended by discarding it explicitly.

```
fn call_it():
 let s: String = "hello world"
 @parameter
 fn inner() -> String:
 return s

 print_it[inner]()
 _ = s^ # discard 's' explicitly
```

A quick note on the behaviour of “stateful” closures. One sharp edge here is that stateful closures are *always* capture-by-copy; Mojo lacks syntax for move-captures and the lifetime tracking necessary for capture-by-reference. Stateful closures are runtime values – they cannot be passed as parameters, and they cannot be parametric. However, a nested function is promoted to a parametric closure if it does not capture anything. That is:

```
fn foo0[f: fn() capturing -> String](): pass
fn foo1[f: fn[a: Int]() capturing -> None](): pass
```

```

fn main():
 let s: String = "hello world"
 fn stateful_captures() -> String:
 return s # 's' is captured by copy

foo0[stateful_captures]() # not ok: 'stateful_captures' is not a parameter
fn stateful_nocapture[a: Int](): # ok: can be parametric, since no captures
 print(a)

fool[stateful_nocapture]() # ok: 'stateful_nocapture' is a parameter

```

## The standard library has limited exceptions use

For historic and performance reasons, core standard library types typically do not use exceptions. For instance, `DynamicVector` will not raise an out-of-bounds access (it will crash), and `Int` does not throw on divide by zero. In other words, most standard library types are considered “unsafe”.

```

let v = DynamicVector[Int](0)
print(v[1]) # could crash or print garbage values (undefined behaviour)

print(1//0) # does not raise and could print anything (undefined behaviour)

```

This is clearly unacceptable given the strong memory safety goals of Mojo. We will circle back to this when more language features and language-level optimizations are available.

## Nested functions cannot be recursive

Nested functions (any function that is not a top-level function) cannot be recursive in any way. Nested functions are considered “parameters”, and although parameter values do not have to obey lexical order, their uses and definitions cannot form a cycle. Current limitations in Mojo mean that nested functions, which are considered parameter values, cannot be cyclic.

```

fn try_recursion():
 fn bar(x: Int): # error: circular reference :(
 if x < 10:
 bar(x + 1)

```

## Only certain loaded MLIR dialects can be accessed

Although Mojo provides features to access the full power of MLIR, in reality only a certain number of loaded MLIR dialects can be accessed in the Playground at the moment.

The upstream dialects available in the Playground are the [index](#) dialect and the [LLVM](#) dialect.

# MojoFAQ

Answers to questions we expect about Mojo.

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!

# Mojo changelog

A history of significant Mojo changes.

This is a running list of significant changes for the Mojo language and tools. It doesn't include all internal implementation changes.

## Update Mojo

If you don't have Mojo yet, see the [get started guide](#).

To see your current Mojo version, run this:

```
mojo --version
```

To update Mojo to the latest release, run this:

```
modular update mojo
```

However, if your current version is 0.3.0 or lower, you'll need these additional commands:

```
sudo apt-get update
sudo apt-get install modular
modular clean
modular install mojo
```

## v0.4.0 for Mac (2023-10-19)

### Legendary

- Mojo for Mac!

The Mojo SDK now works on macOS (Apple silicon). This is the same version previously released for Linux. Get the latest version of the SDK for your Mac system:

[Download Now!](#)

## v0.4.0 (2023-10-05)

### New

- Mojo now supports default parameter values. For example:

```
fn foo[a: Int = 3, msg: StringLiteral = "woof"]():
 print(msg, a)

fn main():
 foo() # prints 'woof 3'
 foo[5]() # prints 'woof 5'
 foo[7, "meow"]() # prints 'meow 7'
```

Inferred parameter values take precedence over defaults:

```
@value
struct Bar[v: Int]:
 pass

fn foo[a: Int = 42, msg: StringLiteral = "quack"](bar: Bar[a]):
 print(msg, a)

fn main():
 foo(Bar[9]()) # prints 'quack 9'
```

Structs also support default parameters:

```
@value
struct DefaultParams[msg: StringLiteral = "woof"]:
 alias message = msg

fn main():
 print(DefaultParams[]().message) # prints 'woof'
 print(DefaultParams["meow"]().message) # prints 'meow'
```

- The new [file](#) module adds basic file I/O support. You can now write:

```
var f = open("my_file.txt", "r")
print(f.read())
f.close()
```

or

```
with open("my_file.txt", "r") as f:
 print(f.read())
```

- Mojo now allows context managers to support an `_enter_` method without implementing support for an `_exit_` method, enabling idioms like this:

```
This context manager consumes itself and returns it as the value.
fn _enter_(owned self) -> Self:
 return self^
```

Here Mojo *cannot* invoke a `noop __exit__` method because the context manager is consumed by the `__enter__` method. This can be used for types (like file descriptors) that are traditionally used with `with` statements, even though Mojo's guaranteed early destruction doesn't require that.

- A very basic version of `pathlib` has been implemented in Mojo. The module will be improved to achieve functional parity with Python in the next few releases.
- The `memory.unsafe` module now contains a `bitcast` function. This is a low-level operation that enables bitcasting between pointers and scalars.
- The input parameters of a parametric type can now be directly accessed as attribute references on the type or variables of the type. For example:

```
@value
struct Thing[param: Int]:
 pass

fn main():
 print(Thing[2].param) # prints '2'
 let x = Thing[9]()
 print(x.param) # prints '9'
```

Input parameters on values can even be accessed in parameter contexts. For example:

```
fn foo[value: Int]():
 print(value)

let y = Thing[12]()
alias constant = y.param + 4
foo[constant]() # prints '16'
```

- The Mojo REPL now supports code completion. Press `Tab` while typing to query potential completion results.
- Error messages from Python are now exposed in Mojo. For example the following should print No module named '`my_uninstalled_module`':

```
fn main():
 try:
 let my_module = Python.import_module("my_uninstalled_module")
 except e:
 print(e)
```

- Error messages can now store dynamic messages. For example, the following should print "Failed on: Hello"

```
fn foo(x:String) raises:
 raise Error("Failed on: " + x)

fn main():
 try:
 foo("Hello")
 except e:
 print(e)
```

## □ Changed

- We have improved and simplified the `parallelize` function. The function now elides some overhead by caching the Mojo parallel runtime.
- The Mojo REPL and Jupyter environments no longer implicitly expose `Python`, `PythonObject`, or `Pointer`. These symbols must now be imported explicitly, for example:

```
from python import Python
from python.object import PythonObject
from memory.unsafe import Pointer
```

- The syntax for specifying attributes with the `__mlir_op` prefix have changed to mimic Python's keyword argument passing syntax. That is, `=` should be used instead of `:`, e.g.:

```
Old syntax, now fails.
__mlir_op.`index.bool.constant`[value : __mlir_attr.`false`]()
New syntax.
__mlir_op.`index.bool.constant`[value=__mlir_attr.`false`()]
```

- You can now print the `Error` object directly. The `message()` method has been removed.

## □ Fixed

- [#794](#) - Parser crash when using the `in` operator.
- [#936](#) - The `Int` constructor now accepts other `Int` instances.
- [#921](#) - Better error message when running `mojo` on a module with no `main` function.
- [#556](#) - `UInt64s` are now printed correctly.
- [#804](#) - Emit error instead of crashing when passing variadic arguments of unsupported types.
- [#833](#) - Parser crash when assigning module value.
- [#752](#) - Parser crash when calling `async def`.
- [#711](#) - The overload resolution logic now correctly prioritizes instance methods over static methods (if candidates are an equally good match otherwise), and no longer crashed if a static method has a `Self` type as its first argument.
- [#859](#) - Fix confusing error and documentation of the `rebind` builtin.
- [#753](#) - Direct use of LLVM dialect produces strange errors in the compiler.
- [#926](#) - Fixes an issue that occurred when a function with a return type of `StringRef` raised an error. When the function raised an error, it incorrectly returned the string value of that error.
- [#536](#) - Report More information on python exception.

# v0.3.1 (2023-09-28)

Our first-ever patch release of the Mojo SDK is here! Release v0.3.1 includes primarily installation-related fixes. If you've had trouble installing the previous versions of the SDK, this release may be for you.

## Fixed

- [#538](#) - Installation hangs during the testing phase. This issue occurs on machines with a low number of CPU cores, such as free AWS EC2 instances and GitHub Codespaces.
- [#590](#) - Installation fails with a “failed to run python” message.
- [#672](#) - Language server hangs on code completion. Related to #538, this occurs on machines with a low number of CPU cores.
- [#913](#) - In the REPL and Jupyter notebooks, inline comments were being parsed incorrectly.

# v0.3.0 (2023-09-21)

There's more Mojo to love in this, the second release of the Mojo SDK! This release includes new features, an API change, and bug fixes.

There's also an updated version of the [Mojo extension for VS Code](#).

## New

- Mojo now has partial support for passing keyword arguments to functions and methods. For example the following should work:

```
fn foo(a: Int, b: Int = 3) -> Int:
 return a * b

fn main():
 print(foo(6, b=7)) # prints '42'
 print(foo(a=6, b=7)) # prints '42'
 print(foo(b=7, a=6)) # prints '42'
```

Parameters can also be inferred from keyword arguments, for example:

```
fn bar[A: AnyType, B: AnyType](a: A, b: B):
 print("Hello □")

fn bar[B: AnyType](a: StringLiteral, b: B):
 print(a)

fn main():
 bar(1, 2) # prints `Hello □`
 bar(b=2, a="Yay!") # prints `Yay!`
```

For the time being, the following notable limitations apply:

- Keyword-only arguments are not supported:  

```
fn baz(*args: Int, b: Int): pass # fails
fn baz(a: Int, *, b: Int): pass # fails
```

(Keyword-only arguments are described in [PEP 3102](#).)
- Variadic keyword arguments are not supported:  

```
fn baz(a: Int, **kwargs: Int): pass # fails
```
- Mojo now supports the `@nonmaterializable` decorator. The purpose is to mark data types that should only exist in the parameter domain. To use it, a struct is decorated with `@nonmaterializable(TargetType)`. Any time the nonmaterializable type is converted from the parameter domain, it is automatically converted to `TargetType`. A nonmaterializable struct should have all of its methods annotated as `@always_inline`, and must be computable in the parameter domain. In the following example, the `NmStruct` type can be added in the parameter domain, but are converted to `HasBool` when materialized.

```
@value
@register_passable("trivial")
struct HasBool:
 var x: Bool
 fn __init__(x: Bool) -> Self:
 return Self {x: x}
 @always_inline("nodebug")
 fn __init__(nms: NmStruct) -> Self:
 return Self {x: True if (nms.x == 77) else False}

@value
@nonmaterializable(HasBool)
@register_passable("trivial")
struct NmStruct:
 var x: Int
 @always_inline("nodebug")
 fn __add__(self: Self, rhs: Self) -> Self:
 return NmStruct(self.x + rhs.x)

alias stillNmStruct = NmStruct(1) + NmStruct(2)
When materializing to a run-time variable, it is automatically converted,
even without a type annotation.
let convertedToHasBool = stillNmStruct
```

- Mojo integer literals now produce the `IntLiteral` infinite precision integer type when used in the parameter domain. `IntLiteral` is materialized to the `Int` type for runtime computation, but intermediate computations at compile time, using supported operators, can now exceed the bit width of the `Int` type.
- The Mojo Language Server now supports top-level code completions, enabling completion when typing a reference to a variable, type, etc. This resolves [#679](#).

- The Mojo REPL now colorizes the resultant variables to help distinguish input expressions from the output variables.

## Changed

- Mojo allows types to implement two forms of move constructors, one that is invoked when the lifetime of one value ends, and one that is invoked if the compiler cannot prove that. These were previously both named `_moveinit_`, with the following two signatures:

```
fn __moveinit__(inout self, owned existing: Self): ...
fn __moveinit__(inout self, inout existing: Self): ...
```

We've changed the second form to get its own name to make it more clear that these are two separate operations: the second has been renamed to `_takeinit_`:

```
fn __moveinit__(inout self, owned existing: Self): ...
fn __takeinit__(inout self, inout existing: Self): ...
```

The name is intended to connote that the operation takes the conceptual value from the source (without destroying it) unlike the first one which "moves" a value from one location to another.

- The Error type in Mojo has changed. Instead of extracting the error message using `error.value` you will now extract the error message using `error.message()`.

For more information, see [Unique "move-only" types](#) in the Mojo docs.

## Fixed

- [#503](#) - Improve error message for failure lowering `kgen.param.constant`.
- [#554](#) - Alias of static tuple fails to expand.
- [#500](#) - Call expansion failed due to verifier error.
- [#422](#) - Incorrect comment detection in multiline strings.
- [#729](#) - Improve messaging on how to exit the REPL.
- [#756](#) - Fix initialization errors of the VS Code extension.
- [#575](#) - Build LLDB/REPL with libedit for a nicer editing experience in the terminal.

## v0.2.1 (2023-09-07)

The first versioned release of Mojo! ☺

All earlier releases were considered version 0.1.

## Legendary

- First release of the Mojo SDK!

You can now develop with Mojo locally. The Mojo SDK is currently available for Ubuntu Linux systems, and support for Windows and macOS is coming soon. You can still develop from a Windows or Mac computer using a container or remote Linux system.

The Mojo SDK includes the Mojo standard library and the [Mojo command-line interface](#) (CLI), which allows you to run, compile, and package Mojo code. It also provides a REPL programming environment.

[Get the Mojo SDK!](#)

- First release of the [Mojo extension for VS Code](#).

This provides essential Mojo language features in Visual Studio Code, such as code completion, code quick fixes, docs tooltips, and more. Even when developing on a remote system, using VS Code with this extension provides a native-like IDE experience.

## New

- A new `clobber_memory` function has been added to the [benchmark](#) module. The clobber memory function tells the system to flush all memory operations at the specified program point. This allows you to benchmark operations without the compiler reordering memory operations.
- A new `keep` function has been added to the [benchmark](#) module. The `keep` function tries to tell the compiler not to optimize the variable away if not used. This allows you to avoid compiler's dead code elimination mechanism, with a low footprint side effect.
- New `shift_right` and `shift_left` functions have been added to the [simd](#) module. They shift the elements in a SIMD vector right/left, filling elements with zeros as needed.
- A new `cumsum` function has been added to the [reduction](#) module that computes the cumulative sum (also known as scan) of input elements.
- Mojo Jupyter kernel now supports code completion.

## Changed

- Extends `rotate_bits_left`, `rotate_left`, `rotate_bits_right`, and `rotate_right` to operate on `Int` values. The ordering of parameters has also been changed to enable type inference. Now it's possible to write `rotate_right[shift_val](simd_val)` and have the `dtype` and `simd_width` inferred from the argument. This addresses [Issue #528](#).

## Fixed

- Fixed a bug causing the parser to crash when the `with` statement was written without a colon. This addresses [Issue #529](#).
- Incorrect imports no longer crash when there are other errors at the top level of a module. This fixes [Issue #531](#).

# August 2023

## 2023-08-24

- Fixed issue where the `with expr as x` statement within `fn` behaved as if it were in a `def`, binding `x` with function scope instead of using lexical scope.

### □ New

- Major refactoring of the standard library to enable packaging and better import ergonomics:
  - The packages are built as binaries to improve startup speed.
  - Package and module names are now lowercase to align with the Python style.
  - Modules have been moved to better reflect the purpose of the underlying functions (e.g. `Pointer` is now within the `unsafe` module in the `memory` package).
  - The following modules are now included as built-ins: `SIMD`, `DType`, `I0`, `Object`, and `String`. This means it's no longer necessary to explicitly import these modules. Instead, these modules will be implicitly imported for the user. Private methods within the module are still accessible using the `builtin.module_name._private_method` import syntax.
  - New `math` package has been added to contain the `bit`, `math`, `numerics`, and `polynomial` modules. The contents of the `math.math` module are re-exported into the `math` package.
- Mojo now supports using memory-only types in parameter expressions and as function or type parameters:

```
@value
struct IntPair:
 var first: Int
 var second: Int

fn add_them[value: IntPair]() -> Int:
 return value.first + value.second

fn main():
 print(add_them[IntPair(1, 2)]()) # prints '3'
```

- In addition, Mojo supports evaluating code that uses heap-allocated memory at compile-time and materializing compile-time values with heap-allocated memory into dynamic values:

```
fn fillVector(lowerBound: Int, upperBound: Int, step: Int) -> DynamicVector[Int]:
 var result = DynamicVector[Int]()
 for i in range(lowerBound, upperBound, step):
 result.push_back(i)
 return result

fn main():
 alias values = fillVector(5, 23, 7)
 for i in range(0, values._len_()):
 print(values[i]) # prints '5', '12', and then '19'
```

### □ Changed

- `def main():`, without the explicit `None` type, can now be used to define the entry point to a Mojo program.
- The `assert_param` function has been renamed to `constrained` and is now a built-in function.
- The `print` function now works on Complex values.

### □ Fixed

- Fixed issues with `print` formatting for `DType.uint16` and `DType.int16`.
- [Issue #499](#) - Two new `rotate_right` and `rotate_left` functions have been added to the `SIMD` module.
- [Issue #429](#) - You can now construct a `Bool` from a `SIMD` type whose element-type is `DType.bool`.
- [Issue #350](#) - Confusing Matrix implementation
- [Issue #349](#) - Missing `load_tr` in struct `Matrix`
- [Issue #501](#) - Missing syntax error messages in Python expressions.

## 2023-08-09

### □ Changed

- The `ref` and `mutref` identifiers are now treated as keywords, which means they cannot be used as variable, attribute, or function names. These keywords are used by the “lifetimes” features, which is still in development. We can consider renaming these (as well as other related keywords) when the development work gels, support is enabled in public Mojo builds, and when we have experience using them.
- The argument handling in `def` functions has changed: previously, they had special behavior that involved mutable copies in the callee. Now, we have a simple rule, which is that `def` argument default to the `owned` convention (`fn` arguments still default to the `borrowed` convention).

This change is mostly an internal cleanup and simplification of the compiler and argument model, but does enable one niche use-case: you can now pass non-copyable types to `def` arguments by transferring ownership of a value into the `def` call. Before, that would not be possible because the copy was made on the callee side, not the caller's side. This also allows the explicit use of the `borrowed` keyword with a `def` that wants to opt-in to that behavior.

## 2023-08-03

### □ New

- A new [Tensor](#) type has been introduced. This tensor type manages its own data (unlike `NDBuffer` and `Buffer` which are just views). Therefore, the tensor type performs its own allocation and free. Here is a simple example of using the tensor type to represent an RGB image and convert it to grayscale:

```
from tensor import Tensor, TensorShape
from utils.index import Index
from random import rand

let height = 256
let width = 256
let channels = 3

Create the tensor of dimensions height, width, channels and fill with
random value.
let image = rand[DTType.float32](height, width, channels)

Declare the grayscale image.
var gray_scale_image = Tensor[DTType.float32](height, width)

Perform the RGB to grayscale transform.
for y in range(height):
 for x in range(width):
 let r = image[y,x,0]
 let g = image[y,x,1]
 let b = image[y,x,2]
 gray_scale_image[Index(y,x)] = 0.299 * r + 0.587 * g + 0.114 * b
```

## □ Fixed

- [Issue #53](#) - Int now implements true division with the `/` operator. Similar to Python, this returns a 64-bit floating point number. The corresponding in-place operator, `/=`, has the same semantics as `//=`.

# July 2023

## 2023-07-26

### □ New

- Types that define both `__getitem__` and `__setitem__` (i.e. where sub-scripting instances creates computed LValues) can now be indexed in parameter expressions.
- Unroll decorator for loops with constant bounds and steps:
  - `@unroll`: Fully unroll a loop.
  - `@unroll(n)`: Unroll a loop by factor of `n`, where `n` is a positive integer.
  - Unroll decorator requires loop bounds and iteration step to be compiler time constant value, otherwise unrolling will fail with compilation error. This also doesn't make loop induction variable a parameter.

```
Fully unroll the loop.
@unroll
for i in range(5):
 print(i)

Unroll the loop by a factor of 4 (with remainder iterations of 2).
@unroll(4)
for i in range(10):
 print(i)
```

- The Mojo REPL now prints the values of variables defined in the REPL. There is full support for scalars and structs. Non-scalar SIMD vectors are not supported at this time.

### □ Fixed

- [Issue #437](#) - Range can now be instantiated with a `PyObject`.
- [Issue #288](#) - Python strings can now be safely copied.

## 2023-07-20

### □ New

- Mojo now includes a `Limits` module, which contains functions to get the max and min values representable by a type, as requested in [Issue #51](#). The following functions moved from `Math` to `Limits`: `inf()`, `neginf()`, `isinf()`, `isfinite()`.
- Mojo decorators are now distinguished between “signature” and “body” decorators and are ordered. Signature decorators, like `@register_passable` and `@parameter`, modify the type of declaration before the body is parsed. Body decorators, like `@value`, modify the body of declaration after it is fully parsed. Due to ordering, a signature decorator cannot be applied after a body decorator. That means the following is now invalid:

```
@register_passable # error: cannot apply signature decorator after a body one!
@value
struct Foo:
 pass
```

- Global variables can now be exported in Mojo compiled archives, using the `@export` decorator. Exported global variables are public symbols in compiled archives and use the variable name as its linkage name, by default. A custom linkage name can be specified with `@export("new_name")`. This does not affect variable names in Mojo code.

- Mojo now supports packages! A Mojo package is defined by placing an `__init__.mojo` or `__init__.moj` within a directory. Other files in the same directory form modules within the package (this works exactly like it does [in Python](#)). Example:

```
main.□
my_package/
 __init__.□
 module.□
 my_other_package/
 __init__.□
 stuff.□
 □

main.□
from my_package.module import some_function
from my_package.my_other_package.stuff import SomeType

fn main():
 var x: SomeType = some_function()□
```

- Mojo now supports direct module and package imports! Modules and packages can be imported and bound to names. Module and package elements, like functions, types, global variables, and other modules, can be accessed using attribute references, like `my_module.foo`. Note that modules lack runtime representations, meaning module references cannot be instantiated.

```
import builtin.io as io
import SIMD

io.print("hello world")
var x: SIMD.Float32 = 1.2□
```

## □ Changed

- Reverted the feature from 2023-02-13 that allowed unqualified struct members. Use the `self` keyword to conveniently access struct members with bound parameters instead. This was required to fix [Issue #260](#).
- Updated the RayTracing notebook: added step 5 to create specular lighting for more realistic images and step 6 to add a background image.

## □ Fixed

- [Issue #260](#) - Definitions inside structs no longer shadow definitions outside of struct definitions.

## 2023-07-12

### □ New

- Mojo now has support for global variables! This enables `var` and `let` declaration at the top-level scope in Mojo files. Global variable initializers are run when code modules are loaded by the platform according to the order of dependencies between global variables, and their destructors are called in the reverse order.
- The [Mojo programming manual](#) is now written as a Jupyter notebook, and available in its entirety in the Mojo Playground (`programming-manual.ipynb`). (Previously, `HelloMojo.ipynb` included most of the same material, but it was not up-to-date.)
- As a result, we've also re-written `HelloMojo.ipynb` to be much shorter and provide a more gentle first-user experience.
- [Coroutine module documentation](#) is now available. Coroutines form the basis of Mojo's support for asynchronous execution. Calls to `async fns` can be stored into a `Coroutine`, from which they can be resumed, awaited upon, and have their results retrieved upon completion.

### □ Changed

- `simd_bit_width` in the `TargetInfo` module has been renamed to `simdbitwidth` to better align with `simdwidhtof`, `bitwidhtof`, etc.

### □ Fixed

- The walrus operator now works in if/while statements without parentheses, e.g. `if x := function():`
- [Issue #428](#) - The `FloatLiteral` and `SIMD` types now support conversion to `Int` via the `to_int` or `__int__` method calls. The behavior matches that of Python, which rounds towards zero.

## 2023-07-05

### □ New

- Tuple expressions now work without parentheses. For example, `a, b = b, a` works as you'd expect in Python.
- Chained assignments (e.g. `a = b = 42`) and the walrus operator (e.g. `some_function(b := 17)`) are now supported.

### □ Changed

- The `simd_width` and `dtype_simd_width` functions in the `TargetInfo` module have been renamed to `simdwidhtof`.
- The `dtype_` prefix has been dropped from `alignof`, `sizeof`, and `bitwidhtof`. You can now use these functions (e.g. `alignof`) with any argument type, including `DType`.
- The `inf`, `neginf`, `nan`, `isinf`, `isfinite`, and `isnan` functions were moved from the `Numerics` module to the `Math` module, to better align with Python's library structure.

### □ Fixed

- [Issue #253](#) - Issue when accessing a struct member alias without providing parameters.
- [Issue #404](#) - The docs now use snake\_case for variable names, which more closely conforms to Python's style.
- [Issue #379](#) - Tuple limitations have been addressed and multiple return values are now supported, even without parentheses.
- [Issue #347](#) - Tuples no longer require parentheses.
- [Issue #320](#) - Python objects are now traversable via for loops.

## June 2023

### 2023-06-29

#### □ New

- You can now share .ipynb notebook files in Mojo Playground. Just save a file in the shared directory, and then right-click the file and select **Copy Sharable link**. To open a shared notebook, you must already have [access to Mojo Playground](#); when you open a shared notebook, click **Import** at the top of the notebook to save your own copy. For more details about this feature, see the instructions inside the help directory, in the Mojo Playground file browser.

#### □ Changed

- The unroll2() and unroll3() functions in the [Functional](#) module have been renamed to overload the unroll() function. These functions unroll 2D and 3D loops and unroll() can determine the intent based on the number of input parameters.

#### □ Fixed

- [Issue #229](#) - Issue when throwing an exception from \_\_init\_\_ before all fields are initialized.
- [Issue #74](#) - Struct definition with recursive reference crashes.
- [Issue #285](#) - The [TargetInfo](#) module now includes is\_little\_endian() and is\_big\_endian() to check if the target host uses either little or big endian.
- [Issue #254](#) - Parameter name shadowing in nested scopes is now handled correctly.

### 2023-06-21

#### □ New

- Added support for overloading on parameter signature. For example, it is now possible to write the following:

```
fn foo[a: Int](x: Int):
 pass

fn foo[a: Int, b: Int](x: Int):
 pass
```

For details on the overload resolution logic, see the [programming manual](#).

- A new cost\_of() function has been added to Autotune. This meta-function must be invoked at compile time, and it returns the number of MLIR operations in a function (at a certain stage in compilation), which can be used to build basic heuristics in higher-order generators.

```
from autotune import cost_of

fn generator[f: fn(Int) -> Int]() -> Int:
 @parameter
 if cost_of[fn(Int) -> Int, f]() < 10:
 return f()
 else:
 # Do something else for slower functions...
```

- Added a new example notebook with a basic Ray Tracing algorithm.

#### □ Changed

- The constrained\_msg() in the Assert module has been renamed to constrained().

#### □ Fixed

- Overloads marked with @adaptive now correctly handle signatures that differ only in declared parameter names, e.g. the following now works correctly:

```
@adaptive
fn foobar[w: Int, T: DType]() -> SIMD[T, w]: ...

@adaptive
fn foobar[w: Int, S: DType]() -> SIMD[S, w]: ...
```

- [Issue #219](#) - Issue when redefining a function and a struct defined in the same cell.

- [Issue #355](#) - The loop order in the Matmul notebook for Python and naive mojo have been reordered for consistency. The loop order now follows (M, K, N) ordering.

- [Issue #309](#) - Use snake case naming within the testing package and move the asserts out of the TestSuite struct.

## 2023-06-14

### □ New

- Tuple type syntax is now supported, e.g. the following works:

```
fn return_tuple() -> (Int, Int):
 return (1, 2)___
```

### □ Changed

- The TupleLiteral type was renamed to just Tuple, e.g. Tuple[Int, Float].

### □ Fixed

- [Issue #354](#) - Returning a tuple doesn't work even with parens.
- [Issue #365](#) - Copy-paste error in FloatLiteral docs.
- [Issue #357](#) - Crash when missing input parameter to variadic parameter struct member function.

## 2023-06-07

### □ New

- Tuple syntax now works on the left-hand side of assignments (in "lvalue" positions), enabling things like `(a, b) = (b, a)`. There are several caveats: the element types must exactly match (no implicit conversions), this only works with values of TupleLiteral type (notably, it will not work with PythonObject yet) and parentheses are required for tuple syntax.

### □ Removed

- Mojo Playground no longer includes the following Python packages (due to size, compute costs, and [environment complications](#)): torch, tensorflow, keras, transformers.

### □ Changed

- The data types and scalar names now conform to the naming convention used by numpy. So we use Int32 instead of S132, similarly using Float32 instead of F32. Closes [Issue #152](#).

### □ Fixed

- [Issue #287](#) - computed lvalues don't handle raising functions correctly
- [Issue #318](#) - Large integers are not being printed correctly
- [Issue #326](#) - Float modulo operator is not working as expected
- [Issue #282](#) - Default arguments are not working as expected
- [Issue #271](#) - Confusing error message when converting between function types with different result semantics

## May 2023

## 2023-05-31

### □ New

- Mojo Playground now includes the following Python packages (in response to [popular demand](#)): torch, tensorflow, polars, opencv-python, keras, Pillow, plotly, seaborn, sympy, transformers.
- A new optimization is applied to non-trivial copyable values that are passed as an owned value without using the transfer (^) operator. Consider code like this:

```
var someValue : T = ...
...
takeValueAsOwned(someValue)
...___
```

When `takeValueAsOwned()` takes its argument as an [owned](#) value (this is common in initializers for example), it is allowed to do whatever it wants with the value and destroy it when it is finished. In order to support this, the Mojo compiler is forced to make a temporary copy of the `someValue` value, and pass that value instead of `someValue`, because there may be other uses of `someValue` after the call.

The Mojo compiler is now smart enough to detect when there are no uses of `someValue` later, and it will elide the copy just as if you had manually specified the transfer operator like `takeValueAsOwned(someValue^)`. This provides a nice "it just works" behavior for non-trivial types without requiring manual management of transfers.

If you'd like to take full control and expose full ownership for your type, just don't make it copyable. Move-only types require the explicit transfer operator so you can see in your code where all ownership transfer happen.

- Similarly, the Mojo compiler now transforms calls to `__copyinit__` methods into calls to `__moveinit__` when that is the last use of the source value along a control flow path. This allows types which are both copyable and movable to get transparent move optimization. For example, the following code is compiled into moves instead of copies even without the use of the transfer operator:

```

var someValue = somethingCopyableAndMovable()
use(someValue)
...
let otherValue = someValue # Last use of someValue
use(otherValue)
...
var yetAnother = otherValue # Last use of otherValue
mutate(yetAnother)_

```

This is a significant performance optimization for things like `PythonObject` (and more complex value semantic types) that are commonly used in a fluid programming style. These don't want extraneous reference counting operations performed by its copy constructor.

If you want explicit control over copying, it is recommended to use a non-dunder `.copy()` method instead of `__copyinit__`, and recall that non-copyable types must always use the transfer operator for those that want fully explicit behavior.

## Fixed

- [Issue #231](#) - Unexpected error when a Python expression raises an exception
- [Issue #119](#) - The REPL fails when a python variable is redefined

## 2023-05-24

### New

- finally clauses are now supported on try statements. In addition, try statements no longer require except clauses, allowing try-finally blocks. finally clauses contain code that is always executed from control-flow leaves any of the other clauses of a try statement by any means.

### Changed

- with statement emission changed to use the new finally logic so that

```
with ContextMgr():
 return_
```

Will correctly execute `ContextMgr.__exit__` before returning.

## Fixed

- [Issue #204](#) - Mojo REPL crash when returning a String at compile-time
- [Issue #143](#) - synthesized init in `@register_passable` type doesn't get correct convention.
- [Issue #201](#) - String literal concatenation is too eager.
- [Issue #209](#) - [QoI] Terrible error message trying to convert a type to itself.
- [Issue #32](#) - Include struct fields in docgen
- [Issue #50](#) - Int to string conversion crashes due to buffer overflow
- [Issue #132](#) - `PythonObject to_int` method has a misleading name
- [Issue #189](#) - `PythonObject bool` conversion is incorrect
- [Issue #65](#) - Add SIMD constructor from `Bool`
- [Issue #153](#) - Meaning of `Time.now` function result is unclear
- [Issue #165](#) - Type in `Pointer.free` documentation
- [Issue #210](#) - Parameter results cannot be declared outside top-level in function
- [Issue #214](#) - Pointer offset calculations at compile-time are incorrect
- [Issue #115](#) - Float printing does not include the right number of digits
- [Issue #202](#) - `kgen.unreachable` inside nested functions is illegal
- [Issue #235](#) - Crash when register passable struct field is not register passable
- [Issue #237](#) - Parameter closure sharp edges are not documented

## 2023-05-16

### New

- Added missing dunder methods to `PythonObject`, enabling the use of common arithmetic and logical operators on imported Python values.
- `PythonObject` is now [printable from Mojo](#), instead of requiring you to import Python's print function.

### Fixed

- [Issue #98](#): Incorrect error with lifetime tracking in loop.
- [Issue #49](#): Type inference issue (?) in 'ternary assignment' operation (`FloatLiteral` vs. '`SIMD[f32, 1]`').
- [Issue #48](#): and/or don't work with memory-only types.
- [Issue #11](#): `setitem` Support for `PythonObject`.

## 2023-05-11

### New

- `NDBuffer` and `Buffer` are now constructable via `Pointer` and `DTypePointer`.
- `String` now supports indexing with either integers or slices.

- Added factorial function to the Math module.

## □ Changed

- The “byref” syntax with the & sigil has changed to use an `inout` keyword to be more similar to the borrowed and owned syntax in arguments. Please see [Issue #7](#) for more information.
- Optimized the Matrix multiplication implementation in the notebook. Initially we were optimizing for expandability rather than performance. We have found a way to get the best of both worlds and now the performance of the optimized Matmul implementation is 3x faster.
- Renamed the [^postfix operator](#) from “consume” to “transfer.”

## □ Fixed

- Fixed missing overloads for `Testing.assertEqual` so that they work on `Integer` and `String` values.
- [Issue #6](#): Playground stops evaluating cells when a simple generic is defined.
- [Issue #18](#): Memory leak in Python interoperability was removed.

## 2023-05-02

### □ Released

- Mojo publicly launched! This was epic, with lots of great coverage online including a [wonderful post by Jeremy Howard](#). The team is busy this week.

### □ New

- Added a Base64 encoding function to perform base64 encoding on strings.

### □ Changed

- Decreased memory usage of serialization of integers to strings.
- Speedup the sort function.

### □ Fixed

- Fixed time unit in the `sleep` function.

## April 2023

### Week of 2023-04-24

- □ The default behavior of nested functions has been changed. Mojo nested functions that capture are by default are non-parametric, runtime closures, meaning that:

```
def foo(x):
 # This:
 def bar(y): return x*y
 # Is the same as:
 let bar = lambda y: x*y
```

These closures cannot have input or result parameters, because they are always materialized as runtime values. Values captured in the closure (x in the above example), are captured by copy: values with copy constructors cannot be copied and captures are immutable in the closure.

Nested functions that don’t capture anything are by default “parametric” closures: they can have parameters and they can be used as parameter values. To restore the previous behavior for capturing closures, “parametric, capture-by-unsafe-reference closures”, tag the nested function with the `@parameter` decorator.

- □ Mojo now has full support for “runtime” closures: nested functions that capture state materialized as runtime values. This includes taking the address of functions, indirect calls, and passing closures around through function arguments. Note that capture-by-reference is still unsafe!

You can also take references to member functions with instances of that class using `foo.member_function`, which creates a closure with `foo` bound to the `self` argument.

- □ Mojo now supports Python style `with` statements and context managers.

These things are very helpful for implementing things like our trace region support and things like Runtime support.

A context manager in Mojo implements three methods:

```
fn __enter__(self) -> T:
fn __exit__(self):
fn __exit__(self, err: Error) -> Bool:
```

The first is invoked when the context is entered, and returns a value that may optionally be bound to a target for use in the `with` body. If the `with` block exits normally, the second method is invoked to clean it up. If an error is raised, the third method is invoked with the `Error` value. If that method returns true, the error is considered handled, if it returns false, the error is re-thrown so propagation continues out of the ‘`with`’ block.

- Mojo functions now support variable scopes! Explicit `var` and `let` declarations inside functions can shadow declarations from higher “scopes”, where a scope is defined as any new indentation block. In addition, the `for` loop iteration variable is now scoped to the loop body, so it is finally possible to write

```
for i in range(1): pass
for i in range(2): pass
```

- Mojo now supports an `@value` decorator on structs to reduce boilerplate and encourage best practices in value semantics. The `@value` decorator looks to see the struct has a memberwise initializer (which has arguments for each field of the struct), a `__copyinit__` method, and a `__moveinit__` method, and synthesizes the missing ones if possible. For example, if you write:

```
@value
struct MyPet:
 var name: String
 var age: Int
```

The `@value` decorator will synthesize the following members for you:

```
fn __init__(inout self, owned name: String, age: Int):
 self.name = name^
 self.age = age
fn __copyinit__(inout self, existing: Self):
 self.name = existing.name
 self.age = existing.age
fn __moveinit__(inout self, owned existing: Self):
 self.name = existing.name^
 self.age = existing.age
```

This decorator can greatly reduce the boilerplate needed to define common aggregates, and gives you best practices in ownership management automatically. The `@value` decorator can be used with types that need custom copy constructors (your definition wins). We can explore having the decorator take arguments to further customize its behavior in the future.

- `Memcpy` and `memcmp` now consistently use `count` as the byte count.
- Add a variadic string join on strings.
- Introduce a `reduce_bit_count` method to count the number of 1 across all elements in a SIMD vector.
- Optimize the `pow` function if the exponent is integral.
- Add a `len` function which dispatches to `__len__` across the different structs that support it.

## Week of 2023-04-17

- Error messages have been significantly improved, thanks to prettier printing for Mojo types in diagnostics.
- Variadic values can now be indexed directly without wrapping them in a `VariadicList`!
- `let` declarations in a function can now be lazily initialized, and `var` declarations that are never mutated get a warning suggesting they be converted to a `let` declaration. Lazy initialization allows more flexible patterns of initialization than requiring the initializer be inline, e.g.:

```
let x : Int
if cond:
 x = foo()
else:
 x = bar()
use(x)
```

- Functions defined with `def` now return `object` by default, instead of `None`. This means you can return values (convertible to `object`) inside `def` functions without specifying a return type.
- The `@raises` decorator has been removed. Raising `fn` should be declared by specifying `raises` after the function argument list. The rationale is that `raises` is part of the type system, instead of a function modifier.
- The `BoolLiteral` type has been removed. Mojo now emits `True` and `False` directly as `Bool`.
- Syntax for function types has been added. You can now write function types with `fn(Int) -> String` or `async def(&String, *Int) -> None`. No more writing `!kgen.signature` types by hand!
- Float literals are not emitted as `FloatLiteral` instead of an MLIR `f64` type!
- Automatic destructors are now supported by Mojo types, currently spelled `fn __del__(owned self):` (the extra underscore will be dropped shortly). These destructors work like Python object destructors and similar to C++ destructors, with the major difference being that they run “as soon as possible” after the last use of a value. This means they are not suitable for use in C++-style RAII patterns (use the `with` statement for that, which is currently unsupported).

These should be generally reliable for both memory-only and register-passable types, with the caveat that closures are known to *not* capture values correctly. Be very careful with interesting types in the vicinity of a closure!

- A new (extremely dangerous!) builtin function is available for low-level ownership muckery. The `__get_address_as_owned_value(x)` builtin takes a low-level address value (of `!kgen.pointer` type) and returns an `owned` value for the memory that is pointed to. This value is assumed live at the invocation of the builtin, but is “owned” so it needs to be consumed by the caller, otherwise it will be automatically destroyed. This is an effective way to do a “placement delete” on a pointer.

```
"Placement delete": destroy the initialized object begin pointed to.
_ = __get_address_as_owned_value(somePointer.value)

Result value can be consumed by anything that takes it as an 'owned'
argument as well.
consume(__get_address_as_owned_value(somePointer.value))
```

- Another magic operator, named `__get_address_as_uninit_lvalue(x)` joins the magic LValue operator family. This operator projects a pointer to an LValue like `__get_address_as_lvalue(x)`. The difference is that `__get_address_as_uninit_lvalue(x)` tells the compiler that the pointee is uninitialized on entry and initialized on exit, which means that you can use it as a “placement new” in C++ sense. `__get_address_as_lvalue(x)` tells the compiler that the pointee is initialized already, so reassigning over it will run the destructor.

```
/*Re*placement new": destroy the existing SomeHeavy value in the memory,
then initialize a new value into the slot.
__get_address_as_lvalue(somePointer.value) = SomeHeavy(4, 5)

Ok to use an lvalue, convert to borrow etc.
use(__get_address_as_lvalue(somePointer.value))

"Placement new": Initialize a new value into uninitialized memory.
__get_address_as_uninit_lvalue(somePointer.value) = SomeHeavy(4, 5)

Error, cannot read from uninitialized memory.
use(__get_address_as_uninit_lvalue(somePointer.value))
```

Note that `__get_address_as_lvalue` assumes that there is already a value at the specified address, so the assignment above will run the `SomeHeavy` destructor (if any) before reassigning over the value.

- Implement full support for `__moveinit_` (aka move constructors)

This implements the ability for memory-only types to define two different types of move ctors if they'd like:

1. fn `__moveinit_(inout self, owned existing: Self)`: Traditional Rust style moving constructors that shuffles data around while taking ownership of the source binding.
2. fn `__moveinit_(inout self, inout existing: Self)`: C++ style “stealing” move constructors that can be used to take from an arbitrary LValue.

This gives us great expressive capability (better than Rust/C++/Swift) and composes naturally into our lifetime tracking and value categorization system.

- The `__call_` method of a callable type has been relaxed to take `self` by borrow, allow non-copyable callees to be called.
- Implicit conversions are now invoked in `raise` statements properly, allowing converting strings to `Error` type.
- Automatic destructors are turned on for `__del_` instead of `__del__`.
- Add the builtin `FloatLiteral` type.
- Add integral `floordiv` and `mod` for the SIMD type that handle negative values.
- Add an `F64` to `String` converter.
- Make the `print` function take variadic inputs.

## Week of 2023-04-10

- Introduce consume operator `x^`

This introduces the postfix consume operator, which produces an RValue given a lifetime tracked object (and, someday, a movable LValue).

- Mojo now automatically synthesizes empty destructor methods for certain types when needed.
- The `object` type has been built out into a fully-dynamic type, with dynamic function dispatch, with full error handling support.

```
def foo(a) -> object:
 return (a + 3.45) < [1, 2, 3] # raises a TypeError
```

- The `@always_inline` decorator is no longer required for passing capturing closures as parameters, for both the functions themselves as functions with capturing closures in their parameters. These functions are still inlined but it is an implementation detail of capturing parameter closures. Mojo now distinguishes between capturing and non-capturing closures. Nested functions are capturing by default and can be made non-capturing with the `@noncapturing` decorator. A top-level function can be passed as a capturing closure by marking it with the `@closure` decorator.

- Support for list literals has been added. List literals `[1, 2, 3]` generate a variadic heterogeneous list type.
- Variadics have been extended to work with memory-primary types.
- Slice syntax is now fully-supported with a new builtin `slice` object, added to the compiler builtins. Slice indexing with `a[1:2:3]` now emits calls to `__getitem_` and `__setitem_` with a slice object.
- Call syntax has been wired up to `__call_`. You can now `f()` on custom types!
- Closures are now explicitly typed as capturing or non-capturing. If a function intends to accept a capturing closure, it must specify the capturing function effect.
- Add a `Tile2D` function to enable generic 2D tiling optimizations.
- Add the `slice` struct to enable getting/setting spans of elements via `getitem/setitem`.
- Add syntax sugar to autotuning for both specifying the autotuned values, searching, and declaring the evaluation function.

## Week of 2023-04-03

- The `AnyType` and `NoneType` aliases were added and auto-imported in all files.

- ☐ The Mojo VS Code extension has been improved with docstring validation. It will now warn when a function's docstring has a wrong argument name, for example.
  - ☐ A new built-in literal type `TupleLiteral` was added in `_CompilerBuiltins`. It represents literal tuple values such as `(1, 2.0)` or `()`.
  - ☐ The `Int` type has been moved to a new `Builtin` module and is auto-imported in all code. The type of integer literals has been changed from the MLIR index type to the `Int` type.
  - Mojo now has a powerful flow-sensitive uninitialized variable checker. This means that you need to initialize values before using them, even if you overwrite all subcomponents. This enables the compiler to reason about the true lifetime of values, which is an important stepping stone to getting automatic value destruction in place.
  - ☐ Call syntax support has been added. Now you can directly call an object that implements the `__call__` method, like `foo(5)`.
  - ☐ The name for copy constructors got renamed from `__copy__` to `__copyinit__`. Furthermore, non-`@register_passable` types now implement it like they do an init method where you fill in a by-reference self, for example:
- ```
fn __copyinit__(inout self, existing: Self):
    self.first = existing.first
    self.second = existing.second
```
- This makes copy construction work more similarly to initialization, and still keeps copies `x = y` distinct from initialization `x = T(y)`.
- ☐ Initializers for memory-primary types are now required to be in the form `__init__(inout self, ...)`: with a `None` result type, but for register primary types, it remains in the form `__init__(...) -> Self`. The `T{}` initializer syntax has been removed for memory-primary types.
 - Mojo String literals now emit a builtin `StringLiteral` type! One less MLIR type to worry about.
 - New `__getattr__` and `__setattr__` dunder methods were added. Mojo calls these methods on a type when attempting member lookup of a non-static member. This allows writing dynamic objects like `x.foo()` where `foo` is not a member of `x`.
 - Early destructor support has been added. Types can now define a special destructor method `__del__` (note three underscores). This is an early feature and it is still being built out. There are many caveats, bugs, and missing pieces. Stay tuned!
 - ☐ Integer division and mod have been corrected for rounding in the presence of negative numbers.
 - ☐ Add scalar types (`UI8`, `SI32`, `F32`, `F64`, etc.) which are aliases to `SIMD[1, type]`.

March 2023

Week of 2023-03-27

- ☐ Parameter names are no longer load-bearing in function signatures. This gives more flexibility in defining higher-order functions, because the functions passed as parameters do not need their parameter names to match.

```
# Define a higher-order function...
fn generator[
    func: __mlir_type['!kgen.signature<', Int, '>() -> !kgen.none`]
]():
    pass

# Int parameter is named "foo".
fn f0[foo: Int]() {
    pass

# Int parameter is named "bar".
fn f1[bar: Int]() {
    pass

fn main():
    # Both can be used as `func`!
    generator[f0]()
    generator[f1()]
```

Stay tuned for improved function type syntax...

- ☐ Two magic operators, named `__get_lvalue_as_address(x)` and `__get_address_as_lvalue` convert stored LValues to and from `!kgen.pointer` types (respectively). This is most useful when using the `Pointer[T]` library type. The `Pointer.address_of(lvalue)` method uses the first one internally. The second one must currently be used explicitly, and can be used to project a pointer to a reference that you can pass around and use as a self value, for example:

```
# "Replacement new" SomeHeavy value into the memory pointed to by a
# Pointer[SomeHeavy].
__get_address_as_lvalue(somePointer.value) = SomeHeavy(4, 5)
```

Note that `__get_address_as_lvalue` assumes that there is already a value at the specified address, so the assignment above will run the `SomeHeavy` destructor (if any) before reassigning over the value.

- The `((x))` syntax is `__mlir_op` has been removed in favor of `__get_lvalue_as_address` which solves the same problem and is more general.
- ☐ When using a mutable `self` argument to a struct `__init__` method, it now must be declared with `&`, like any other mutable method. This clarifies the mutation model by making `__init__` consistent with other mutating methods.
- ☐ Add variadic string join function.
- ☐ Default initialize values with 0 or null if possible.
- ☐ Add compressed, aligned, and mask store intrinsics.

Week of 2023-03-20

- Initial String type is added to the standard library with some very basic methods.
 - Add DimList to remove the need to use an MLIR list type throughout the standard library.
 - The __clone__ method for copying a value is now named __copy__ to better follow Python term of art.
 - The __copy__ method now takes its self argument as a “borrowed” value, instead of taking it by reference. This makes it easier to write, works for @register_passable types, and exposes more optimization opportunities to the early optimizer and dataflow analysis passes.
- # Before:
fn __clone__(inout self) -> Self: ...
- # After:
fn __copy__(self) -> Self: ...
- A new @register_passable("trivial") may be applied to structs that have no need for a custom __copy__ or __del__ method, and whose state is only made up of @register_passable("trivial") types. This eliminates the need to define __copy__ boilerplate and reduces the amount of IR generated by the compiler for trivial types like Int.
 - You can now write back to attributes of structs that are produced by a computed lvalue expression. For example a[i].x = .. works when a[i] is produced with a __getitem__ / __setitem__ call. This is implemented by performing a read of a[i], updating the temporary, then doing a writeback.
 - The remaining hurdles to using non-parametric, @register_passable types as parameter values have been cleared. Types like Int should enjoy full use as parameter values.
 - Parameter pack inference has been added to function calls. Calls to functions with parameter packs can now elide the pack types:

```
fn foo[*Ts: AnyType](args: *Ts): pass  
foo(1, 1.2, True, "hello")
```

Note that the syntax for parameter packs has been changed as well.

- Add the runtime string type.
- Introduce the DimList struct to remove the need to use low-level MLIR operations.

Week of 2023-03-13

- Initializers for structs now use __init__ instead of __new__, following standard practice in Python. You can write them in one of two styles, either traditional where you mutate self:

```
fn __init__(self, x: Int):  
    self.x = x
```

or as a function that returns an instance:

```
fn __init__(x: Int) -> Self:  
    return Self {x: x}
```

Note that @register_passable types must use the later style.

- The default argument convention is now the borrowed convention. A “borrowed” argument is passed like a C++ const& so it doesn’t need to invoke the copy constructor (aka the __clone__ method) when passing a value to the function. There are two differences from C++ const&:

1. A future borrow checker will make sure there are no mutable aliases with an immutable borrow.
2. @register_passable values are passed directly in an SSA register (and thus, usually in a machine register) instead of using an extra reference wrapper. This is more efficient and is the ‘right default’ for @register_passable values like integers and pointers.

This also paves the way to remove the reference requirement from __clone__ method arguments, which will allow us to fill in more support for them.

- Support for variadic pack arguments has been added to Mojo. You can now write heterogeneous variadic packs like:

```
fn foo[*Ts: AnyType](args*: Ts): pass  
foo[Int, F32, String, Bool](1, 1.5, "hello", True)
```

- The owned argument convention has been added. This argument convention indicates that the function takes ownership of the argument and is responsible for managing its lifetime.
- The borrowed argument convention has been added. This convention signifies the callee gets an immutable shared reference to a value in the caller’s context.
- Add the getenv function to the os module to enable getting environment variables.
- Enable the use of dynamic strides in NDBuffer.

Week of 2023-03-06

- Support added for using capturing async functions as parameters.
- Returning result parameters has been moved from return statements to a new param_return statement. This allows returning result parameters from throwing functions:

```
@raises
fn foo() -> out: Int():
    param_return[42]
    raise Error()
```

And returning different parameters along @parameter if branches:

```
fn bar[in: Bool -> out: Int]()@parameter:
    if in:
        param_return[1]
    else:
        param_return[2]
```

- Mojo now supports omitting returns at the end of functions when they would not be reachable. For instance,

```
fn foo(cond: Bool) -> Int:
    if cond:
        return 0
    else:
        return 1

fn bar() -> Int:
    while True:
        pass
```

- String literals now support concatenation, so "hello " "world" is treated the same as "hello world".
- Empty bodies on functions, structs, and control flow statements are no longer allowed. Please use `pass` in them to explicitly mark that they are empty, just like in Python.
- Structs in Mojo now default to living in memory instead of being passed around in registers. This is the right default for generality (large structures, structures whose pointer identity matters, etc) and is a key technology that enables the borrow model. For simple types like `Int` and `SIMD`, they can be marked as `@register_passable`.

Note that memory-only types currently have some limitations: they cannot be used in generic algorithms that take and return a `!mlir_type` argument, and they cannot be used in parameter expressions. Because of this, a lot of types have to be marked `@register_passable` just to work around the limitations. We expect to enable these use-cases over time.

- Mojo now supports computed lvalues, which means you can finally assign to subscript expressions instead of having to call `__setitem__` explicitly.

Some details on this: Mojo allows you to define multiple `__setitem__` overloads, but will pick the one that matches your `__getitem__` type if present. It allows you to pass computed lvalues into inout arguments by introducing a temporary copy of the value in question.

- Mojo now has much better support for using register-primary struct types in parameter expressions and as the types of parameter values. This will allow migration of many standard library types away from using bare MLIR types like `_mlir_type.index` and towards using `Int`. This moves us towards getting rid of MLIR types everywhere and makes struct types first-class citizens in the parameter system.
- Add a `sort` function.
- Add non-temporal store to enable cache bypass.

February 2023

Week of 2023-02-27

- The `@interface`, `@implements`, and `@evaluator` trio of decorators have been removed, replaced by the `@parameter if` and `@adaptive` features.
- Parameter inference can now infer the type of variadic lists.
- Memory primary types are now supported in function results. A result slot is allocated in the caller, and the callee writes the result of the function into that slot. This is more efficient for large types that don't fit into registers neatly! And initializers for memory-primary types now initialize the value in-place, instead of emitting a copy!
- Support for `let` decls of memory primary types has been implemented. These are constant, ready-only values of memory primary types but which are allocated on the function stack.
- Overload conversion resolution and parameter inference has been improved:
 - Inference now works with `let` decls in some scenarios that weren't working before.
 - Parameter bindings can now infer types into parameter expressions. This helps resolve higher-order functions in parameter expressions.
- Optimize `floor`, `ceil`, and `ldexp` on X86 hardware.
- Implement the log math function.

Week of 2023-02-20

- A new `__memory_primary` struct decorator has been introduced. Memory primary types must always have an address. For instance, they are always stack-allocated when declared in a function and their values are passed into function calls by address instead of copy. This is in contrast with register primary types that may not have an address, and which are passed by value in function calls. Memory-primary fields are not allowed inside register-primary structs, because struct elements are stored in-line.

- ☐ A new `_CompilerBuiltin` module was added. This module defines core types and functions of the language that are referenced by the parser, and hence, is auto-imported by all other modules. For example new types for literal values like the boolean `True`/`False` will be included in `_CompilerBuiltin`.
- ☐ A special `__adaptive_set` property can be accessed on a function reference marked as `@adaptive`. The property returns the adaptive overload set of that function. The return type is a `!kgen.variadic`. This feature is useful to implement a generic `evaluate` function in the standard library.
- ☐ A new built-in literal type `BoolLiteral` was added in `_CompilerBuiltin`. It represents the literal boolean values `True` and `False`. This is the first Mojo literal to be emitted as a standard library type!
- ☐ Add the prefetch intrinsic to enable HW prefetching a cache line.
- ☐ Add the `InlinedFixedVector`, which is optimized for small vectors and stores values on both the stack and the heap.

Week of 2023-02-13

- Unqualified lookups of struct members apply contextual parameters. This means for instance that you can refer to static methods without binding the struct parameters.

```
struct Foo[x: Int]:
    @staticmethod
    bar(): pass

    foo(self):
        bar()          # implicitly binds to Foo[x].bar()
        Foo[2].bar()  # explicitly bind to another parameter
```

- ☐ A new `Self` type refers to the enclosing type with all parameters bound to their current values. This is useful when working with complex parametric types, e.g.:

```
struct MyArray[size: Int, element_type: type]:
    fn __new__() -> Self:
        return Self {...}
```

which is a lot nicer than having to say `MyArray[size, element_type]` over and over again.

- ☐ Mojo now supports an `@adaptive` decorator. This decorator will supersede interfaces, and it represents an overloaded function that is allowed to resolve to multiple valid candidates. In that case, the call is emitted as a fork, resulting in multiple function candidates to search over.

```
@adaptive
fn sort(arr: ArraySlice[Int]):
    bubble_sort(arr)

@adaptive
fn sort(arr: ArraySlice[Int]):
    merge_sort(arr)

fn concat_and_sort(lhs: ArraySlice[Int], rhs: ArraySlice[Int]):
    let arr = lhs + rhs
    sort(arr) # this forks compilation, creating two instances
              # of the surrounding function
```

- ☐ Mojo now requires that types implement the `__clone__` special member in order to copy them. This allows the safe definition of non-copyable types like `Atomic`. Note that Mojo still doesn't implement destructors, and (due to the absence of non-mutable references) it doesn't actually invoke the `__clone__` member when copying a `let` value. As such, this forces to you as a Mojo user to write maximal boilerplate without getting much value out of it.

In the future, we will reduce the boilerplate with decorators, and we will actually start using it. This will take some time to build out though.

- ☐ A special `__mlir_region` statement was added to provide stronger invariants around defining MLIR operation regions in Mojo. It has similar syntax to function declarations, except it there are no results and no input conventions.
- ☐ Implement the log math function.
- ☐ Improve the `DType` struct to enable compile-time equality checks.
- ☐ Add the `Complex` struct class.

Week of 2023-02-06

- ☐ The `if` statement now supports a `@parameter` decorator, which requires its condition to be a parameter expression, but which only emits the 'True' side of the condition to the binary, providing a "static if" functionality. This should eliminate many uses of `@interface` that are just used to provide different constraint on the implementations.
- ☐ `fn main()`: is now automatically exported and directly runnable by the command-line `mojo` tool. This is a stop-gap solution to enable script-like use cases until we have more of the language built out.
- ☐ The `@nodebug_inline` feature has been removed, please use `@alwaysinline("nodebug")` for methods that must be inlined and that we don't want to step into.
- ☐ Python chained comparisons, ex. `a < b < c`, are now supported in Mojo.
- ☐ Functions can now be defined with default argument values, such as `def f(x: Int, y: Int = 5):`. The default argument value is used when callers do not provide a value for that argument: `f(3)`, for example, uses the default argument value of `y = 5`.
- Unused coroutine results are now nicely diagnosed as "missing await" warnings.

- ☐ Introduce a vectorized reduction operations to the SIMD type.

January 2023

Week of 2023-01-30

- A basic Mojo language server has been added to the VS Code extension, which parses your code as you write it, and provides warnings, errors, and fix-it suggestions!
- ☐ The Mojo standard library is now implicitly imported by default.
- The coroutine lowering support was reworked and a new `Coroutine[T]` type was implemented. Now, the result of a call to an `async` function MUST be wrapped in a `Coroutine[T]`, or else memory will leak. In the future, when Mojo supports destructors and library types as literal types, the results of `async` function calls will automatically wrapped in a `Coroutine[T]`. But today, it must be done manually. This type implements all the expected hooks, such as `_await_`, and `get()` to retrieve the result. Typical usage:

```
async fn add_three(a: Int, b: Int, c: Int) -> Int:
    return a + b + c

async fn call_it():
    let task: Coroutine[Int] = add_three(1, 2, 3)
    print(await task)
```

- ☐ We now diagnose unused expression values at statement context in `fn` declarations (but not in `defs`). This catches bugs with unused values, e.g. when you forget the parens to call a function.
- ☐ An `@always_inline("nodebug")` function decorator can be used on functions that need to be force inlined, but when they should not have debug info in the result. This should be used on methods like `Int.__add__` which should be treated as builtin.
- ☐ The `@export` decorator now supports an explicit symbol name to export to, for example:

```
@export("baz") # exported as 'baz'
fn some_mojo_fn_name():
```

- ☐ Subscript syntax is now wired up to the `__getitem__` dunder method.

This allows type authors to implement the `__getitem__` method to enable values to be subscripted. This is an extended version of the Python semantics (given we support overloading) that allows you to define N indices instead of a single version that takes a tuple (also convenient because we don't have tuples yet).

Note that this has a very, very important limitation: subscripts are NOT wired up to `__setitem__` yet. This means that you can read values with `.. = v[i]` but you cannot store to them with `v[i] = ...` For this, please continue to call `__setitem__` directly.

- ☐ Function calls support parameter inference.

For calls to functions that have an insufficient number of parameters specified at the callsite, we can now infer them from the argument list. We do this by matching up the parallel type structure to infer what the parameters must be.

Note that this works left to right in the parameter list, applying explicitly specified parameters before trying to infer new ones. This is similar to how C++ does things, which means that you may want to reorder the list of parameters with this in mind. For example, a `dyn_cast`-like function will be more elegant when implemented as:

```
fn dyn_cast[DstType: type, SrcType: type](src: SrcType) -> DstType:
```

Than with the `SrcType/DstType` parameters flipped around.

- ☐ Add the growable Dynamic vector struct.

Week of 2023-01-23

- Inplace operations like `+=/__iadd__` may now take `self` by-val if they want to, instead of requiring it to be by-ref.
- ☐ Inplace operations are no longer allowed to return a non-None value. The corresponding syntax is a statement, not an expression.
- A new `TaskGroup` type was added to the standard library. This type can be used to schedule multiple tasks on a multi-threaded workqueue to be executed in parallel. An `async` function can await all the tasks at once with the `taskgroup`.
- ☐ We now support for loops! A type that defines an `__iter__` method that returns a type that defines `__next__` and `__len__` methods is eligible to be used in the statement `for el in X()`. Control flow exits the loop when the length is zero.

This means things like this now work:

```
for item in range(start, end, step):
    print(item)
```

- Result parameters now have names. This is useful for referring to result parameters in the return types of a function:

```
fn return_simd() -> nelts: Int() -> SIMD[f32, nelts]:
```

- ☐ We now support homogeneous variadics in value argument lists, using the standard Python `fn thing(*args: Int):` syntax! Variadics also have support in parameter lists:

```
fn variadic_params_and_args[*a: Int](*b: Int):
    print(a[0])
    print(b[1])
```

- ☐ Add the range struct to enable for ... range(...) loops.

- ☐ Introduce the unroll generator to allow one to unroll loops via a library function.

Week of 2023-01-16

- ☐ Struct field references are now supported in parameter context, so you can use `someInt.value` to get the underlying MLIR thing out of it. This should allow using first-class types in parameters more widely.
- ☐ We now support “pretty” initialization syntax for structs, e.g.:

```
struct Int:
    var value: __mlir_type.index
    fn __new__(value: __mlir_type.index) -> Int:
        return Int {value: value}()
```

This eliminates the need to directly use the MLIR `lit.struct.create` op in struct initializers. This syntax may change in the future when ownership comes in, because we will be able to support the standard `__init__` model then.

- ☐ It is now possible to attach regions to `__mlir_op` operations. This is done with a hack that allows an optional `_region` attribute that lists references to the region bodies (max 1 region right now due to lack of list [] literal).
- Nested functions now parse, e.g.:

```
fn foo():
    fn bar():
        pass
    bar()
```

- Python-style `async` functions should now work and the `await` expression prefix is now supported. This provides the joy of `async/await` syntactic sugar when working with asynchronous functions. This is still somewhat dangerous to use because we don’t have proper memory ownership support yet.
- String literals are now supported.
- Return processing is now handled by a dataflow pass inside the compiler, so it is possible to return early out of if statements.
- The parser now supports generating ‘fixit’ hints on diagnostics, and uses them when a dictionary literal uses a colon instead of equal, e.g.:

```
x.mojo:8:48: error: expected ':' in subscript slice, not '='
    return __mlir_op.'lit.struct.create'[value = 42]()
               ^
:
```

- ☐ Add reduction methods which operate on buffers.
- ☐ Add more math functions like sigmoid, sqrt, rsqrt, etc.
- ☐ Add partial load / store which enable loads and stores that are predicated on a condition.

Week of 2023-01-09

- The `/` and `*` markers in function signatures are now parsed and their invariants are checked. We do not yet support keyword arguments yet though, so they aren’t very useful.
- Functions now support a new `@nodebug_inline` decorator. (Historical note: this was later replaced with `@alwaysinline("nodebug")`).

Many of the things at the bottom level of the Mojo stack are trivial zero-abstraction wrappers around MLIR things, for example, the `+` operator on `Int` or the `__bool__` method on `Bool` itself. These operators need to be force inlined even at `-O0`, but they have some additional things that we need to wrestle with:

1. In no case would a user actually want to step into the `__bool__` method on `Bool` or the `+` method on `Int`. This would be terrible debugger QoL for unless you’re debugging `Int` itself. We need something like `__always_inline__`, `__nodebug__` attributes that clang uses in headers like `xmmmintrin.h`.
2. Similarly, these “operators” should be treated by users as primitives: they don’t want to know about MLIR or internal implementation details of `Int`.
3. These trivial zero abstraction things should be eliminated early in the compiler pipeline so they don’t slow down the compiler, bloating out the call graph with trivial leaves. Such thing slows down the elaborator, interferes with basic MLIR things like `fold()`, bloats out the IR, or bloats out generated debug info.
4. In a parameter context, we want some of these things to get inlined so they can be simplified by the attribute logic and play more nicely with canonical types. This is just a nice to have thing those of us who have to stare at generated IR.

The solution to this is a new `@nodebug_inline` decorator. This decorator causes the parser to force-inline the callee instead of generating a call to it. While doing so, it gives the operations the location of the call itself (that’s the “nodebug” part) and strips out let decls that were part of the internal implementation details.

This is a super-power-user-feature intended for those building the standard library itself, so it is intentionally limited in power and scope: It can only be used on small functions, it doesn’t support regions, by-ref, throws, async, etc.

- Separately, we now support an `@alwaysInline` decorator on functions. This is a general decorator that works on any function, and indicates that the function must be inlined. Unlike `@nodebug_inline`, this kind of inlining is performed later in the compilation pipeline.
- The `__include` hack has been removed now that we have proper import support.
- `__mlir_op` can now get address of l-value:

You can use magic (((x))) syntax in `_mlir_op` that forces the `x` expression to be an lvalue, and yields its address. This provides an escape hatch (isolated off in `_mlir_op` land) that allows unsafe access to lvalue addresses.

- We now support `_rlshift_` and `_rtruediv_`.
- □ The parser now resolves scoped alias references. This allows us to support things like `SomeType.someAlias`, forward substituting the value. This unblocks use of aliases in types like `DTyep`. We'd like to eventually preserve the reference in the AST, but this unblocks library development.
- □ Add a `now` function and `Benchmark` struct to enable timing and benchmarking.
- □ Move more of the computation in `NDBuffer` from runtime to compile time if possible (e.g. when the dimensions are known at compile time).

Week of 2023-01-02

- □ Added the `print` function which works on Integers and SIMD values.
- The frontend now has a new diagnostic subsystem used by the `kgen` tool (but not by `kgen-translate` for tests) that supports source ranges on diagnostics. Before we'd emit an error like:

```
x.mojo:13:3: error: invalid call to 'callee': in argument #0, value of type '$F32::F32' cannot be converted to expected type '$int::Int'
  callee(1.0+F32(2.0))
  ^
x.lit:4:1: note: function declared here
fn callee(a: Int):
^
```

now we produce:

```
x.mojo:13:3: error: invalid call to 'callee': in argument #0, value of type '$F32::F32' cannot be converted to expected type '$int::Int'
  callee(1.0+F32(2.0))
  ^
  ~~~~~
x.lit:4:1: note: function declared here
fn callee(a: Int):
^
```

- □ Parameter results are now supported in a proper way. They are now forward declared with an alias declaration and then bound in a call with an arrow, e.g.:

```
alias a : __mlir_type.index
alias b : __mlir_type.index
idx_result_params[xyz*2 -> a, b]()
```

- Various minor issues with implicit conversions are fixed. For instances, implicit conversions are now supported in parameter binding contexts and alias declarations with explicit types.
- Doc strings are allowed on functions and structs, but they are currently discarded by the parser.
- □ Add a `print` method!!!
- □ Demonstrate a naive matmul in Mojo.
- □ Initial work on functions that depend on types (e.g. `FPUtols`, `nan`, `inf`, etc.)
- □ Allow one to query hardware properties such as `simd_width`, `os`, etc. via `TargetInfo` at compile time.

December 2022

Week of 2022-12-26

- □ You can now call functions in a parameter context! Calling a function in a parameter context will evaluate the function at compile time. The result can then be used as parameter values. For example,

```
fn fma(x: Int, y: Int, z: Int) -> Int:
  return a + b * c

fn parameter_call():
  alias nelts = fma(32, 2, 16)
  var x: SIMD[f32, nelts]
```

- You can now disable printing of types in an `_mlir_attr` substitution by using unary + expression.
- □ `let` declarations are now supported in functions. `let` declarations are local run-time constant values, which are always rvalues. They complement 'var' decls (which are mutable lvalues) and are the normal thing to use in most cases. They also generate less IR and are always in SSA form when initialized.

We will want to extend this to support 'let' decls in structs at some point and support lazy initialized 'let' declarations (using dataflow analysis) but that isn't supported yet.

- □ Add the `NDBuffer` struct.
- Happy new year.

Week of 2022-12-19

- □ Start of the Standard library:
 1. Added Integer and SIMD structs to bootstrap the standard library.
 2. Added very basic buffer data structure.

- We have basic support for parsing parameter results in function calls! Result parameters are an important Mojo metaprogramming feature. They allow functions to return compile-time constants.

```
fn get_preferred_simdwidthof() -> nelts: Int():
    return[2]
```

```
fn vectorized_function():
    get_preferred_simdwidthof() -> nelts]()
    var x: SIMD[f32, nelts] =
```

- Types can now be used as parameters of `!kgen.mlirtype` in many more cases.
- MLIR operations with zero results don't need to specify `_type: []` anymore.
- We support parsing triple quoted strings, for writing docstrings for your functions and structs!
- A new `_mlir_type[a,b,c]` syntax is available for substituting into MLIR types and attributes is available, and the old placeholder approach is removed. This approach has a few advantages beyond what placeholders do:

1. It's simpler.
2. It doesn't form the intermediate result with placeholders, which gets rejected by MLIR's semantic analysis, e.g. the complex case couldn't be expressed before.
3. It provides a simple way to break long attrs/types across multiple lines.

- We now support an `@evaluator` decorator on functions for KGEN evaluators. This enables specifying user-defined interface evaluators when performing search during compilation.
- `import` syntax is now supported!

This handles packaging imported modules into file ops, enables effective isolation from the other decls. "import" into the desired context is just aliasing decls, with the proper symbols references handle automatically during IR generation. As a starting point, this doesn't handle any notion of packages (as those haven't been sketched out enough).

- Reversed binary operators (like `_radd_`) are now looked up and used if the forward version (like `_add_`) doesn't work for some reason.
- Implicit conversions are now generally available, e.g. in assign statements, variable initializers etc. There are probably a few more places they should work, but we can start eliminating all the extraneous explicit casts from literals now.
- Happy Holidays

Week of 2022-12-12

- Function overloading now works. Call resolution filters candidate list according to the actual parameter and value argument specified at the site of the call, diagnosing an error if none of the candidates are viable or if multiple are viable and ambiguous. We also consider implicit conversions in overload look:

```
fn foo(x: Int): pass
fn foo(x: F64): pass

foo(Int(1)) # resolves to the first overload
foo(1.0)    # resolves to the second overload
foo(1)      # error: both candidates viable with 1 implicit conversion! =
```

- The short circuiting binary `and` and `or` expressions are now supported.
- Unary operator processing is a lot more robust, now handling the `not` expression and `~x` on `Bool`.
- The compiler now generates debug information for use with GDB/LLDB that describes variables and functions.
- The first version of the Mojo Visual Studio Code extension has been released! It supports syntax highlighting for Mojo files.
- The first version of the `Bool` type has landed in the new Mojo standard library!
- Implicit conversions are now supported in return statements.

Week of 2022-12-05

- "Discard" patterns are now supported, e.g. `_ = foo()`
- We now support implicit conversions in function call arguments, e.g. converting an `index` value to `Int` automatically. This eliminates a bunch of casts, e.g. the need to say `F32(1.0)` everywhere.

This is limited for a few reasons that will be improved later:

1. We don't support overloading, so lots of types aren't convertible from all the things they should be, e.g. you can't pass "1" to something that expects `F32`, because `F32` can't be created from `index`.
2. This doesn't "check to see if we can invoke `_new_`" it force applies it on a mismatch, which leads to poor QoL.
3. This doesn't fix things that need `radd`.

November 2022

Week of 2022-11-28

- We support the `True` and `False` keywords as expressions.
- A new `alias` declaration is supported which allows defining local parameter values. This will eventually subsume type aliases and other things as it gets built out.

- We now have end-to-end execution of Mojo files using the `kgen` tool! Functions exported with `@export` can be executed.
- We have `try-except-else` and `raise` statements and implicit error propagation! The error semantics are that `def` can raise by default, but `fn` must explicitly declare raising with a `@raises` decorator. Stub out basic `Error` type.
- The & sigil for by-ref arguments is now specified after the identifier. Postfix works better for ref and move operators on the expression side because it chains and mentally associates correctly: `thing.method().result^`. We don't do that yet, but align param decl syntax to it so that things won't be odd looking when we do. In practice this looks like:

```
def mutate_argument(a&: index):
    a = 25
```

Week of 2022-11-21

- The magic `index` type is gone. Long live `_mlir_type.index`.
- Implement parameter substitution into parametric `_mlir_type` decls. This allows us to define parametric opaque MLIR types with exposed parameters using a new "placeholder" attribute. This allows us to expose the power of the KGEN type parametric system directly into Mojo.
- Fully-parametric custom types can now be defined and work in Mojo, bringing together a lot of the recent work. We can write the SIMD type directly as a wrapper around the KGEN type, for example:

```
struct SIMD[dt: _mlir_type.`!kgen.dtype`, nelts: _mlir_type.index]:
    var value:
        _mlir_type.`!pop.simd<#lit<placeholder index>, #lit<placeholder !kgen.dtype>>`[nelts, dt]

fn __add__(self, rhs: SIMD[dt, nelts]) -> SIMD[dt, nelts]:
    return __mlir_op.`pop.add`(self.value, rhs.value)
```

Week of 2022-11-14

- Implement a magic `_mlir_type` declaration that can be used to access any MLIR type. E.g. `_mlir_type.f64`.
- Add an `fn` declaration. These are like `def` declarations, but are more strict in a few ways: they require type annotations on arguments, don't allow implicit variable declarations in their body, and make their arguments `rvalues` instead of `lvalues`.
- Implemented Swift-style backtick identifiers, which are useful for code migration where names may collide with new keywords.
- A new `_include` directive has been added that performs source-level textual includes. This is temporary until we have an `import` model.
- Implement IR generation for arithmetic operators like `+` and `*` in terms of the `_add_` and `_mul_` methods.
- Added support for `break` and `continue` statements, as well as early returns inside loops and conditionals!
- Implemented augmented assignment operators, like `+=` and `@=`.
- Mojo now has access to generating any MLIR operations (without regions) with a new `_mlir_op` magic declaration. We can start to build out the language's builtin types with this:

```
struct Int:
    var value: _mlir_type.index

fn __add__(self, rhs: Int) -> Int:
    return __mlir_op.`index.add`(self.value, rhs.value)
```

Attributes can be attached to the declaration with subscript `[]` syntax, and an explicit result type can be specified with a special `_type` attribute if it cannot be inferred. Attributes can be accessed via the `_mlir_attr` magic decl:

```
_mlir_op.`index.cmp`[
    _type: _mlir_type.i1,
    pred: _mlir_attr.`#index<cmp_predicate slt>`
](lhs, rhs)
```

- Improved diagnostics emissions with ranges! Now errors highlight the whole section of code and not just the first character.

Week of 2022-11-07

- Implemented the `@interface` and `@implements` decorators, which provide access to KGEN generator interfaces. A function marked as an `@interface` has no body, but it can be implemented by multiple other functions.

```
@interface
def add(lhs: index, rhs: index):

@implements(add)
def normal_add(lhs: index, rhs: index) -> index:
    return lhs + rhs

@implements(add)
def slow_add(lhs: index, rhs: index) -> index:
    wait(1000)
    return normal_add(lhs, rhs)
```

- Support for static struct methods and initializer syntax has been added. Initializing a struct with `Foo()` calls an implicitly static `_new_` method. This method should be used instead of `_init_` inside structs.

```
struct Foo:
    var value: index

def __new__() -> Foo:
```

```

var result: Foo
result.value = Foo.return_a_number() # static method!
return result

@staticmethod
def return_a_number() -> index:
    return 42

```

- Full by-ref argument support. It's now possible to define in-place operators like `__iadd__` and functions like `swap(x, y)` correctly.
- Implemented support for field extract from rvalues, like `x.value` where `x` is not an lvalue (var declaration or by-ref function argument).

October 2022

Week of 2022-10-31

- Revised return handling so that a return statement with no expression is syntax sugar for `return None`. This enables early exits in functions that implicitly return `None` to be cleaner:

```
def just_return():
    return
```

- Added support for parsing more expressions: if-else, bitwise operators, shift operators, comparisons, floor division, remainder, and matmul.
- The type of the `self` argument can now be omitted on member methods.

Week of 2022-10-24

- Added parser support for right-associativity and unary ops, like the power operator `a ** b ** c` and negation operator `-a`.
- Add support for `&expr` in Mojo, which allows denoting a by-ref argument in functions. This is required because the `self` type of a struct method is implicitly a pointer.
- Implemented support for parametric function declarations, such as:

```

struct SIMD[dt: DType, width: index]:
    fn struct_method(self: &SIMD[dt, width]):
        pass

def fancy_add[dt: DType, width: index](
    lhs: SIMD[dt, width], rhs: SIMD[dt, width]) -> index:
    return width

```

Week of 2022-10-17

- Added explicit variable declarations with `var`, for declaring variables both inside functions and structs, with support for type references. Added `index` as a temporary built-in type.

```
def foo(lhs: index, rhs: index) -> index:
    var result: index = lhs + rhs
    return result
```

- Implemented support for parsing struct declarations and references to type declarations in functions! In `def`, the type can be omitted to signal an object type.

```

struct Foo:
    var member: index

def bar(x: Foo, obj) -> index:
    return x.member

```

- Implemented parser support for `if` statements and `while` loops!

```

def if_stmt(c: index, a: index, b: index) -> index:
    var result: index = 0
    if c:
        result = a
    else:
        result = b
    return result

def while_stmt(init: index):
    while init > 1:
        init = init - 1

```

- Significantly improved error emission and handling, allowing the parser to emit multiple errors while parsing a file.

Week of 2022-10-10

- Added support for parsing integer, float, and string literals.
- Implemented parser support for function input parameters and results. You can now write parametric functions like,

```
def foo[param: Int](arg: Int) -> Int:
    result = param + arg
    return result
```

Week of 2022-10-03

- Added some basic parser scaffolding and initial parser productions, including trivial expressions and assignment parser productions.

- Implemented basic scope handling and function IR generation, with support for forward declarations. Simple functions like,

```
def foo(x: Int):
```

Now parse! But all argument types are hard-coded to the MLIR `index` type.

- Added IR emission for simple arithmetic expressions on builtin types, like `x + y`.

September 2022

Week of 2022-09-26

- Mojo's first patch to add a lexer was Sep 27, 2022.
- Settled on `[]` for Mojo generics instead of `<>`. Square brackets are consistent with Python generics and don't have the less than ambiguity other languages have.

Mojo community

Resources to share feedback, report issues, and chat.

Mojo is still very young, but we believe an active community and a strong feedback pipeline is key to its success.

We'd love to hear from you through the following community channels.

[Ask a question](#)

[See existing GitHub Discussion posts, ask new questions, and share your ideas.](#)

This is a forum for ideas and questions, moderated by the Modular team.

[Report an issue](#)

[Report bugs or other issues with the Mojo SDK or Mojo Playground.](#)

[Before reporting an issue, see the Mojo roadmap & sharp edges.](#)

[Chat on Discord](#)

[Join our realtime chat on Discord to discuss the Mojo language and tools with the community.](#)

This is a community space where you can chat with other Mojo developers in realtime.

Why Mojo

A backstory and rationale for why we created the Mojo language.

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.

To see all the features we've built with Mojo so far, see the [Mojo programming manual](#).

Mojo

A new programming language that bridges the gap between research and production by combining the best of Python with systems and metaprogramming.

Mojo is a new programming language that bridges the gap between research and production by combining the best of Python syntax with systems programming and metaprogramming. With Mojo, you can write portable code that's faster than C and seamlessly inter-op with the Python ecosystem.

Mojo is now available for local development! □

[Get the Mojo SDK](#)

[Join our community](#)

Docs

[Get started with Mojo](#)

[Get the Mojo SDK or try coding in the Mojo Playground.](#)

[Why Mojo](#)

[A backstory and rationale for why we created the Mojo language.](#)

[Mojo programming manual](#)

[A tour of major Mojo language features with code examples.](#)

[Mojo modules](#)

[A list of all modules in the current standard library.](#)

[Mojo notebooks](#)

[All the Jupyter notebooks we've created for the Mojo Playground.](#)

[Mojo roadmap & sharp edges](#)

[A summary of our Mojo plans, including upcoming features and things we need to fix.](#)

[Mojo FAQ](#)

[Answers to questions we expect about Mojo.](#)

[Mojo changelog](#)

[A history of significant Mojo changes.](#)

[Mojo community](#)

[Resources to share feedback, report issues, and chat.](#)

No matching items