

# Mojo changelog

This is a list of changes to the Mojo language, standard library, and tools.

To check your current version, run `mojo --version`. To update the version of Mojo for your project with the `magic` package manager, follow the instructions in [Update a package](#) to update the `max` package.

## Switch to Magic

The `modular` command-line tool is deprecated (see [how to uninstall it](#)). We recommend that you now [manage your packages with `magic`](#), but you can also [use `conda`](#).

## v24.5 (2024-09-13)

### Highlights

Here's a brief summary of some of the major changes in this release, with more detailed information in the following sections:

- Mojo now supports Python 3.12 interoperability.
- The set of automatically imported entities (types, aliases, functions) into users' Mojo programs has been dramatically reduced. This can break existing user code as users will need to explicitly import what they're using for cases previously automatically included before.
- [print\(\)](#) now requires that its arguments conform to the [Formattable](#) trait. This enables efficient stream-based writing by default, avoiding unnecessary intermediate String heap allocations.
- The new builtin [input\(\)](#) function prints an optional prompt and reads a line from standard input, in the same way as Python.
- Mojo now allows implicit definitions of variables within a `fn` in the same way that has been allowed in a `def`. The `var` keyword is still allowed, but is now optional.
- Mojo now diagnoses "argument exclusivity" violations due to aliasing references. Mojo requires references (including implicit references due to `borrowed` / `inout` arguments) to be uniquely referenced (non-aliased) if mutable. This is a warning in the 24.5 release, but will be upgraded to an error in subsequent releases.
- Mojo now supports "conditional conformances" where some methods on a struct have additional trait requirements that the struct itself doesn't.

- `DTypePointer`, `LegacyPointer`, and `Pointer` have been removed. Use [UnsafePointer](#) instead. Functions that previously took a `DTypePointer` now take an equivalent `UnsafePointer`. For more information on using pointers, see [Unsafe pointers](#) in the Mojo Manual.
- There are many new standard library APIs, with new features for strings, collections, and interacting with the filesystem and environment. Changes are listed in the standard library section.
- The VS Code extension now supports a vendored MAX SDK for VS Code, which is automatically downloaded by the extension and it's used for all Mojo features, including the Mojo Language Server, the Mojo debugger, the Mojo formatter, and more.
- [mojo test](#) now uses the Mojo compiler for running unit tests. This will resolve compilation issues that sometimes appeared, and will also improve overall test execution times.

## Language changes

- Mojo now allows implicit definitions of variables within a `fn` in the same way that has been allowed in a `def`. The `var` keyword is still allowed and still denotes the declaration of a new variable with a scope (in both `def` and `fn`). Relaxing this makes `fn` and `def` more similar, but they still differ in other important ways.
- Mojo now diagnoses "argument exclusivity" violations due to aliasing references. Mojo requires references (including implicit references due to `borrowed` / `inout` arguments) to be uniquely referenced (non-aliased) if mutable. This is important for code safety, because it allows the compiler (and readers of code) to understand where and when a value is mutated. It is also useful for performance optimization because it allows the compiler to know that accesses through immutable references cannot change behind the scenes. Here is an invalid example:

```
fn take_two_strings(a: String, inout b: String):
    # Mojo knows 'a' and 'b' cannot be the same string.
    b += a

fn invalid_access():
    var my_string = String()

    # error: passing `my_string` inout is invalid since it is also passed
    # borrowed.
    take_two_strings(my_string, my_string)
```

This is similar to [Swift exclusivity checking](#) and the [Rust language](#) sometimes known as "aliasing xor mutability". That said, the Mojo implementation details are somewhat different because lifetimes are embedded in types.

This is a warning in the 24.5 release, but will be upgraded to an error in subsequent releases.

Argument exclusivity is not enforced for register-passable types. They are passed by copy, so they don't form aliases.

- Mojo now supports "conditional conformances" where some methods on a struct have additional trait requirements that the struct itself doesn't. This is expressed through an explicitly declared `self` type:

```
struct GenericThing[Type: AnyType]: # Works with anything
  # Sugar for 'fn normal_method[Type: AnyType](self: GenericThing[Type]):'
  fn normal_method(self): ...

  # Just redeclare the requirements with more specific types:
  fn needs_move[Type: Movable](self: GenericThing[Type], owned val: Type):
    var tmp = val^ # Ok to move 'val' since it is Movable
    ...
fn usage_example():
  var a = GenericThing[Int]()
  a.normal_method() # Ok, Int conforms to AnyType
  a.needs_move(42)  # Ok, Int is movable

  var b = GenericThing[NonMovable]()
  b.normal_method() # Ok, NonMovable conforms to AnyType

  # error: argument type 'NonMovable' does not conform to trait 'Movable'
  b.needs_move(NonMovable())
```

Conditional conformance works with dunder methods and other things as well.

- As a specific form of "conditional conformances", initializers in a struct may indicate specific parameter bindings to use in the type of their `self` argument. For example:

```
@value
struct MyStruct[size: Int]:
  fn __init__(inout self: MyStruct[0]): pass
  fn __init__(inout self: MyStruct[1], a: Int): pass
  fn __init__(inout self: MyStruct[2], a: Int, b: Int): pass

def test(x: Int):
  a = MyStruct()      # Infers size=0 from 'self' type.
  b = MyStruct(x)     # Infers size=1 from 'self' type.
  c = MyStruct(x, x)  # Infers size=2 from 'self' type.
```

- Mojo now supports named result bindings. Named result bindings are useful for directly emplacing function results into the output slot of a function. This feature provides more flexibility and guarantees around emplacing the result of a function compared to "guaranteed" named return value optimization (NRVO). If a `@register_passable` result is bound to a name, the result value is made accessible as a mutable reference.

```
fn efficiently_return_string(b: Bool) -> String as output:
  if b:
```

```

        output = "emplaced!"
        mutate(output)
        return
    return "regular return"

```

If we used a temporary for `output` instead, we would need to move into the result slot, which wouldn't work if the result type was non-movable.

In a function with a named result, `return` may be used with no operand to signal an exit from the function, or it can be used normally to specify the return value of the function. The compiler will error if the result is not initialized on all normal exit paths from the function.

- `__setitem__()` now works with variadic argument lists such as:

```

struct YourType:
    fn __setitem__(inout self, *indices: Int, val: Int): ...

```

The Mojo compiler now always passes the "new value" being set using the last keyword argument of the `__setitem__()`, e.g. turning `yourType[1, 2] = 3` into `yourType.__setitem__(1, 2, val=3)`. This fixes [Issue #248](#).

- Mojo context managers used in regions of code that may raise no longer need to define a "conditional" exit function in the form of `fn __exit__(self, e: Error) -> Bool`. This function allows the context manager to conditionally intercept and handle the error and allow the function to continue executing. This is useful for some applications, but in many cases the conditional exit would delegate to the unconditional exit function `fn __exit__(self)`.

Concretely, this enables defining `with` regions that unconditionally propagate inner errors, allowing code like:

```

def might_raise() -> Int:
    ...

def foo() -> Int:
    with ContextMgr():
        return might_raise()
    # no longer complains about missing return

def bar():
    var x: Int
    with ContextMgr():
        x = might_raise()
    print(x) # no longer complains about 'x' being uninitialized

```

- `async` functions now support memory-only results (like `String`, `List`, etc.) and `raises`. Accordingly, both [Coroutine](#) and [RaisingCoroutine](#) have been changed to accept `AnyType` instead of `AnyTrivialRegType`. This means the result types of `async` functions do not need to be `Movable`.

```

async fn raise_or_string(c: Bool) raises -> String:
  if c:
    raise "whoops!"
  return "hello world!"

```

Note that `async` functions do not yet support indirect calls, `ref` results, and constructors.

- The [Reference](#) type (and many iterators) now use [infer-only parameters](#) to represent the mutability of their lifetime, simplifying the interface.
- The environment variable `MOJO_PYTHON` can be pointed to an executable to pin Mojo to a specific version:

```
$ export MOJO_PYTHON="/usr/bin/python3.11"
```

Or a virtual environment to always have access to those Python modules:

```
$ export MOJO_PYTHON="~/venv/bin/python"
```

`MOJO_PYTHON_LIBRARY` still exists for environments with a dynamic `libpython` but no Python executable.

- The pointer aliasing semantics of Mojo have changed. Initially, Mojo adopted a C-like set of semantics around pointer aliasing and derivation. However, the C semantics bring a lot of history and baggage that are not needed in Mojo and which complicate compiler optimizations. The language overall provides a stronger set of invariants around pointer aliasing with lifetimes and exclusive mutable references to values, etc.

It is now forbidden to convert a non-pointer-typed value derived from a Mojo-allocated pointer, such as an integer address, to a pointer-typed value. "Derived" means there is overlap in the bits of the non-pointer-typed value with the original pointer value. Accordingly, the [UnsafePointer](#) constructor that took an `address` keyword argument has been removed.

It is still possible to make this conversion in certain cases where it is absolutely necessary, such as interoperating with other languages like Python. In this case, the compiler makes two assumptions: any pointer derived from a non-pointer-typed value does not alias any Mojo-derived pointer and that any external function calls have arbitrary memory effects.

- `await` on a coroutine now consumes it. This strengthens the invariant that coroutines can be awaited only once.

## Standard library changes

- [builtin](#) package:
  - The set of automatically imported entities (types, aliases, functions) into users' Mojo programs has been dramatically reduced. Before, with the way the `builtin` module was handled, all of the entities in the following modules would be automatically included:

memory, sys, os, utils, python, bit, random, math, builtin, collections

Now, only the explicitly enumerated entities in `prelude/__init__.mojo` are the ones automatically imported into users' Mojo programs. This will break a lot of user code as users will need to explicitly import what they're using for cases previously commonly included before (such as [Optional](#), [Variant](#), and functions such as [abort\(\)](#), [alignof\(\)](#), [bitcast\(\)](#), [bitwidthof\(\)](#), [external\\_call\(\)](#), [simdwidthof\(\)](#), and [sizeof\(\)](#)).

- Some types from the `builtin` module have been moved to different modules for clarity which is made possible now that we have a `prelude` module that can re-export symbols from modules other than `builtin`.

In particular, the `builtin.string` module has been moved to [collections.string](#).

- Input and output:

- Added the builtin [input\(\)](#) function, which behaves the same as Python. ([PR #3392](#))

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

If the user enters "Mojo" it returns "Hello, Mojo!"

There is a known issue when running the `input()` function with JIT compilation (see issue [#3479](#)).

- [print\(\)](#) now requires that its arguments conform to the [Formattable](#) trait. This enables efficient stream-based writing by default, avoiding unnecessary intermediate String heap allocations.

Previously, `print()` required types conform to [Stringable](#). This meant that to execute a call like `print(a, b, c)`, at least three separate String heap allocations were down, to hold the formatted values of `a`, `b`, and `c` respectively. The total number of allocations could be much higher if, for example, `a.__str__()` was implemented to concatenate together the fields of `a`, like in the following example:

```
struct Point(Stringable):
    var x: Float64
    var y: Float64

    fn __str__(self) -> String:
        # Performs 3 allocations: 1 each for str(..) of each of the fields,
        # and then the final returned `String` allocation.
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

A type like the one above can transition to additionally implementing `Formattable` with the following changes:

```
struct Point(Stringable, Formattable):
    var x: Float64
    var y: Float64

    fn __str__(self) -> String:
```

```
return String.format_sequence(self)
```

```
fn format_to(self, inout writer: Formatter):  
    writer.write("(", self.x, ", ", self.y, ")")
```

In the example above, [String.format\\_sequence\(\)](#) is used to construct a `String` from a type that implements `Formattable`. This pattern of implementing a type's `Stringable` implementation in terms of its `Formattable` implementation minimizes boilerplate and duplicated code, while retaining backwards compatibility with the requirements of the commonly used `str()` function.

### Note

The error shown when passing a type that does not implement `Formattable` to `print()` is currently not entirely descriptive of the underlying cause:

```
error: invalid call to 'print': callee with non-empty variadic pack argument expects 0  
positional operands, but 1 was specified  
    print(point)  
    ~~~~~^~~~~~
```

If you see the above error, ensure that all argument types implement `Formattable`.

- [debug\\_assert\(\)](#) now also requires that its `message` argument conform to `Formattable`.
- Added [TemporaryDirectory](#) in module `tempfile`. ([PR 2743](#))
- Added [NamedTemporaryFile](#) in module `tempfile`. ([PR 2762](#))
- [String](#) and friends:
  - The `builtin.string` module has been moved to [collections.string](#).
  - Added the [String.format\(\)](#) method. ([PR #2771](#))

Supports automatic and manual indexing of `*args`.

Examples:

```
print(  
    String("{1} Welcome to {0} {1}").format("mojo", "🔥")  
)  
# 🔥 Welcome to mojo 🔥
```

```
print(String("{} {} {}").format(True, 1.125, 2))  
#True 1.125 2
```

- [String.format\(\)](#) now supports conversion flags `!s` and `!r`, allowing for `str()` and `repr()` conversions within format strings. ([PR #3279](#))

Example:

```
String("{} {}").format("Mojo", "Mojo")  
# "Mojo 'Mojo' "
```

```
String("{0!s} {0!r}").format("Mojo")  
# "Mojo 'Mojo' "
```

- The `String` class now has [`rjust\(\)`](#), [`ljust\(\)`](#), and [`center\(\)`](#) methods to return a justified string based on width and fillchar. ([PR #3278](#))
- The [`atol\(\)`](#) function now correctly supports leading underscores, (e.g. `atol("0x_ff", 0)`), when the appropriate base is specified or inferred (base 0). non-base-10 integer literals as per Python's [Integer Literals](#). ([PR #3180](#))
- Added the [`unsafe\_cstr\_ptr\(\)`](#) method to `String` and `StringLiteral`, which returns an `UnsafePointer[C_char]` for convenient interoperability with C APIs.
- Added the `byte_length()` method to [String](#), [StringSlice](#), and [StringLiteral](#) and deprecated their private `_byte_length()` methods. Added a warning to the [String.\\_\\_len\\_\\_\(\)](#) method that it will return the length in Unicode codepoints in the future and [StringSlice.\\_\\_len\\_\\_\(\)](#) now does return the Unicode codepoints length. ([PR #2960](#))
- Added a new [StaticString](#) type alias. This can be used in place of [StringLiteral](#) for runtime string arguments.
- Added a [StringSlice](#) initializer that accepts a `StringLiteral`.
- The [StringRef](#) constructors from `DTypePointer.int8` have been changed to take a `UnsafePointer[C_char]`, reflecting their use for compatibility with C APIs.
- Continued the transition to `UnsafePointer` and unsigned byte type for strings:
  - [String.unsafe\\_ptr\(\)](#) now returns an `UnsafePointer[UInt8]` (was `UnsafePointer[Int8]`)
  - [StringLiteral.unsafe\\_ptr\(\)](#) now returns an `UnsafePointer[UInt8]` (was `UnsafePointer[Int8]`)
- [UnsafePointer](#) and other reference type changes:
  - `DTypePointer`, `LegacyPointer`, and `Pointer` have been removed. Use [UnsafePointer](#) instead. For more information on using pointers, see [Unsafe pointers](#) in the Mojo Manual.

Functions that previously took a `DTypePointer` now take an equivalent `UnsafePointer`. A quick rule for conversion from `DTypePointer` to `UnsafePointer` is:

```
DTypePointer[type] -> UnsafePointer[Scalar[type]]
```

There could be places that you have code of the form:

```
fn f(ptr: DTypePointer):
```



which is equivalent to `DTypePointer[*_]`. In this case you would have to add an `infer-only` type parameter to the function:

```
fn f[type: DType, //](ptr: UnsafePointer[Scalar[type]]):
```

because we can't have an unbound parameter inside the struct.

There could also be places where you use `DTypePointer[Scalar[DType.invalid/index]]`, and it would be natural to change these to `UnsafePointer[NoneType/Int]`. But since these are not an `UnsafePointer` that stores a `Scalar`, you might have to `rebind/bitcast` to appropriate types.

- The `DTypePointer` [load\(\)](#) and [store\(\)](#) methods have been moved to `UnsafePointer`.
- `UnsafePointer` now supports [strided\\_load\(\)](#), [strided\\_store\(\)](#), [gather\(\)](#), and [scatter\(\)](#) when the underlying type is `Scalar[DType]`.
- The global functions for working with `UnsafePointer` have transitioned to being methods through the use of conditional conformances:
  - `destroy_pointee(p) => p.destroy\_pointee\(\)`
  - `move_from_pointee(p) => p.take\_pointee\(\)`
  - `initialize_pointee_move(p, value) => p.init\_pointee\_move\(value\)`
  - `initialize_pointee_copy(p, value) => p.init\_pointee\_copy\(value\)`
  - `move_pointee(src=p1, dst=p2) => p.move\_pointee\_into\(p2\)`
- The `UnsafePointer.offset()` method is deprecated and will be removed in a future release. Use [pointer arithmetic](#) instead.

```
new_ptr = ptr.offset(1)
```

Becomes:

```
new_ptr = ptr + 1
```

- `UnsafePointer` now has an [alignment](#) parameter to specify the static alignment of the pointer. Consequently, [UnsafePointer.alloc\(\)](#) no longer takes in an alignment parameter, and the alignment should be specified in the type.

```
UnsafePointer[type].alloc[alignment](x) # now becomes  
UnsafePointer[type, alignment].alloc(x)
```

- `UnsafePointer` has a new [exclusive: Bool = False](#) parameter. Setting this parameter to true tells the compiler that the user knows this pointer and all those derived from it have exclusive access to the underlying memory allocation. The compiler is not guaranteed to do anything with this information.

- It is no longer possible to cast (implicitly or explicitly) from `Reference` to `UnsafePointer`. Instead of `UnsafePointer(someRef)` please use the `UnsafePointer.address_of(someRef[])` which makes the code explicit that the `UnsafePointer` gets the address of what the reference points to.
- Python interoperability changes:
  - Mojo now supports Python 3.12 interoperability.
  - Creating a nested `PythonObject` from a list or tuple of Python objects is possible now:

```
var np = Python.import_module("numpy")
var a = np.array([1, 2, 3])
var b = np.array([4, 5, 6])
var arrays = PythonObject([a, b])
assert_equal(len(arrays), 2)
```

Also allowing more convenient call syntax:

```
var stacked = np.hstack((a, b))
assert_equal(str(stacked), "[1 2 3 4 5 6]")
```

([PR #3264](#))

- Accessing local Python modules with `Python.add_to_path(".")` is no longer required. It now behaves the same as Python. You can access modules in the same folder as the target file:
  - `mojo run /tmp/main.mojo` can access `/tmp/mymodule.py`
  - `mojo build main.mojo -o ~/myexe && ~/myexe` can access `~/mymodule.py`
- Collections:
  - `List` values are now equality comparable with `==` and `!=` when their element type is equality comparable. ([PR #3195](#))
  - `Optional` values are now equality comparable with `==` and `!=` when their element type is equality comparable.
  - Added a new `Counter` dictionary-like type, matching most of the features of the Python one. ([PR #2910](#))
  - `Dict` now implements `setdefault()`, which gets a value from the dictionary by key, or sets it to a default if it doesn't exist. ([PR #2803](#))
  - `Dict` now supports `popitem()`, which removes and returns the last item in the `Dict`. ([PR #2701](#))
  - Added a `Dict.__init__()` overload to specify initial capacity. ([PR #3171](#))

The capacity has to be a power of two and greater than or equal to 8.

It allows for faster initialization by skipping incremental growth steps.

Example:

```
var dictionary = Dict[Int,Int](power_of_two_initial_capacity = 1024)
# Insert (2/3 of 1024) entries
```

- `ListLiteral` now supports `__contains__()`. ([PR #3251](#))
- Filesystem and environment utilities:
  - [Path.home\(\)](#) has been added to return a path of the user's home directory.
  - [os.path.expanduser\(\)](#) and [pathlib.Path.expanduser\(\)](#) have been added to allow expanding a prefixed `~` in a `String` or `Path` with the user's home path:

```
import os
print(os.path.expanduser("~/modular"))
# /Users/username/.modular
print(os.path.expanduser("~/root/folder"))
# /var/root/folder (on macos)
# /root/folder      (on linux)
```

- [os.path.split\(\)](#) has been added for splitting a path into `head`, `tail`:

```
import os
head, tail = os.path.split("/this/is/head/tail")
print("head:", head)
print("tail:", tail)
# head: /this/is/head
# tail: tail
```

- [os.makedirs\(\)](#) and [os.removedirs\(\)](#) have been added for creating and removing nested directories:

```
import os
path = os.path.join("dir1", "dir2", "dir3")
os.path.makedirs(path, exist_ok=True)
os.path.removedirs(path)
```

- The [pwd](#) module has been added for accessing user information in `/etc/passwd` on POSIX systems. This follows the same logic as Python:

```
import pwd
import os
current_user = pwd.getpwuid(os.getuid())
print(current_user)

# pwd.struct_passwd(pw_name='jack', pw_passwd='*****', pw_uid=501,
# pw_gid=20, pw_gecos='Jack Clayton', pw_dir='/Users/jack',
# pw_shell='/bin/zsh')

print(current_user.pw_uid)
```

# 501

```
root = pwd.getpwnam("root")
print(root)
```

```
# pwd.struct_passwd(pw_name='root', pw_passwd='*', pw_uid=0, pw_gid=0,
# pw_gecos='System Administrator', pw_dir='/var/root', pw_shell='/bin/zsh')
```

- Other new traits and related features:

- Added the [ExplicitlyCopyable](#) trait to mark types that can be copied explicitly, but which might not be implicitly copyable.

This supports work to transition the standard library collection types away from implicit copyability, which can lead to unintended expensive copies.

- Added the [Identifiable](#) trait, used to describe types that implement the `__is__()` and `__isnot__()` trait methods. ([PR #2807](#))
- Types conforming to [Boolable](#) (that is, those implementing `__bool__()`) no longer implicitly convert to `Bool`. A new [ImplicitlyBoolable](#) trait is introduced for types where this behavior is desired.

- Miscellaneous:

- [NoneType](#) is now a normal standard library type, and not an alias for a raw MLIR type.

Function signatures written as `fn() -> NoneType` should transition to being written as `fn() -> None`.

- Mojo now has a [UInt](#) type for modeling unsigned (scalar) integers with a platform-dependent width. `UInt` implements most arithmetic operations that make sense for integers, with the notable exception of `__neg__()`. Builtin functions such as `min()/max()`, as well as math functions like `ceildiv()`, `align_down()`, and `align_up()` are also implemented for `UInt`.
- Now that we have a `UInt` type, use this to represent the return type of a hash. In general, hashes should be an unsigned integer, and can also lead to improved performance in certain cases.
- Added the [C\\_char](#) type alias in `sys ffi`.
- [sort\(\)](#) now supports a `stable` parameter. It can be called by

```
sort(cmp_fn, stable=True)(list)
```

The algorithm requires  $O(N)$  auxiliary memory. If extra memory allocation fails, the program crashes.

- `sort()` no longer takes `LegacyPointer` since that type is now removed.
- Added the [oct\(\)](#) builtin function for formatting an integer in octal. ([PR #2914](#))
- Added the [assert\\_is\(\)](#) and [assert\\_is\\_not\(\)](#) test functions to the `testing` module.
- The [math](#) package now includes the `pi`, `e`, and `tau` constants (Closes Issue [#2135](#)).
- The [ulp](#) function from `numerics` has been moved to the `math` module.

- `bit` module now supports [bit\\_reverse\(\)](#), [byte\\_swap\(\)](#), and [pop\\_count\(\)](#) for the `Int` type. ([PR #3150](#))
- A few `bit` functions have been renamed for clarity:
  - `countl_zero()` -> [count\\_leading\\_zeros\(\)](#)
  - `countr_zero()` -> [count\\_trailing\\_zeros\(\)](#)
- [Slice](#) now uses `OptionalReg[Int]` for `start` and `end` and implements a constructor which accepts optional values. `Slice._has_end()` has also been removed since a `Slice` with no end is now represented by an empty `Slice.end` option. ([PR #2495](#))

```
var s = Slice(1, None, 2)
print(s.start.value()) # must retrieve the value from the optional
```

- The `rank` argument for [algorithm.elementwise\(\)](#) is no longer required and is only inferred.
- The `time.now()` function has been deprecated. Please use [time.perf\\_counter\(\)](#) or [time.perf\\_counter\\_ns](#) instead.
- [SIMD](#) construction from `Bool` has been restricted to `DType.bool` data type.

## Tooling changes

- [mojo test](#) new features and changes:
  - `mojo test` now uses the Mojo compiler for running unit tests. This will resolve compilation issues that sometimes appeared, and will also improve overall test times, since we will only compile unit tests once before executing all of them.

These changes do not apply to doctests, due to their different semantics.

  - The `mojo test` command now accepts a `--filter` option that will narrow the set of tests collected and executed. The filter string is a POSIX extended regular expression.
  - The `mojo test` command now supports using the same compilation options as `mojo build`.
  - You can now debug unit tests using `mojo test` by passing the `--debug` flag. Most debug flags are supported; run `mojo test --help` for a full listing.

Debugging doctests is not currently supported.
- Mojo debugger new features and changes:
  - The `mojo debug --rpc` command has been renamed to [mojo debug --vscode](#), which is now able to manage multiple VS Code windows.
  - The Mojo debugger now supports a `break-on-raise` command that indicated the debugger to stop at any `raise` statements. A similar features has been added to the debugger on VS Code.
  - The Mojo debugger now hides the artificial function arguments `__result__` and `__error__` created by the compiler for Mojo code.

- VS Code support changes:
  - The VS Code extension now supports a vendored MAX SDK for VS Code, which is automatically downloaded by the extension and it's used for all Mojo features, including the Mojo Language Server, the Mojo debugger, the Mojo formatter, and more.
  - A proxy has been added to the Mojo Language Server on VS Code that handles crashes more gracefully.
- The Mojo Language Server no longer sets `.` as a commit character for auto-completion.

## Removed

- Support for the legacy `fn __init__(...) -> Self:` form has been removed from the compiler, please switch to using `fn __init__(inout self, ...):` instead.
- The builtin `tensor` module has been removed. Identical functionality is available in [max.tensor](#), but it is generally recommended to use structs from the [buffer](#) module when possible instead.
- Removed `String.unsafe_uint8_ptr()`. `String.unsafe_ptr()` now returns the same thing.
- Removed `StringLiteral.unsafe_uint8_ptr()` and `StringLiteral.as_uint8_ptr()`.
- Removed `SIMD.splat(value: Scalar[type])`. Use the constructor for `SIMD` instead.
- Removed the `SIMD.{add,mul,sub}_with_overflow()` methods.
- Removed the `SIMD.min()` and `SIMD.max()` methods. Identical functionality is available using the builtin [min\(\)](#) and [max\(\)](#) functions.
- Removed the Mojo Language Server warnings for unused function arguments.
- Run Mojo File in Dedicated Terminal action has been removed, and the action Run Mojo File will always open a dedicated terminal for each mojo file to guarantee a correct environment.

## Fixed

- Fixed a crash in the Mojo Language Server when importing the current file.
- Fixed crash when specifying variadic keyword arguments without a type expression in `def` functions, e.g.:

```
def foo(**kwargs): ... # now works
```

- Mojo now prints `ref` arguments and results in generated documentation correctly.
- [#1734](#) - Calling `__copyinit__` on self causes crash.
- [#3142](#) - [QoL] Confusing `__setitem__` method is failing with a "must be mutable" error.
- [#248](#) - [Feature] Enable `__setitem__` to take variadic arguments
- [#3065](#) - Fix incorrect behavior of `SIMD.__int__` on unsigned types

- [#3045](#) - Disable implicit SIMD conversion routes through `Bool`
- [#3126](#) - [BUG] List doesn't work at compile time.
- [#3237](#) - [BUG] Difference between `__getitem__` and `[.]` operator.
- [#3336](#) - Fix outdated references to `let` in REPL documentation.
- The VS Code extension no longer caches the information of the selected MAX SDK, which was causing issues upon changes in the SDK.
- The Mojo debugger now stops showing spurious warnings when parsing closures.

## Special thanks

Special thanks to our community contributors: [@jjvraw](#), [@artemiogr97](#), [@martinvuyk](#), [@jayzhan211](#), [@bgreni](#), [@mzaks](#), [@msaelices](#), [@rd4com](#), [@jiex-liu](#), [@kszucs](#), [@thatstoasty](#)

## v24.4 (2024-06-07)

### ✨ Highlights

Big themes for this release:

- Improvements to the performance and ease-of-use for `def` functions.
- Continued unification of standard library APIs around the `UnsafePointer` type.
- Many quality-of-life improvements for the standard library collection types.
- Significant performance improvements when inserting into a `Dict`. Performance on this metric is still not where we'd like it to be, but it is much improved.
- A new `@parameter_for` mechanism for expressing compile-time loops, which replaces the earlier (and less reliable) `@unroll` decorator.
- New Mojo Manual pages on [Control flow](#), [Testing](#) and using [unsafe pointers](#).

## Language changes

- Mojo has changed how `def` function arguments are processed. Previously, by default, arguments to a `def` were treated according to the `owned` convention, which makes a copy of the value, enabling that value to be mutable in the callee.

This could lead to major performance issues because of the proliferation of unnecessary copies. It also required you to declare non-copyable types as `borrowed` explicitly. Now Mojo takes a different approach: `def`

functions take arguments as borrowed by default (consistent with `fn` functions) but will make a local copy of the value **only if the argument is mutated** in the body of the function.

This improves consistency, performance, and ease of use.

- Implicit variable definitions in a `def` function are more flexible: you can now implicitly declare variables as the result of a tuple return, using `a, b, c = foo()`. For example:

```
def return_two(i: Int) -> (Int, Int):  
    return i, i+1  
  
a, b = return_two(5)
```

Implicit variable declarations can also now shadow global immutable symbols (such as module names and built-ins) without getting a compiler error. For example:

```
slice = foo()
```

- Mojo functions can return an auto-dereferenced reference to storage with a new `ref` keyword in the result type specifier. For example:

```
@value  
struct Pair:  
    var first: Int  
    var second: Int  
  
    fn get_first_ref(inout self) -> ref[__lifetime_of(self)] Int:  
        return self.first  
  
fn show_mutation():  
    var somePair = Pair(5, 6)  
    somePair.get_first_ref() = 1
```

This approach provides a general way to return an "automatically dereferenced" reference of a given type. Notably, this eliminates the need for `__refitem__()` to exist. `__refitem__()` has thus been removed and replaced with `__getitem__()` that returns a reference.

- Mojo added support for *infer-only parameters*. Infer-only parameters must appear at the beginning of the parameter list and cannot be explicitly specified by the user. They are declared to the left of a `//` marker, much like positional-only parameters. This allows programmers to define functions with dependent parameters to be called without the caller specifying all the necessary parameters. For example:

```
fn parameter_simd[dt: DType, //, value: Scalar[dt]]():  
    print(value)
```



```
fn call_it():
    parameter_simd[Int32(42)]()
```

In the above example, `Int32(42)` is passed directly into `value`, the first parameter that isn't infer-only. `dt` is inferred from the parameter itself to be `DType.int32`.

This also works with structs. For example:

```
struct ScalarContainer[dt: DType, //, value: Scalar[dt]]:
    pass

fn foo(x: ScalarContainer[Int32(0)]): # 'dt' is inferred as `DType.int32`
    pass
```

This should make working with dependent parameters more ergonomic. See [Infer-only parameters](#) in the Mojo Manual.

- Mojo now allows functions overloaded on parameters to be resolved when forming references to, but not calling, those functions. For example, the following now works:

```
fn overloaded_parameters[value: Int32]():
    pass

fn overloaded_parameters[value: Float32]():
    pass

fn form_reference():
    alias ref = overloaded_parameters[Int32()] # works!
```

- Mojo now supports adding a `@deprecated` decorator on structs, functions, traits, aliases, and global variables. The decorator marks the attached declaration as deprecated and causes a warning to be emitted when the deprecated declaration is referenced in user code. The decorator requires a deprecation message, specified as a string literal.

```
@deprecated("Foo is deprecated, use Bar instead")
struct Foo:
    pass

fn outdated_api(x: Foo): # warning: Foo is deprecated, use Bar instead
    pass

@deprecated("use another function!")
fn bar():
    pass

fn techdebt():
    bar() # warning: use another function!
```

- Mojo has introduced [@parameter for](#), a new feature for compile-time programming. `@parameter for` defines a for loop where the sequence and the induction values in the sequence must be parameter values. For example:

```
fn parameter_for[max: Int]():
  @parameter
  for i in range(max)
    @parameter
    if i == 10:
      print("found 10!")
```

Currently, `@parameter for` requires the sequence's `__iter__()` method to return a `_StridedRangeIterator`, meaning the induction variables must be `Int`. The intention is to lift these restrictions in the future.

- The `is_mutable` parameter of `Reference` and `AnyLifetime` is now a `Bool`, not a low-level `__mlir_type.i1` value.

This improves the ergonomics of spelling out a `Reference` type explicitly.

- Mojo will now link to a Python dynamic library based on the Python on top of your search path: `PATH`. This enables you to activate a virtual environment like `conda` and have access to Python modules installed in that environment without setting `MOJO_PYTHON_LIBRARY`. Previously Mojo would find a `libpython` dynamic library on installation and put the path in `.modular/modular.cfg`, which could result in version conflicts if you activated a virtual environment of a different Python version.
- `AnyRegType` has been renamed to `AnyTrivialRegType` and Mojo now forbids binding non-trivial register-passable types to `AnyTrivialRegType`. This closes a major safety hole in the language. Please use `AnyType` for generic code going forward.
- The `let` keyword has been completely removed from the language. We previously removed `let` declarations but still provided an error message to users. Now, it is completely gone from the grammar.

## Standard library changes

- New traits and related features:
  - Added built-in [repr\(\)](#) function and [Representable](#) trait. ([PR #2361](#))
  - Added the [Indexer](#) trait to denote types that implement the `__index__()` method which allows these types to be accepted in common `__getitem__()` and `__setitem__()` implementations, as well as allow a new built-in [index\(\)](#) function to be called on them. Most standard library containers can now be indexed by any type that implements `Indexer`. For example:

```
@value
struct AlwaysZero(Indexer):
  fn __index__(self) -> Int:
    return 0
```

```

struct MyList:
    var data: List[Int]

    fn __init__(inout self):
        self.data = List[Int](1, 2, 3, 4)

    fn __getitem__[T: Indexer](self, idx: T) -> Int:
        return self.data[index(idx)]

print(MyList()[AlwaysZero()]) # prints `1`

```

Types conforming to the `Indexer` trait are implicitly convertible to `Int`. This means you can write generic APIs that take `Int` instead of making them take a generic type that conforms to `Indexer`. For example:

```

@value
struct AlwaysZero(Indexer):
    fn __index__(self) -> Int:
        return 0

@value
struct Incrementer:
    fn __getitem__(self, idx: Int) -> Int:
        return idx + 1

var a = Incrementer()
print(a[AlwaysZero()]) # works and prints 1

```

([PR #2685](#))

- Added traits allowing user-defined types to be supported by various built-in and math functions.

Function	Trait	Required method
<a href="#">abs()</a>	<a href="#">Absable</a>	<code>__abs__()</code>
<a href="#">pow()</a>	<a href="#">Powable</a>	<code>__pow__()</code>
<a href="#">round()</a>	<a href="#">Roundable</a>	<code>__round__()</code>
<a href="#">math.ceil</a>	<code>math.Ceilable</code>	<code>__ceil__()</code>
<a href="#">math.ceildiv</a>	<code>math.CeilDivable</code> <code>math.CeilDivableRaising</code>	<code>__ceildiv__()</code>
<a href="#">math.floor</a>	<code>math.Floorable</code>	<code>__floor__()</code>

Function	Trait	Required method
<a href="#">math.trunc</a>	Truncable	<code>__trunc__()</code>

Notes:

- Conforming to the `Powable` trait also means that the type can be used with the power operator (`**`).
- For `ceildiv()`, structs can conform to either the `CeilDivable` trait or `CeilDivableRaising` trait.
- Due to ongoing refactoring, the traits `Ceilable`, `CeilDivable`, `Floorable`, and `Truncable` do not appear in the API reference. They should be imported from the `math` module, except for `Truncable` which is (temporarily) available as a built-in trait and does not need to be imported.

Example:

```
from math import sqrt

@value
struct Complex2(Absable, Roundable):
    var re: Float64
    var im: Float64

    fn __abs__(self) -> Self:
        return Self(sqrt(self.re * self.re + self.im * self.im), 0.0)

    fn __round__(self) -> Self:
        return Self(round(self.re, 0), round(self.im, 0))

    fn __round__(self, ndigits: Int) -> Self:
        return Self(round(self.re, ndigits), round(self.im, ndigits))
```

- Benchmarking:
  - The [bencher](#) module as part of the `benchmark` package is now public and documented. This module provides types such as `Bencher` which provides the ability to execute a `Benchmark` and allows for benchmarking configuration via the `BenchmarkConfig` struct.
- [String](#) and friends:
  - **Breaking.** Implicit conversion to `String` is now removed for builtin classes/types. Use [str\(\)](#) explicitly to convert to `String`.
  - Added [String.isspace\(\)](#) method conformant with Python's universal separators. This replaces the `isspace()` free function from the `string` module. (If you need the old function, it is temporarily available as `_isspace()`. It now takes a `UInt8` but is otherwise unchanged.)
  - [String.split\(\)](#) now defaults to whitespace and has Pythonic behavior in that it removes all adjacent whitespace by default.

- [String.strip\(\)](#), [lstrip\(\)](#) and [rstrip\(\)](#) can now remove custom characters other than whitespace. In addition, there are now several useful aliases for whitespace, ASCII lower/uppercase, and so on. ([PR #2555](#))
- String now has a [splitlines\(\)](#) method, which allows splitting strings at line boundaries. This method supports [universal newlines](#) and provides an option to retain or remove the line break characters. ([PR #2810](#))
- InlinedString has been renamed to [InlineString](#) to be consistent with other types.
- [StringRef](#) now implements [strip\(\)](#), which can be used to remove leading and trailing whitespace. ([PR #2683](#))
- StringRef now implements [startswith\(\)](#) and [endswith\(\)](#). ([PR #2710](#))
- Added a new [StringSlice](#) type, to replace uses of the unsafe StringRef type in standard library code.  
StringSlice is a non-owning reference to encoded string data. Unlike StringRef, a StringSlice is safely tied to the lifetime of the data it points to.
  - Added new [as\\_string\\_slice\(\)](#) methods to String and StringLiteral.
  - Added StringSlice initializer from an UnsafePointer and a length in bytes.
- Added a new [as\\_bytes\\_slice\(\)](#) method to String and StringLiteral, which returns a Span of the bytes owned by the string.
- Continued transition to [UnsafePointer](#) and unsigned byte type for strings:
  - Renamed String.\_as\_ptr() to [String.unsafe\\_ptr\(\)](#), and changed return type to UnsafePointer (was DTypePointer).
  - Renamed StringLiteral.data() to [StringLiteral.unsafe\\_ptr\(\)](#), and changed return type to UnsafePointer (was DTypePointer).
  - InlineString.as\_ptr() has been renamed to [unsafe\\_ptr\(\)](#) and now returns an UnsafePointer[UInt8] (was DTypePointer[DType.int8]).
  - StringRef.data is now an UnsafePointer (was DTypePointer) and [StringRef.unsafe\\_ptr\(\)](#) now returns an UnsafePointer[UInt8] (was DTypePointer[DType.int8]).
- Other built-ins:
  - The [Slice.\\_\\_len\\_\\_\(\)](#) function has been removed and [Slice](#) no longer conforms to the Sized trait. This clarifies the ambiguity of the semantics: the length of a slice always depends on the length of the object being sliced. Users that need the existing functionality can use the [Slice.unsafe\\_indices\(\)](#) method. This makes it explicit that this implementation does not check if the slice bounds are concrete or within any given object's length.
  - Added a built-in [sort\(\)](#) function for lists of elements that conform to the [ComparableCollectionElement](#) trait. ([PR #2609](#))
  - [int\(\)](#) can now take a string and a specified base to parse an integer from a string: `int("ff", 16)` returns 255. Additionally, if a base of zero is specified, the string will be parsed as if it was an integer literal,

with the base determined by whether the string contains the prefix "0x", "0o", or "0b". ([PR #2273](#), fixes [#2274](#))

- Added the [bin\(\)](#) built-in function to convert integral types into their binary string representation. ([PR #2603](#))
- Added the [atof\(\)](#) built-in function, which can convert a `String` to a `float64`. ([PR #2649](#))
- You can now use the built-in [any\(\)](#) and [all\(\)](#) functions to check for truthy elements in a collection. Because `SIMD.__bool__()` is now constrained to `size=1`, You must explicitly use these to get the truthy value of a SIMD vector with more than one element. This avoids common bugs around implicit conversion of SIMD to Bool. ([PR #2600](#))

For example:

```
fn truthy_simd():  
    var vec = SIMD[DType.int32, 4](0, 1, 2, 3)  
    if any(vec):  
        print("any elements are truthy")  
    if all(vec):  
        print("all elements are truthy")
```

- [object](#) now implements all the bitwise operators. ([PR #2324](#))
- [Tuple](#) now supports `__contains__()`. ([PR #2709](#)) For example:

```
var x = Tuple(1, 2, True)  
if 1 in x:  
    print("x contains 1")
```

- [ListLiteral](#) and [Tuple](#) now only require that element types be `Movable`. Consequently, `ListLiteral` and `Tuple` are themselves no longer `Copyable`.
- Added new `ImmutableStaticLifetime` and `MutableStaticLifetime` helpers.
- [UnsafePointer](#) and others:
  - Added new [memcpy\(\)](#) overload for `UnsafePointer[Scalar[_]]` pointers.
  - Removed the `get_null()` method from `UnsafePointer` and other pointer types. Please use the default constructor instead: `UnsafePointer[T]()`.
  - Many functions returning a pointer type have been unified to have a public API function of `unsafe_ptr()`.
  - The `Tensor.data()` method has been renamed to `unsafe_ptr()`. The return type is still a `DTypePointer[T]`.
- Collections:
  - [List](#) now has an [index\(\)](#) method that allows you to find the (first) location of an element in a `List` of `EqualityComparable` types. For example:

```
var my_list = List[Int](2, 3, 5, 7, 3)
print(my_list.index(3)) # prints 1
```

- List can now be converted to a String with a simplified syntax:

```
var my_list = List[Int](2, 3)
print(my_list.__str__()) # prints [2, 3]
```

Note that List doesn't conform to the Stringable trait yet so you cannot use `str(my_list)` yet. ([PR #2673](#))

- List has a simplified syntax to call the [count\(\)](#) method: `my_list.count(x)`. ([PR #2675](#))
- List() now supports `__contains__()`, so you can now use lists with the `in` operator:

```
if x in my_list:
```

([PR #2667](#))

- List now has an [unsafe\\_get\(\)](#) to get the reference to an element without bounds check or wraparound for negative indices. Note that this method is unsafe. Use with caution. ([PR #2800](#))
- Added a [fromkeys\(\)](#) method to Dict to return a Dict with the specified keys and values. ([PR 2622](#))
- Added a [clear\(\)](#) method to Dict. ([PR 2627](#))
- Dict now supports [reversed\(\)](#) for its `items()` and `values()` iterators. ([PR #2340](#))
- Dict now has a simplified conversion to String with `my_dict.__str__()`. Note that Dict does not conform to the Stringable trait so `str(my_dict)` is not possible yet. ([PR #2674](#))
- Dict now implements [get\(key\)](#) and `get(key, default)` functions. ([PR #2519](#))
- Added a temporary `__get_ref(key)` method to Dict, allowing you to get a Reference to a dictionary value.
- Added a new [InlineList](#) type, a stack-allocated list with a static maximum size. ([PR 2587#](#)) ([PR #2703](#))
- Added a new [Span](#) type for taking slices of contiguous collections. ([PR #2595](#))
- [os](#) module:
  - The `os` module now provides functionality for adding and removing directories using [mkdir\(\)](#) and [rmdir\(\)](#). ([PR #2430](#))
  - Added the [os.path.getsize\(\)](#) function, which gives the size in bytes of the file identified by the path. ([PR 2626](#))
  - Added [os.path.join\(\)](#) function. ([PR 2792](#))
  - Added a new [tempfile](#) module, with `gettempdir()` and `mkdtemp()` functions. ([PR 2742](#))
- [SIMD](#) type:

- Added [SIMD.shuffle\(\)](#) with StaticIntTuple mask. ([PR #2315](#))
- [SIMD.\\_\\_bool\\_\\_\(\)](#) is constrained such that it only works when `size` is 1. For SIMD vectors with more than one element, use [any\(\)](#) or [all\(\)](#). ([PR #2502](#))
- The [SIMD.reduce\\_or\(\)](#) and [SIMD.reduce\\_and\(\)](#) methods are now bitwise operations, and support integer types. ([PR #2671](#))
- Added [SIMD.\\_\\_repr\\_\\_\(\)](#) to get the verbose string representation of SIMD types. ([PR #2728](#))
- [math](#) package:
  - The `math.bit` module has been moved to a new top-level [bit](#) module. The following functions in this module have been renamed:
    - `ctlz` -> `countl_zero`
    - `cttz` -> `countr_zero`
    - `bit_length` -> `bit_width`
    - `ctpop` -> `pop_count`
    - `bswap` -> `byte_swap`
    - `bitreverse` -> `bit_reverse`
  - The `math.rotate_bits_left()` and `math.rotate_bits_right()` functions have been moved to the `bit` module.
  - The `is_power_of_2()` function in the `math` module is now called `is_power_of_two()` and located in the `bit` module.
  - The `abs()`, `round()`, `min()`, `max()`, `pow()`, and `divmod()` functions have moved from `math` to `builtin`, so you no longer need to import these functions.
  - The `math.tgamma()` function has been renamed to [math.gamma\(\)](#) to conform with Python's naming.
  - The implementation of the following functions have been moved from the `math` module to the new [utils.numerics](#) module: `isfinite()`, `isinf()`, `isnan()`, `nan()`, `nextafter()`, and `ulp()`. The functions continue to be exposed in the `math` module.
  - [math.gcd\(\)](#) now works on negative inputs, and like Python's implementation, accepts a variadic list of integers. New overloads for a `List` or `Span` of integers are also added. ([PR #2777](#))
- Async and coroutines:
  - [Coroutine](#) now requires a lifetime parameter. This parameter is set automatically by the parser when calling an async function. It contains the lifetimes of all the arguments and any lifetime accesses by the arguments. This ensures that argument captures by async functions keep the arguments alive as long as the coroutine is alive.
  - Async function calls are no longer allowed to borrow non-trivial register-passable types. Because async functions capture their arguments but register-passable types don't have lifetimes (yet), Mojo is not able to correctly track the reference, making this unsafe. To cover this safety gap, Mojo has temporarily disallowed binding non-trivial register-passable types to borrowed arguments in async functions.



- Miscellaneous:
  - Added an [InlineArray](#) type that works on memory-only types. Compare with the existing [StaticTuple](#) type, which is conceptually an array type, but only works on `AnyTrivialRegType`. ([PR #2294](#))
  - The [base64](#) package now includes encoding and decoding support for both the Base64 and Base16 encoding schemes. ([PR #2364](#)) ([PR #2584](#))
  - The `take()` function in [Variant](#) and [Optional](#) has been renamed to `unsafe_take()`.
  - The `get()` function in `Variant` has been replaced by `__getitem__()`. That is, `v.get[T]()` should be replaced with `v[T]`.
  - Various functions in the `algorithm` module are now built-in functions. This includes `sort()`, `swap()`, and `partition()`. `swap()` and `partition()` will likely shuffle around as we're reworking our built-in `sort()` function and optimizing it.
- `infinity` and `NaN` are now correctly handled in [testing.assert\\_almost\\_equal\(\)](#) and an `inf` function has been added to `utils/numerics.mojo`. ([PR #2375](#))

## Tooling changes

- Invoking `mojo package my-package -o my-dir` on the command line, where `my-package` is a Mojo package source directory, and `my-dir` is an existing directory, now outputs a Mojo package to `my-dir/my-package.mojopkg`. Previously, this had to be spelled out, as in `-o my-dir/my-package.mojo pkg`.
- The Mojo Language Server now reports a warning when a local variable is unused.
- Several `mojo` subcommands now support a `--diagnostic-format` option that changes the format with which errors, warnings, and other diagnostics are printed. By specifying `--diagnostic-format json` on the command line, errors and other diagnostics will be output in a structured [JSON Lines](#) format that is easier for machines to parse.

The full list of subcommands that support `--diagnostic-format` is as follows: `mojo build`, `mojo doc`, `mojo run`, `mojo package`, and `mojo test`. Further, the `mojo test --json` option has been subsumed into this new option; for the same behavior, run `mojo test --diagnostic-format json`.

Note that the format of the JSON output may change; we don't currently guarantee its stability across releases of Mojo.

- A new `--validate-doc-strings` option has been added to `mojo` to emit errors on invalid doc strings instead of warnings.
- The `--warn-missing-doc-strings` flag for `mojo` has been renamed to `--diagnose-missing-doc-strings`.
- A new decorator, `@doc_private`, was added that can be used to hide a declaration from being generated in the output of `mojo doc`. It also removes the requirement that the declaration has documentation (for example, when used with `--diagnose-missing-doc-strings`).
- Debugger users can now set breakpoints on function calls in OO builds even if the call has been inlined by the compiler.

- The Mojo Language Server now supports renaming local variables.

## Other changes

### ✗ Removed

- The `@unroll` decorator has been deprecated and removed. The decorator was supposed to guarantee that a decorated loop would be unrolled, or else the compiler would error. In practice, this guarantee was eroded over time, as a compiler-based approach cannot be as robust as the Mojo parameter system. In addition, the `@unroll` decorator did not make the loop induction variables parameter values, limiting its usefulness. Please see `@parameter` for a replacement!
- The method `object.print()` has been removed. Since `object` now conforms to the `Stringable` trait, you can use `print(my_object)` instead.
- The following functions have been removed from the `math` module:
  - `clamp()`; use the new `SIMD.clamp()` method instead.
  - `round_half_down()` and `round_half_up()`; these can be trivially implemented using the `ceil()` and `floor()` functions.
  - `add()`, `sub()`, `mul()`, `div()`, `mod()`, `greater()`, `greater_equal()`, `less()`, `less_equal()`, `equal()`, `not_equal()`, `logical_and()`, `logical_xor()`, and `logical_not()`; Instead, users should rely directly on the corresponding operators (`+`, `-`, `*`, `/`, `%`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&`, `^`, and `~`).
  - `identity()` and `reciprocal()`; users can implement these trivially.
  - `select()`; removed in favor of using `SIMD.select()` directly.
  - `is_even()` and `is_odd()`; these can be trivially implemented using bitwise `&` with `1`.
  - `roundeven()`; the new `SIMD.roundeven()` method now provides the identical functionality.
  - `div_ceil()`; use the new `ceildiv()` function.
  - `rotate_left()` and `rotate_right()`; the same functionality is available in the builtin `SIMD.rotate_{left,right}()` methods for `SIMD` types, and the `bit.rotate_bits_{left,right}()` methods for `Int`.
  - An overload of `math.pow()` taking an integer parameter exponent.
  - `align_down_residual()`; it can be trivially implemented using `align_down()`.
  - `all_true()`, `any_true()`, and `none_true()`; use `SIMD.reduce_and()` and `SIMD.reduce_or()` directly.
  - `reduce_bit_count()`; use the new `SIMD.reduce_bit_count()` directly.
  - `rint()` and `nearbyint()`; use `round()` or `SIMD.roundeven()` as appropriate.
- The `EvaluationMethod` has been removed from `math.polynomial` and Estrin's method is no longer available. This method was limited to degree 10 or less, underutilized, and its performance unclear. In the future, this might be reintroduced with an improved implementation if needed, when better performance benchmarking infrastructure is available. The default behavior of `math.polynomial.polynomial_evaluate()` is unchanged (Horner's method).

- The `math.bit.select()` and `math.bit.bit_and()` functions have been removed. The same functionality is available in the builtin `SIMD.select` and `SIMD.__and__()` methods, respectively.
- The `math.limit` module has been removed. The same functionality is available as follows:
  - `math.limit.inf()`: use `utils.numerics.max_or_inf()`
  - `math.limit.neginf()`: use `utils.numerics.min_or_neg_inf()`
  - `math.limit.max_finite()`: use `utils.numerics.max_finite()`
  - `math.limit.min_finite()`: use `utils.numerics.min_finite()`
- The `tensor.random` module has been removed. The same functionality is now accessible via the [Tensor.rand\(\)](#) and [Tensor.randn\(\)](#) static methods.
- The builtin `SIMD` struct no longer conforms to `Indexer`; users must explicitly cast `Scalar` values using `int`.

## Fixed

- [#1837](#) Fix self-referential variant crashing the compiler.
- [#2363](#) Fix LSP crashing on simple trait definitions.
- [#1787](#) Fix error when using `//` on `FloatLiteral` in alias expression.
- Made several improvements to dictionary performance. Dicts with integer keys are most heavily affected, but large dicts and dicts with large values will also see large improvements.
- [#2692](#) Fix `assert_raises` to include calling location.

## Special thanks

Special thanks to our community contributors:

[@rd4com](#), [@toiletsandpaper](#), [@helehex](#), [@artemiogr97](#), [@mikowals](#), [@kernhanda](#), [@lsh](#), [@LJ-9801](#), [@YichengDWu](#), [@gabrieldemarmiesse](#), [@fknfilewalker](#), [@jayzhan211](#), [@martinvuyk](#), [@ChristopherLR](#), [@mzaks](#), [@bgreni](#), [@Brian-M-J](#), [@leandrolcampos](#)

## v24.3 (2024-05-02)

## Highlights

- `AnyPointer` was renamed to [UnsafePointer](#) and is now Mojo's preferred unsafe pointer type. It has several enhancements, including:
  - The element type can now be any type: it doesn't require `Movable`.
  - Because of this, the `take_value()`, `emplace_value()`, and `move_into()` methods have been changed to top-level functions and renamed. The new functions are:

- [initialize\\_pointee\\_copy](#)
- [initialize\\_pointee\\_move](#)
- [move\\_from\\_pointee\(\)](#)
- [move\\_pointee](#)
- A new [destroy\\_pointee\(\)](#) function runs the destructor on the pointee.
- `UnsafePointer` can be initialized directly from a [Reference](#) with `UnsafePointer(someRef)` and can convert to a reference with `yourPointer[ ]`. Both infer element type and address space. Note that when you convert a pointer to a reference, there's no way for Mojo to track the lifetime of the original value. So the resulting reference is no safer than the original pointer.
- All of the pointer types received some cleanup to make them more consistent, for example the `unsafe.bitcast()` global function is now a consistent [bitcast\(\)](#) method on the pointers, which can convert element type and address space.
- Improvements to variadic arguments support.
  - Heterogeneous variadic pack arguments now work reliably even with memory types, and have a more convenient API to use, as defined by the [VariadicPack](#) type. For example, a simplified version of `print` can be implemented like this:

```
fn print[T: Stringable, *Ts: Stringable](first: T, *rest: *Ts):
    print_string(str(first))

    @parameter
    fn print_elt[T: Stringable](a: T):
        print_string(" ")
        print_string(a)
    rest.each[print_elt]()
```

- Mojo now supports declaring functions that have both optional and variadic arguments, both positional and keyword-only. For example, this now works:

```
fn variadic_arg_after_default(
    a: Int, b: Int = 3, *args: Int, c: Int, d: Int = 1, **kwargs: Int
): ...
```

Positional variadic parameters also work in the presence of optional parameters. That is:

```
fn variadic_param_after_default[e: Int, f: Int = 2, *params: Int]():
    pass
```

Note that variadic keyword parameters are not supported yet.

For more information, see [Variadic arguments](#) in the Mojo Manual.

- The `mojo build` and `mojo run` commands now support a `-g` option. This shorter alias is equivalent to writing `--debug-level full`. This option is also available in the `mojo debug` command, but is already the default.
- Many new standard library APIs have been filled in, including many community contributions. Changes are listed in the standard library section.
- The Mojo Manual has a new page on [Types](#).

## Language changes

- Certain dunder methods that take indices (`__getitem__()`, `__setitem__()`, and `__refitem__()`) or names (`__getattr__()` and `__setattr__()`) can now take the index or name as a parameter value instead of an argument value. This is enabled when you define one of these methods with no argument other than `self` (for a getter) or `self` and the set value (for a setter).

This enables types that can only be subscripted into with parameters, as well as things like the following example, which passes the attribute name as a parameter so that attribute names can be checked at compile time.

```
struct RGB:
    fn __getattr__[name: StringLiteral](self) -> Int:
        @parameter
        if name == "r":    return ...
        elif name == "g": return ...
        else:
            constrained[name == "b", "can only access with r, g, or b members"]()
            return ...

var rgb = RGB()
print(rgb.b) # Works
print(rgb.q) # Compile error
```

- Mojo now allows users to capture the source location of code and call location of functions dynamically using the `__source_location()` and `__call_location()` functions. For example:

```
from builtin._location import __call_location

@always_inline
fn my_assert(cond: Bool, msg: String):
    if not cond:
        var call_loc = __call_location()
        print("In", call_loc.file_name, "on line", str(call_loc.line) + ":", msg)

fn main():
    my_assert(False, "always fails") # some_file.mojo, line 193
```

This prints "In /path/to/some\_file.mojo on line 193: always fails". Note that `__call_location()` only works in `@always_inline` or `@always_inline("nodebug")` functions. It gives incorrect results if placed in an `@always_inline` function that's called *from* an `@always_inline("nodebug")` function.

This feature is still evolving and for the time being you need to explicitly import these APIs, as shown above. In the future, these will probably be built-in functions and not require an import statement.

Neither `__source_location()` nor `__call_location()` work when called in a parameter context. For example:

```
from builtin._location import __call_location

@always_inline
fn mystery_location() -> String:
  var loc = __call_location()
  return str(loc.file_name)

def main():
  alias doesnt_work = mystery_location() # <unknown location in parameter context>
```

## Standard library changes

### ★ New

- [List](#) has several new methods:
  - `pop(index)` for removing an element at a particular index. By default, `List.pop()` removes the last element in the list. ([@LJ-9801](#), fixes [#2017](#))
  - `resize(new_size)` for resizing the list without the need to specify an additional value. ([@mikowals](#), fixes [#2133](#))
  - `insert(index, value)` for inserting a value at a specified index into the `List`. ([@whym1here](#), fixes [#2134](#))
  - A new constructor `List(ptr, size, capacity)` to to avoid needing to do a deep copy of an existing contiguous memory allocation when constructing a new `List`. ([@StandinKP](#), fixes [#2170](#))
- [Dict](#) now has a `update()` method to update keys/values from another `Dict`. ([@gabriel-demarmiesse](#))
- [Set](#) now has named methods for set operations:
  - `difference()` mapping to `-`
  - `difference_update()` mapping to `--`
  - `intersection_update()` mapping to `&=`
  - `update()` mapping to `|=`([@arvindavoudi](#))
- `Dict`, `List`, and `Set` all conform to the `Boolable` trait. The collections evaluate to `True` if they contain any elements, `False` otherwise:

```
def list_names(names: List[String]):
  if names:
    for name in names:
      print(name[])
  else:
    print("No names to list.")
```

([@gabrieldemarmiesse](#))

- Added [reversed\(\)](#) function for creating reversed iterators. Several range types, List, and Dict now support iterating in reverse.

```
var numbers = List(1, 2, 3, 4, 5)
for number in reversed(numbers):
  print(number)
```

([@helehex](#) and [@jayzhan211](#), contributes towards [#2325](#))

- [Optional](#) now implements `__is__` and `__isnot__` methods so that you can compare an `Optional` with `None`. For example:

```
var opt = Optional(1)
if opt is not None:
  print(opt.value()[])
```

([@gabrieldemarmiesse](#))

- [Tuple](#) now works with memory-only element types like `String` and allows you to directly index into it with a parameter expression. This means you can now simply use `x = tup[1]` like Python instead of `x = tup.get[1, Int]()`. You can also assign into tuple elements now as well with `tup[1] = x`.

```
var tuple = ("Green", 9.3)
var name = tuple[0]
var value = tuple[1]
```

Note that because the subscript must be a parameter expression, you can't iterate through a `Tuple` using an ordinary `for` loop.

- The [Reference](#) type has several changes, including:
  - It has moved to the `memory.reference` module instead of `memory.unsafe`.
  - `Reference` now has an [unsafe\\_bitcast\(\)](#) method, similar to the pointer types.
  - Several unsafe methods were removed, including `offset()`, `destroy_element_unsafe()` and `emplace_ref_unsafe()`. This is because `Reference` is a safe type—use `UnsafePointer` to do unsafe operations.

- [Bool](#) can now be implicitly converted from any type conforming to the [Boolable](#) trait. This means that you no longer need to write code like this:

```
@value
struct MyBoolable:
  fn __bool__(self) -> Bool: ...

fn takes_boolable[T: Boolable](cond: T): ...

takes_boolable(MyBoolable())
```

Instead, you can simply write:

```
fn takes_bool(cond: Bool): ...

takes_bool(MyBoolable())
```

Note that calls to `takes_bool()` will perform the implicit conversion, so in some cases it is still better to explicitly declare a type parameter, e.g.:

```
fn takes_two_boolables[T: Boolable](a: T, b: T):
  # Short circuit means `b.__bool__()` might not be evaluated.
  if a.__bool__() and b.__bool__():
    ...
```

- [PythonObject](#) now conforms to the [KeyElement](#) trait, meaning that it can be used as key type for [Dict](#). This allows you to easily build and interact with Python dictionaries in Mojo:

```
def main():
  d = PythonObject(Dict[PythonObject, PythonObject]())
  d["foo"] = 12
  d[7] = "bar"
  d["foo"] = [1, 2, "something else"]
  print(d) # prints `{'foo': [1, 2, 'something else'], 7: 'bar'}`
```

- [FileHandle.seek\(\)](#) now has a `whence` argument that defaults to `os.SEEK_SET` to seek from the beginning of the file. You can now set to `os.SEEK_CUR` to offset by the current `FileHandle` seek position:

```
var f = open("/tmp/example.txt")
# Skip 32 bytes
f.seek(os.SEEK_CUR, 32)
```

Or `os.SEEK_END` to offset from the end of file:



```
# Start from 32 bytes before the end of the file
f.seek(os.SEEK_END, -32)
```

- [FileHandle.read\(\)](#) can now read straight into a [DTypePointer](#):

```
var file = open("/tmp/example.txt", "r")

# Allocate and load 8 elements
var ptr = DTypePointer[DType.float32].alloc(8)
var bytes = file.read(ptr, 8)
print("bytes read", bytes)
print(ptr.load[width=8]())
```

- The `sys` module now contains an `exit()` function that would exit a Mojo program with the specified error code.

```
from sys import exit

exit(0)
```

- The constructors for [Tensor](#) have been changed to be more consistent. As a result, constructors take the shape as the first argument (instead of the second) when constructing a tensor with pointer data.

If you pass a single scalar value to the `Tensor` constructor, it now broadcasts the value to all elements in the tensor. For example, `Tensor[DType.float32](TensorShape(2,2), 0)` constructs a 2x2 tensor initialized with all zeros. This provides an easy way to fill in the data of a tensor.

- [String](#) now has `removeprefix()` and `removesuffix()` methods. ([@gabriel demarmiesse](#))
- The [ord](#) and [chr](#) functions have been improved to accept any Unicode character. ([@mzaks](#), contributes towards [#1616](#))
- [atoi\(\)](#) now handles whitespace. The `atoi()` function is used internally by `String.__int__()`, so `int(String(" 10 "))` now returns 10 instead of raising an error. ([@artemiogr97](#))
- [SIMD](#) now implements the `__rmod__()` method. ([@bgreni](#), fixes [#1482](#))
- [bool\(None\)](#) is now implemented. ([@zhoujingya](#))
- The [DTypePointer](#) type now implements `gather()` for gathering a SIMD vector from offsets of a current pointer. Similarly, support for `scatter()` was added to scatter a SIMD vector into offsets of the current pointer. ([@leandrolcampos](#))
- The [len\(\)](#) function now handles a [range\(\)](#) specified with a negative end value, so that things like `len(range(-1))` work correctly. ([@soraros](#))
- [debug\\_assert\(\)](#) now prints its location (filename, line, and column where it was called) in its error message. Similarly, the `assert` helpers in the [testing](#) module now include location information in their messages.

- The [testing.assert\\_equal SIMD\(\)](#) function now raises if any of the elements mismatch in the two SIMD arguments being compared. ([@gabrieldemarmiesse](#))
- The [testing.assert\\_almost\\_equal\(\)](#) and [math.isclose\(\)](#) functions now have an `equal_nan` flag. When set to `True`, then NaNs are considered equal.
- The [object](#) type now supports the division, modulo, and left and right shift operators, including the in-place and reverse variants. ([@LJ-9801](#), fixes [#2224](#))
- Added checked arithmetic operations for SIMD integers.

SIMD integer types (including the sized integer scalars like `Int64`) can now perform checked additions, subtractions, and multiplications using the following new methods:

- `add_with_overflow()`
- `sub_with_overflow()`
- `mul_with_overflow()`

Checked arithmetic allows the caller to determine if an operation exceeded the numeric limits of the type. For example:

```
var simd = SIMD[DType.int8, 4](7, 11, 13, 17)
var product: SIMD[DType.int8, 4]
var overflow: SIMD[DType.bool, 4]
(product, overflow) = simd.mul_with_overflow(simd)
for i in range(len(product)):
    if overflow[i]:
        print("<overflow>")
    else:
        print(product[i])
```

([@lsh](#))

- Added [os.remove\(\)](#) and [os.unlink\(\)](#) for deleting files. ([@artemiogr97](#), fixes [#2306](#))

## Changed

- The [parallel\\_memcpy\(\)](#) function has moved from the `buffer` package to the `algorithm` package. Please update your imports accordingly.
- [Optional.value\(\)](#) now returns a reference instead of a copy of the contained value.

To perform a copy manually, dereference the result:

```
var result = Optional(123)

var value = result.value()[!]
```

([@lsh](#), fixes [#2179](#))

- Per the accepted community proposal, [Standardize the representation of byte sequence as a sequence of unsigned 8-bit integers](#), began transition to using `UInt8` by changing the data pointer of `Error` to `DTypePointer[DType.uint8]`. ([@gabrieldemarmiesse](#), contributes towards [#2317](#))
- Continued transition to `UnsafePointer` from the legacy `Pointer` type in various standard library APIs and internals. ([@gabrieldemarmiesse](#))

## Tooling changes

- The behavior of `mojo build` when invoked without an output `-o` argument has changed slightly: `mojo build ./test-dir/program.mojo` now outputs an executable to the path `./program`, whereas before it would output to the path `./test-dir/program`.
- The `mojo package` command no longer supports the `-D` flag. All compilation environment flags should be provided at the point of package use (e.g. `mojo run` or `mojo build`).
- The REPL no longer allows type level variable declarations to be uninitialized, e.g. it will reject `var s: String`. This is because it does not do proper lifetime tracking (yet!) across cells, and so such code would lead to a crash. You can work around this by initializing to a dummy value and overwriting later. This limitation only applies to top level variables, variables in functions work as they always have.


## Other changes

### Low-level language changes

- A low-level `__get_mvalue_as_litref(x)` builtin was added to give access to the underlying memory representation as a `!lit.ref` value without checking initialization status of the underlying value. This is useful in very low-level logic but isn't designed for general usability and will likely change in the future.
- Properties can now be specified on inline MLIR ops:

```
_ = __mlir_op.`kgen.source_loc`[
  _type = (
    __mlir_type.index, __mlir_type.index, __mlir_type.`!kgen.string`
  ),
  _properties = __mlir_attr.`{inlineCount = 1 : i64}`,
]()
```

As the example shows above, the protected `_properties` attribute can be passed during op construction, with an MLIR `DictionaryAttr` value.

 Removed

- Support for "register only" variadic packs has been removed. Instead of `AnyRegType`, please upgrade your code to `AnyType` in examples like this:

```
fn your_function[*Types: AnyRegType](*args: *Ts): ...
```

This move gives you access to a nicer API and has the benefit of being memory safe and correct for non-trivial types. If you need specific APIs on the types, please use the correct trait instead of `AnyType`.

- `List.pop_back()` has been removed. Use `List.pop()` instead which defaults to popping the last element in the list.
- `SIMD.to_int(value)` has been removed. Use `int(value)` instead.
- The `__get_lvalue_as_address(x)` magic function has been removed. To get a reference to a value use `Reference(x)` and if you need an unsafe pointer, you can use `UnsafePointer.address_of(x)`.

## Fixed

- [#516](#) and [#1817](#) and many others, e.g. "Can't create a function that returns two strings."
- [#1178](#) (os/kern) failure (5).
- [#1609](#) alias with `DynamicVector[Tuple[Int]]` fails.
- [#1987](#) Defining `main` in a Mojo package is an error, for now. This is not intended to work yet, erroring for now will help to prevent accidental undefined behavior.
- [#1215](#) and [#1949](#) The Mojo LSP server no longer cuts off hover previews for functions with functional arguments, parameters, or results.
- [#1901](#) Fixed Mojo LSP and documentation generation handling of inout arguments.
- [#1913](#) - `0__` no longer crashes the Mojo parser.
- [#1924](#) JIT debugging on Mac has been fixed.
- [#1941](#) Mojo variadic arguments don't work with non-trivial register-only types.
- [#1963](#) `a!=0` is now parsed and formatted correctly by `mojo format`.
- [#1676](#) Fix a crash related to `@value` decorator and structs with empty body.
- [#1917](#) Fix a crash after syntax error during tuple creation.
- [#2006](#) The Mojo LSP now properly supports signature types with named arguments and parameters.
- [#2007](#) and [#1997](#) The Mojo LSP no longer crashes on certain types of closures.
- [#1675](#) Ensure `@value` decorator fails gracefully after duplicate field error.
- [#2068](#) Fix `SIMD.reduce()` for `size_out == 2`. ([@soraros](#))

# v24.2.1 (2024-04-11)

## v24.2 (2024-03-28)

### Legendary

- The Mojo standard library is now open source! Check out the [README](#) for everything you need to get started.
- Structs and other nominal types are now allowed to implicitly conform to traits. A struct implicitly conforms to a trait if it implements all the requirements for the trait. For example, any struct that implements the `__str__()` method implicitly conforms to `Stringable`, and is usable with the `str()` built-in function.

```
@value
struct Foo:
    fn __str__(self) -> String:
        return "foo!"

fn main():
    print(str(Foo())) # prints 'foo!'
```

We still strongly encourage you to explicitly list the traits a struct conforms to when possible:

```
@value
struct Foo(Stringable): ...
```

Not only is this useful for documentation and for communicating intentions, but in the future, explicit conformance will be useful for features like default methods and extensions.

- Mojo's Python interoperability now supports passing keyword arguments to Python functions:

```
from python import Python

def main():
    plt = Python.import_module("matplotlib.pyplot")
    plt.plot((5, 10), (10, 15), color="red")
    plt.show()
```

## Language changes

### New

- Mojo now has support for variadic keyword arguments, often referred to as `**kwargs`. This means you can now declare and call functions like this:

```
fn print_nicely(**kwargs: Int) raises:
  for key in kwargs.keys():
    print(key[], "=", kwargs[key[]])

# prints:
# `a = 7`
# `y = 8`
print_nicely(a=7, y=8)
```

For more details (and a list of current limitations), see [Variadic keyword arguments](#) in the Mojo manual.

## Changed or removed

- `let` declarations now produce a compile time error instead of a warning, our next step in [removing let declarations](#). The compiler still recognizes the `let` keyword for now in order to produce a good error message, but that will be removed in subsequent releases.
- Mojo now warns about unused values in both `def` and `fn` declarations, instead of completely disabling the warning in `def` s. It never warns about unused `object` or `PythonObject` values, tying the warning to these types instead of the kind of function they are unused in. This will help catch API usage bugs in `def` s and make imported Python APIs more ergonomic in `fn` s.
- For the time being, dynamic type values will be disabled in the language. For example, the following will now fail with an error:

```
var t = Int # dynamic type values not allowed

struct SomeType: ...

takes_type(SomeType) # dynamic type values not allowed
```

We want to take a step back and (re)design type valued variables, existentials, and other dynamic features. This does not affect type valued **parameters**, so the following works as before:

```
alias t = Int # still 🔥

struct SomeType: ...

takes_type[SomeType]() # already 🔥

>fn uses_trait[T: SomeTrait](value: T): ... # still 🔥
```

- The `*_` expression in parameter expressions is now required to occur at the end of a positional parameter list, instead of being allowed in the middle.

```
# No longer supported
alias FirstUnbound = SomeStruct[*_, 42]
alias MidUnbound   = SomeStruct[7, *_ , 6]
# Still supported
alias LastUnbound  = SomeStruct[42, *_]
```

We narrowed this because we want to encourage type designers to get the order of parameters right, and want to extend `*_` to support keyword parameters as well in the future.

## Standard library changes

### ★ New

- `DynamicVector` has been renamed to [List](#), and has moved from the `collections.vector` module to the `collections.list` module. In addition:
  - You can now construct a `List` from a variadic number of values. For example:

```
var numbers = List[Int](1, 2, 3)
```

- `List` and [InlinedFixedVector](#) types now support negative indexing. This means that you can write `vec[-1]` which is equivalent to `vec[len(vec)-1]`.
  - `List.push_back()` has been removed. Please use the `append()` function instead.
- The [print\(\)](#) function now takes `sep` and `end` keyword arguments. This means that you can write:

```
print("Hello", "Mojo", sep=", ", end="!!!\n") # prints Hello, Mojo!!!
```

`sep` defaults to the empty string and `end` defaults to `"\n"`.

Also, the `print_no_newline()` function has been removed. Please use `print(end="")` instead.

- The [FloatLiteral](#) type is now an infinite-precision nonmaterializable type. This means you can do compile-time calculations using `FloatLiteral` without rounding errors. When materialized at runtime, a `FloatLiteral` value is converted to a [Float64](#).

```
# third is an infinite-precision FloatLiteral value
alias third = 1.0 / 3.0
# t is a Float64
var t = third
```

- String types all conform to the [IntableRaising](#) trait. This means that you can now call `int("123")` to get the integer `123`. If the integer cannot be parsed from the string, then an error is raised.
- The `Tensor` type now has [argmax\(\)](#) and [argmin\(\)](#) functions to compute the position of the max or min value. Note: this should return a `Tensor[Int]` but currently the output tensor is the same type as the input tensor. This will be fixed in a future release.
- Added a new [collections.OptionalReg](#) type, a register-passable alternative to [Optional](#).
- The [ulp\(\)](#) function has been added to the `math` module. This allows you to get the units of least precision (or units of last place) of a floating point value.

## Changed

- The `simd_load()`, `simd_store()`, `aligned_simd_load()`, and `aligned_simd_store()` methods on [DTypePointer](#), [Buffer](#), and [NDBuffer](#) have been merged into a more expressive set of `load()` and `store()` methods with keyword-only `width` and `alignment` parameters:

```
# Doesn't work
my_simd = my_buffer.simd_load[simd_width](index)
# Works
my_simd = my_buffer.load[width=simd_width](index)
# Doesn't work
my_buffer.aligned_simd_store[width, alignment](my_simd)
# Works
my_buffer.store[width=width, alignment=alignment](my_simd)
```

- The [EqualityComparable](#) trait now requires the `__ne__()` method for conformance in addition to the previously required `__eq__()` method.
- Many types now declare conformance to `EqualityComparable` trait.
- [StaticTuple](#) parameter order has changed to `StaticTuple[type, size]` for consistency with SIMD and similar collection types.
- The signature of the [elementwise\(\)](#) function has been changed. The new order is `function`, `simd_width`, and then `rank`. As a result, the `rank` parameter can now be inferred and one can call `elementwise()` without it:

```
elementwise[func, simd_width](shape)
```

- `PythonObject` is now register-passable.
- `PythonObject.__iter__()` now works correctly on more types of iterable Python objects. Attempting to iterate over non-iterable objects will now raise an exception instead of behaving as if iterating over an empty sequence. `__iter__()` also now borrows `self` rather than requiring `inout`, allowing code like:



```
for value in my_dict.values():  
    ...
```



## Moved

- We took the opportunity to rehome some modules into their correct package as we were going through the process of open-sourcing the Mojo standard library. Specifically, the following are some breaking changes worth calling out. Please update your import statements accordingly.

- [Buffer](#), [NDBuffer](#), and friends have moved from the `memory` package into a new `buffer` package.

```
from buffer import Buffer, NDBuffer
```

- `utils.list`, including the [Dim](#) and [DimList](#) types, has moved to the `buffer` package.

```
from buffer import Dim, DimList
```

- The [parallel\\_memcpy\(\)](#) function has moved from the `memory` package into the `buffer` package.

```
from buffer import parallel_memcpy
```

- The [rand\(\)](#) and [randn\(\)](#) functions from the `random` package that return a `Tensor` have moved to the `tensor` package. Note that the overloads that write to a `DTypePointer` remain in the `random` package.

If you happen to be using both versions in the same source file, you can import them both using the `import as` syntax:

```
from tensor import rand  
from random import rand as rand_dt
```

- The `trap()` function has been renamed to [abort\(\)](#). It also has moved from the `debug` module to the `os` module.

```
from os import abort
```

- The [isinf\(\)](#) and [isfinite\(\)](#) methods have been moved from `math.limits` to the `math` module.

```
from math import ininf, isfinite
```

## Tooling changes

## ★ New

- Docstring code blocks can now use `##` to hide lines of code from documentation generation.

For example:

```
var value = 5
## print(value)
```

Will generate documentation of the form:

```
var value = 5
```

Hidden lines are processed as if they were normal code lines during test execution. This allows for writing additional code within a docstring example that is only used to ensure the example is runnable/testable.

- The Mojo LSP server now allow you to specify additional search paths to use when resolving imported modules in a document. You can specify search paths on the command line, using the `-I` option, or you can add them to the `mojo.lsp.includeDirs` setting in the VS Code extension.

## Other changes

### ✗ Removed

- The `__get_address_as_lvalue` magic function has been removed. You can now get an LValue from a `Pointer` or `Reference` by using the dereference operator (`[]`):

```
var ptr: Pointer[MyRecord]
...
# Doesn't work
__get_address_as_lvalue(ptr.value) = MyRecord(3, 5)
# Works
ptr[] = MyRecord(3, 5)
```

- The type parameter for the `memcpy` function is now automatically inferred. This means that calls to `memcpy` of the form `memcpy[DType.xyz](...)` will no longer work and the user would have to change the code to `memcpy(...)`.
- The [memcpy\(\)](#) overload that worked on [Buffer](#) types has been removed in favor of just overloads for [Pointer](#) and [DTypePointer](#):

```
# Doesn't work
memcpy(destBuffer, srcBuffer, count)
```

```
# Works  
memcpy(destBuffer.data, srcBuffer.data, count)
```

- The functions `max_or_inf()`, `min_or_neginf()` have been removed from `math.limit`. These functions were only used by the SIMD type.
- As mentioned previously, the `print_no_newline()` function has been removed. Please use `print(end="")` instead.

## Fixed

- [#1362](#) - Parameter inference now recursively matches function types.
- [#951](#) - Functions that were both `async` and `@always_inline` incorrectly errored.
- [#1858](#) - Trait with parametric methods regression.
- [#1892](#) - Forbid unsupported decorators on traits.
- [#1735](#) - Trait-typed values are incorrectly considered equal.
- [#1909](#) - Crash due to nested import in unreachable block.
- [#1921](#) - Parser crashes binding Reference to lvalue with subtype lifetime.
- [#1945](#) - `Optional[T].orElse()` should return `T` instead of `Optional[T]`.
- [#1940](#) - Constrain `math.copysign` to floating point or integral types.
- [#1838](#) - Variadic `print` does not work when specifying `end=""`
- [#1826](#) - The `SIMD.reduce` methods correctly handle edge cases where `size_out >= size`.

## v24.1.1 (2024-03-18)

This release includes installer improvements and enhanced error reporting for installation issues. Otherwise it is functionally identical to Mojo 24.1.

## v24.1 (2024-02-29)

### Legendary

- Mojo is now bundled with [the MAX platform!](#)

As such, the Mojo package version now matches the MAX version, which follows a `YY.MAJOR.MINOR` version scheme. Because this is our first release in 2024, that makes this version `24.1`.

- Mojo debugging support is here! The Mojo VS Code extension includes debugger support. For details, see [Debugging](#) in the Mojo Manual.

## ★ New

- We now have a [Set](#) type in our collections! `Set` is backed by a `Dict`, so it has fast add, remove, and `in` checks, and requires member elements to conform to the `KeyElement` trait.

```
from collections import Set

var set = Set[Int](1, 2, 3)
print(len(set)) # 3
set.add(4)

for element in set:
    print(element[])

set -= Set[Int](3, 4, 5)
print(set == Set[Int](1, 2)) # True
print(set | Set[Int](0, 1) == Set[Int](0, 1, 2)) # True
let element = set.pop()
print(len(set)) # 1
```

- Mojo now supports the `x in y` expression as syntax sugar for `y.__contains__(x)` as well as `x not in y`.
- Mojo now has support for keyword-only arguments and parameters. For example:

```
fn my_product(a: Int, b: Int = 1, *, c: Int, d: Int = 2):
    print(a * b * c * d)

my_product(3, c=5) # prints '30'
my_product(3, 5, d=7) # error: missing 1 required keyword-only argument: 'c'
```

This includes support for declaring signatures that use both variadic and keyword-only arguments/parameters. For example, the following is now possible:

```
fn prod_with_offset(*args: Int, offset: Int = 0) -> Int:
    var res = 1
    for i in range(len(args)):
        res *= args[i]
    return res + offset

print(prod_with_offset(2, 3, 4, 10)) # prints 240
print(prod_with_offset(2, 3, 4, offset=10)) # prints 34
```

Note that variadic keyword-only arguments/parameters (for example, `**kwargs`) are not supported yet. That is, the following is not allowed:

```
fn variadic_kw_only(a: Int, **kwargs): ...
```

For more information, see [Positional-only and keyword-only arguments](#) in the Mojo Manual.

- The `print()` function now accepts a keyword-only argument for the `end` which is useful for controlling whether a newline is printed or not after printing the elements. By default, `end` defaults to `"\n"` as before.
- The Mojo SDK can now be installed on AWS Graviton instances.
- A new version of the [Mojo Playground](#) is available. The new playground is a simple interactive editor for Mojo code, similar to the Rust Playground or Go Playground. The old JupyterLab based playground will remain online until March 20th.
- The Mojo LSP server will now generate fixits for populating empty documentation strings:

```
fn foo(arg: Int):  
    """ # Unexpected empty documentation string
```

Applying the fixit from above will generate:

```
fn foo(arg: Int):  
    """[summary].  
  
    Args:  
        arg: [description].  
    """
```

- Added new `*_` syntax that allows users to explicitly unbind any number of positional parameters. For example:

```
struct StructWithDefault[a: Int, b: Int, c: Int = 8, d: Int = 9]: pass  
  
alias all_unbound = StructWithDefault[*_]  
# equivalent to  
alias all_unbound = StructWithDefault[_ , _ , _ , _]  
  
alias first_bound = StructWithDefault[5, *_]   
# equivalent to  
alias first_bound = StructWithDefault[5, _ , _ , _]  
  
alias last_bound = StructWithDefault[*_, 6]  
# equivalent to  
alias last_bound = StructWithDefault[_ , _ , _ , 6]  
  
alias mid_unbound = StructWithDefault[3, *_, 4]  
# equivalent to  
alias mid_unbound = StructWithDefault[3, _ , _ , 4]
```

As demonstrated above, this syntax can be used to explicitly unbind an arbitrary number of parameters, at the beginning, at the end, or in the middle of the operand list. Since these unbound parameters must be explicitly specified at some point, default values for these parameters are not applied. For example:

```
alias last_bound = StructWithDefault[*_, 6]
# When using last_bound, you must specify a, b, and c. last_bound
# doesn't have a default value for `c`.
var s = last_bound[1, 2, 3]()
```

For more information see the Mojo Manual sections on [partially-bound types](#) and [automatic parameterization of functions](#).

- [DynamicVector](#) now supports iteration. Iteration values are instances of [Reference](#) and require dereferencing:

```
var v: DynamicVector[String]()
v.append("Alice")
v.append("Bob")
v.append("Charlie")
for x in v:
    x[] = str("Hello, ") + x[]
for x in v:
    print(x[])
```

- `DynamicVector` now has [reverse\(\)](#) and [extend\(\)](#) methods.
- The `mojo` package command now produces compilation agnostic packages. Compilation options such as `OO`, or `--debug-level`, are no longer needed or accepted. As a result, packages are now smaller, and extremely portable.
- Initializers for `@register_passable` values can (and should!) now be specified with `inout self` arguments just like memory-only types:

```
@register_passable
struct YourPair:
    var a: Int
    var b: Int
    fn __init__(inout self):
        self.a = 42
        self.b = 17
    fn __copyinit__(inout self, existing: Self):
        self.a = existing.a
        self.b = existing.b
```

This form makes the language more consistent, more similar to Python, and easier to implement advanced features for. There is also no performance impact of using this new form: the compiler arranges to automatically return the value in a register without requiring you to worry about it.

The older `-> Self` syntax is still supported in this release, but will be removed in a subsequent one, so please migrate your code. One thing to watch out for: a given struct should use one style or the other, mixing some of each won't work well.

- The `inout self` initializer form is **required** for initializers of `@register_passable` types that may raise errors:

```
@register_passable
struct RaisingCtor:
    fn __init__(inout self) raises:
        raise
```

- async functions that may raise errors have been temporarily disabled in this build. The implementation of Mojo async is undergoing a rework 🚧.
- The standard library `slice` type has been renamed to [Slice](#), and a `slice` function has been introduced. This makes Mojo closer to Python and makes the `Slice` type follow the naming conventions of other types like `Int`.
- "Slice" syntax in subscripts is no longer hard coded to the builtin `slice` type: it now works with any type accepted by a container's `__getitem__()` method. For example:

```
@value
struct UnusualSlice:
    var a: Int
    var b: Float64
    var c: String

struct YourContainer:
    fn __getitem__(self, slice: UnusualSlice) -> T: ...
```

Given this implementation, you can subscript into an instance of `YourContainer` like `yc[42:3.14:"🔥"]` and the three values are passed to the `UnusualSlice` constructor.

- The `__refitem__()` accessor method may now return a [Reference](#) instead of having to return an MLIR internal reference type.
- Added [AnyPointer.move\\_into\(\)](#) method, for moving a value from one pointer memory location to another.
- Added built-in [hex\(\)](#) function, which can be used to format any value whose type implements the [Intable](#) trait as a hexadecimal string.
- [PythonObject](#) now implements `__is__` and `__isnot__` so that you can use expressions of the form `x is y` and `x is not y` with `PythonObject`.
- [PythonObject](#) now conforms to the `SizedRaising` trait. This means the built-in [len\(\)](#) function now works on `PythonObject`.
- The `os` package now contains the [stat\(\)](#) and [lstat\(\)](#) functions.
- A new [os.path](#) package now allows you to query properties on paths.
- The `os` package now has a [PathLike](#) trait. A struct conforms to the `PathLike` trait by implementing the `__fspath__()` function.
- The [pathlib.Path](#) now has functions to query properties of the path.

- The [listdir\(\)](#) method now exists on [pathlib.Path](#) and also exists in the `os` module to work on `PathLike` structs. For example, the following sample lists all the directories in the `/tmp` directory:

```
from pathlib import Path

fn walktree(top: Path, inout files: DynamicVector[Path]):
    try:
        var ls = top.listdir()
        for i in range(len(ls)):
            var child = top / ls[i]
            if child.is_dir():
                walktree(child, files)
            elif child.is_file():
                files.append(child)
            else:
                print("Skipping '" + str(child) + "'")
    except:
        return

fn main():
    var files = DynamicVector[Path]()

    walktree(Path("/tmp"), files)

    for i in range(len(files)):
        print(files[i])
```

- The [find\(\)](#), [rfind\(\)](#), [count\(\)](#), and [\\_\\_contains\\_\\_\(\)](#) methods now work on string literals. This means that you can write:

```
if "Mojo" in "Hello Mojo":
    ...
```

- Breakpoints can now be inserted programmatically within the code using the builtin [breakpoint\(\)](#) function.  
Note: on Graviton instances, the debugger might not be able to resume after hitting this kind of breakpoint.
- Added a builtin [Boolable](#) trait that describes a type that can be represented as a boolean value. To conform to the trait, a type must implement the `__bool__()` method.
- Modules within packages can now use purely relative `from` imports:

```
from . import another_module
```

- Trivial types, like MLIR types and function types, can now be bound implicitly to traits that require copy constructors or move constructors, such as [Movable](#), [Copyable](#), and [CollectionElement](#).
- A new magic `__lifetime_of(expr)` call will yield the lifetime of a memory value. We hope and expect that this will eventually be replaced by `Reference(expr).lifetime` as the parameter system evolves, but this is



important in the meantime for use in function signatures.

- A new magic `__type_of(expr)` call will yield the type of a value. This allows one to refer to types of other variables. For example:

```
fn my_function(x: Int, y: __type_of(x)) -> Int:
  let z: __type_of(x) = y
  return z
```

## Changed

- As another step towards [removing let declarations](#) we have removed support for let declarations inside the compiler. To ease migration, we parse `let` declarations as a `var` declaration so your code won't break. We emit a warning about this, but please switch your code to using `var` explicitly, because this migration support will be removed in a subsequent update.

```
fn test():
  # treated as a var, but please update your code!
  let x = 42 # warning: 'let' is being removed, please use 'var' instead
  x = 9
```

- It is no longer possible to explicitly specify implicit argument parameters in [automatically parameterized functions](#). This ability was an oversight and this is now an error:

```
fn autoparameterized(x: SIMD):
  pass
```

```
autoparameterized[DType.int32, 1](3) # error: too many parameters
```

- `vectorize_unroll` has been removed, and [vectorize](#) now has a parameter named `unroll_factor` with a default value of 1. Increasing `unroll_factor` may improve performance at the cost of binary size. See the [loop unrolling blog here](#) for more details.
- The `vectorize` signatures have changed with the closure `func` moved to the first parameter:

```
vectorize[func, width, unroll_factor = 1](size)
vectorize[func, width, size, unroll_factor = 1]()
```

The doc string has been updated with examples demonstrating the difference between the two signatures.

- The `unroll` signatures have changed with the closure `func` moved to the first parameter:

```
unroll[func, unroll_count]()
```

- The signature of the [NDBuffer](#) and [Buffer](#) types have changed. Now, both take the type as the first parameter and no longer require the shape parameter. This allows you to use these types and have sensible defaults. For example:

```
NDBuffer[DType.float32, 3]
```

is equivalent to

```
NDBuffer[DType.float32, 3, DimList.create_unknown[3]()]
```

Users can still specify the static shape (if known) to the type:

```
NDBuffer[DType.float32, 3, DimList(128, 128, 3)]
```

- The error message for missing function arguments is improved: instead of describing the number of arguments (e.g. callee expects at least 3 arguments, but 1 was specified) the missing arguments are now described by name (e.g. missing 2 required positional arguments: 'b', 'c').
- The [CollectionElement](#) trait is now a built-in trait and has been removed from `collections.vector`.
- The `DynamicVector(capacity: Int)` constructor has been changed to take `capacity` as a keyword-only argument to prevent implicit conversion from `Int`.
- [Variant.get\[T\]\(\)](#) now returns a [Reference](#) to the value rather than a copy.
- The [String](#) methods `tolower()` and `toupper()` have been renamed to `str.lower()` and `str.upper()`.
- The `ref` and `mutref` identifiers are no longer reserved as Mojo keywords. We originally thought about using those as language sugar for references, but we believe that generic language features combined with the [Reference](#) type will provide a good experience without dedicated sugar.

## Fixed

- [#435](#) Structs with Self type don't always work.
- [#1540](#) Crash in register\_passable self referencing struct.
- [#1664](#) - Improve error message when `StaticTuple` is constructed with a negative size for the number of elements.
- [#1679](#) - crash on SIMD of zero elements.
- Various crashes on invalid code: [#1230](#), [#1699](#), [#1708](#)
- [#1223](#) - Crash when parametric function is passed as (runtime) argument. The parser now errors out instead.
- [#1530](#) - Crash during diagnostic emission for parameter deduction failure.
- [#1538](#) and [#1607](#) - Crash when returning type value instead of instance of expected type. This is a common mistake and the error now includes a hint to point users to the problem.

- [#1613](#) - Wrong type name in error for incorrect `self` argument type in trait method declaration.
- [#1670](#) - Crash on implicit conversion in a global variable declaration.
- [#1741](#) - Mojo documentation generation doesn't show `inout` / `owned` on variadic arguments.
- [#1621](#) - VS Code does not highlight `raises` and `capturing` in functional type expressions.
- [#1617](#) - VS Code does not highlight `fn` in specific contexts.
- [#1740](#) - LSP shows unrelated info when hovering over a struct.
- [#1238](#) - File shadows Mojo package path.
- [#1429](#) - Crash when using nested import statement.
- [#1322](#) - Crash when missing types in variadic argument.
- [#1314](#) - Typecheck error when binding alias to parametric function with default argument.
- [#1248](#) - Crash when importing from file the same name as another file in the search path.
- [#1354](#) - Crash when importing from local package.
- [#1488](#) - Crash when setting generic element field.
- [#1476](#) - Crash in interpreter when calling functions in parameter context.
- [#1537](#) - Crash when copying parameter value.
- [#1546](#) - Modify nested vector element crashes parser.
- [#1558](#) - Invalid import causes parser to crash.
- [#1562](#) - Crash when calling parametric type member function.
- [#1577](#) - Crash when using unresolved package as a variable.
- [#1579](#) - Member access into type instances causes a crash.
- [#1602](#) - Interpreter failure when constructing strings at compile time.
- [#1696](#) - Fixed an issue that caused syntax highlighting to occasionally fail.
- [#1549](#) - Fixed an issue when the shift amount is out of range in `SIMD.shift_left` and `SIMD.shift_right`.

## v0.7.0 (2024-01-25)

### ★ New

- A new Mojo-native dictionary type, [Dict](#) for storing key-value pairs. `Dict` stores values that conform to the [CollectionElement](#) trait. Keys need to conform to the new [KeyElement](#) trait, which is not yet implemented by other standard library types. In the short term, you can create your own wrapper types to use as keys. For example, the following sample defines a `StringKey` type and uses it to create a dictionary that maps strings to `Int` values:

```
from collections.dict import Dict, KeyElement

@value
```

```

struct StringKey(KeyElement):
    var s: String

    fn __init__(inout self, owned s: String):
        self.s = s ^

    fn __init__(inout self, s: StringLiteral):
        self.s = String(s)

    fn __hash__(self) -> Int:
        return hash(self.s)

    fn __eq__(self, other: Self) -> Bool:
        return self.s == other.s

fn main() raises:
    var d = Dict[StringKey, Int]()
    d["cats"] = 1
    d["dogs"] = 2
    print(len(d))          # prints 2
    print(d["cats"])       # prints 1
    print(d.pop("dogs"))   # prints 2
    print(len(d))          # prints 1

```

We plan to add `KeyElement` conformance to standard library types in subsequent releases.

- Users can opt-in to assertions used in the standard library code by specifying `-D MOJO_ENABLE_ASSERTIONS` when invoking `mojo` to compile your source file(s). In the case that an assertion is fired, the assertion message will be printed along with the stack trace before the program exits. By default, assertions are *not enabled* in the standard library right now for performance reasons.
- The Mojo Language Server now implements the References request. IDEs use this to provide support for **Go to References** and **Find All References**. A current limitation is that references outside of the current document are not supported, which will be addressed in the future.
- The [sys.info](#) module now includes `num_physical_cores()`, `num_logical_cores()`, and `num_performance_cores()` functions.
- Homogeneous variadic arguments consisting of memory-only types, such as `String` are more powerful and easier to use. These arguments are projected into a [VariadicListMem](#).

(Previous releases made it easier to use variadic lists of register-passable types, like `Int`.)

Subscripting into a `VariadicListMem` now returns the element instead of an obscure internal type. In addition, we now support `inout` and `owned` variadic arguments:

```

fn make_worldly(inout *strs: String):
    # This "just works" as you'd expect!
    for i in range(len(strs)):
        strs[i] += " world"
fn main():

```

```

var s1: String = "hello"
var s2: String = "konnichiwa"
var s3: String = "bonjour"
make_worldly(s1, s2, s3)
print(s1) # hello world
print(s2) # konnichiwa world
print(s3) # bonjour world

```

(Previous releases made it easier to use variadic lists, but subscripting into a `VariadicListMem` returned a low-level pointer, which required the user to call `__get_address_as_lvalue()` to access the element.)

Note that subscripting the variadic list works nicely as above, but iterating over the variadic list directly with a `for` loop produces a [Reference](#) (described below) instead of the desired value, so an extra subscript is required; We intend to fix this in the future.

```

fn make_worldly(inout *strs: String):
    # Requires extra [] to dereference the reference for now.
    for i in strs:
        i[] += " world"

```

Heterogeneous variadic arguments have not yet been moved to the new model, but will in future updates.

Note that for variadic arguments of register-passable types like `Int`, the variadic list contains values, not references, so the dereference operator (`[]`) is not required. This code continues to work as it did previously:

```

fn print_ints(*nums: Int):
    for num in nums:
        print(num)
    print(len(nums))

```

- Mojo now has a prototype version of a safe [Reference](#) type. The compiler's lifetime tracking pass can reason about references to safely extend local variable lifetime, and check indirect access safety. The `Reference` type is brand new (and currently has no syntactic sugar) so it must be explicitly dereferenced with an empty subscript: `ref[]` provides access to the underlying value.

```

fn main():
    var a: String = "hello"
    var b: String = " references"

    var aref = Reference(a)
    aref[] += b
    print(a) # prints "hello references"

    aref[] += b
    # ^last use of b, it is destroyed here.

```

```
print(aref[]) # prints "hello references"
# ^last use of a, it is destroyed here.
```

While the `Reference` type has the same in-memory representation as a C pointer or the `Mojo Pointer` type, it also tracks a symbolic "lifetime" value so the compiler can reason about the potentially accessed set of values. This lifetime is part of the static type of the reference, so it propagates through generic algorithms and abstractions built around it.

The `Reference` type can form references to both mutable and immutable memory objects, e.g. those on the stack or borrowed/inout/owned function arguments. It is fully parametric over mutability, eliminating the [problems with code duplication due to mutability specifiers](#) and provides the base for unified user-level types. For example, it could be used to implement an array slice object that handles both mutable and immutable array slices.

While this is a major step forward for the lifetimes system in Mojo, it is still *very* early and awkward to use. Notably, there is no syntactic sugar for using references, such as automatic dereferencing. Several aspects of it need to be more baked. It is getting exercised by variadic memory arguments, which is why they are starting to behave better now.

Note: the safe `Reference` type and the unsafe pointer types are defined in the same module, currently named `memory.unsafe`. We expect to restructure this module in a future release.

- Mojo now allows types to implement `__refattr__()` and `__refitem__()` to enable attribute and subscript syntax with computed accessors that return references. For common situations where these address a value in memory this provides a more convenient and significantly more performant alternative to implementing the traditional get/set pairs. Note: this may be changed in the future when references auto-dereference—at that point we may switch to just returning a reference from `__getattr__()`.
- Parametric closures can now capture register passable typed values by copy using the `__copy_capture` decorator. For example, the following code will print `5`, not `2`.

```
fn foo(x: Int):
    var z = x

    @__copy_capture(z)
    @parameter
    fn formatter() -> Int:
        return z
    z = 2
    print(formatter())

fn main():
    foo(5)
```

- `String` now implements `KeyElement` and may be used as a key in `Dict`.
- More robust support for structs with fields of self referencing types. For example, the following code will work and print `0`:

```

struct Foo(CollectionElement):
    var vec: DynamicVector[Self]

    fn __init__(inout self: Self):
        self.vec = DynamicVector[Self]()

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

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

fn main():
    var foo = Foo()
    print(len(foo.vec))

```

## Removed

- The `__takeinit__` special constructor form has been removed from the language. This "non-destructive move" operation was previously wired into the `x^` transfer operator, but had unpredictable behavior that wasn't consistent. Now that Mojo has traits, it is better to model this as an explicit `.take()` operation on a type, which would transfer out the contents of the type without ending its lifetime. For example, for a type that holds a pointer, `take()` might return a new instance pointing to the same data, and null out its own internal pointer. This change makes it clear when a lifetime is ended versus when the contents of an LValue are explicitly taken.
- The current implementation of autotuning has been deprecated, as Mojo's autotuning implementation is undergoing a redesign. Tutorials around the current implementation have also been removed as they are being rewritten.

Consequently, the `autotune()`, `autotune_fork()`, and `search()` functions have been removed from the standard library.

- The `_OldDynamicVector` type that worked only on register passable element types has been removed. Please migrate uses to [DynamicVector](#) which works on both register passable and memory types.
- The `UnsafeFixedVector` in `utils.vector` has been removed. We recommend using either [DynamicVector](#) or [InlinedFixedVector](#) instead.
- The `@adaptive` decorator has been removed from the language. Any uses of the decorator in a non-search context can be replaced with `@parameter if`. For example:

```

@adaptive
fn foo[a: Bool]():
    constrained[a]()
    body1()

```

```

@adaptive

```

```
fn foo[a: Bool]() :
  constrained[not a]()
  body2()
```

Can be rewritten as:

```
fn foo[a: Bool]() :
  @parameter
  if a:
    body1()
  else:
    body2()
```

Consequently, the special `__adaptive_set` attribute has been removed as well.

- Result parameters have been removed from Mojo. Result parameter declarations in function parameter lists are no longer allowed, nor are forward alias declarations. This includes removing the `param_return` statement.
- The `@noncapturing` and `@closure` decorators have been removed due to refinements and improvements to the closure model. See below for more details!

## Changed

- The Mojo closure model has been refined to be more straightforward and safe. Mojo has two closure types: parameter closures and runtime closures. Parameter closures can be used in higher-order functions and are the backbone of functions like `vectorize` and `parallelize`. They are always denoted by `@parameter` and have type `fn() capturing -> T` (where `T` is the return type).

On the other hand, runtime closures are always dynamic values, capture values by invoking their copy constructor, and retain ownership of their capture state. You can define a runtime closure by writing a nested function that captures values:

```
fn outer(b: Bool, x: String) -> fn() escaping -> None:
  fn closure():
    print(x) # 'x' is captured by calling String.__copyinit__

  fn bare_function():
    print("hello") # nothing is captured

  if b:
    # closure can be safely returned because it owns its state
    return closure^

# function pointers can be converted to runtime closures
return bare_function
```



The type of runtime closures are of the form `fn() escaping -> T`. You can pass equivalent function pointers as runtime closures.

Stay tuned for capture list syntax for move capture and capture by reference, and a more unified closure model!

- The `@unroll(n)` decorator can now take a parameter expression for the unroll factor, i.e. `n` can be a parameter expression that is of integer type.
- The `cpython` module in the `python` package has been moved to be an internal module, i.e. `_cpython`.
- `AnyType` and `Destructable` have been unified into a single trait, `AnyType`. Every nominal type (i.e. all structs) now automatically conform to `AnyType`.
- Previously, the `mojo` package command would output a Mojo package that included both partly-compiled Mojo code, as well as fully-compiled machine code for a specific computer architecture -- the architecture of the machine being used to invoke the `mojo` package command.

Now, `mojo` package only includes partly-compiled Mojo code. It is only fully compiled for the specific computer architecture being used at the point that the package is first `import`-ed. As a result, Mojo packages are smaller and more portable.

- The `simd_width` and `dtype` parameters of `polynomial_evaluate` have been switched. Based on the request in [#1587](#), the `polynomial_evaluate` function has also been extended so that the `coefficients` parameter can take either a either a [StaticTuple](#) or a [VariadicList](#).
- As a tiny step towards removing `let` declarations, this release removes the warning: `'var' was never mutated, consider switching to a 'let'`.

## Fixed

- [#1595](#) - Improve error message when trying to materialize `IntLiteral` in runtime code.
- Raising an error from the initializer of a memory-only type now works correctly in the presence of complex control flow. Previously Mojo could run the destructor on `self` before it was initialized when exiting with an error.
- [#1096](#) - Improve warning messages for dead code in conditionals like `or` expressions.
- [#1419](#) - Fix assertion failure with uninitialized lattice values.
- [#1402](#) - Fix movable trait not detected on recursive struct implemented with `AnyPointer`.
- [#1399](#) - Fix parser crash when a parameter type in a struct that implements a trait is misspelled.
- [#1152](#) - Allow mutable `self` argument when overloading operators using dunder methods.
- [#1493](#) - Fix crash in `DynamicVector` copy constructor in certain situations.
- [#1316](#) - The `benchmark.keep` function now properly handles vector types.
- [#1505](#) - The `simd.shuffle` operation now works on 64 element permutations.
- [#1355](#) - Fix `String.find()` returning wrong value when starting index is non-zero.
- [#1367](#) - Fix `String.replace()` returning incorrect results for multi-character search strings.

- [#1535](#) - Invalid error field 'w.x.y' destroyed out of the middle of a value, preventing the overall value from being destroyed.
- [#1475](#) - Assertion failure in nested loop.
- [#1591](#) - Assertion failure when using `AnyType` struct member.
- [#1503](#) - Rename the mojo build of LLDB to `mojo-lldb`, to prevent name collisions with the system's LLDB.
- [#1542](#) - `@unroll` does not accept alias as unroll factor.
- [#1443](#) - Compiler crash on variadic list of traits.
- [#1604](#) - Variable of trivial type not destroyed by transferring ownership.
- [#1341](#) - Segmentation fault when passing closures around.
- [#217](#) - Closure state is stack allocated.

## v0.6.1 (2023-12-18)

### ★ New

- The Mojo REPL now provides limited support for the `%cd` magic command.

This command automatically maintains an internal stack of directories you visit during the REPL session. Usage:

- `%cd 'dir'` : change to directory `dir` and push it on the directory stack.
- `%cd -` : pop the directory stack and change to the last visited directory.

- Structs decorated with `@value` now automatically conform to the [Movable](#) and [Copyable](#) built-in traits.
- [String](#) now has new [toupper\(\)](#) and [tolower\(\)](#) methods analogous, respectively, to Python's `str.toupper()` and `str.tolower()`.
- Added a [hash\(\)](#) built-in function and [Hashable](#) trait for types implementing the `__hash__()` method. Future releases will add `Hashable` support to Standard Library types. In the meantime, the `hash` module includes a version of the `hash()` function that works on arbitrary byte strings. To generate hashes for [SIMD](#) types, you use the internal `_hash_simd()` function:

```
from builtin.hash import _hash_simd

fn gen_simd_hash():
    let vector = SIMD[DType.int64, 4](1, 2, 3, 4)
    let hash = _hash_simd(vector)
```

- Several standard library types now conform to the [CollectionElement](#) trait. These types include [Bool](#), [StringLiteral](#), [DynamicVector](#), [Tensor](#), [TensorShape](#), and [TensorSpec](#).

### 🦋 Changed

- `utils.vector` has been moved to a new `collections` package to make space for new collections. This means that if you had previous code that did `from utils.vector import DynamicVector`, it now needs to be `from collections.vector import DynamicVector` due to the move.
- The special destructor method `__del__()` has been changed to enforce that it cannot raise an error. Raising destructors are not supported properly at the moment.

## Fixed

- [#1421](#) - Fixed a crash when using Tuples in the REPL.
- [#222](#) - Generate an error for obviously self recursive functions.
- [#1408](#) - Fix overload resolution when candidates can return generic types.
- [#1413](#) and [#1395](#) - Do not crash when re-declaring a builtin declaration.
- [#1307](#) - Fix compatibility of function signatures that only differ in default argument values.
- [#1380](#) - Fix printing of empty `String`.

## v0.6.0 (2023-12-04)

## Legendary

- Traits have arrived!

You can now define a *trait*, which consists of a required set of method prototypes. A struct can *conform to* the trait by implementing these methods. This lets you write generic functions that work on any structs that conform to a given trait.

The following section gives a brief overview of traits—see the [Mojo Manual](#) and this [traits blog post](#) for more details!

Traits are declared with the `trait` keyword. The bodies of traits should contain method signatures declared with `...` as their bodies. Default method implementations are not supported yet.

```
trait SomeTrait:  
    fn required_method(self, x: Int): ...
```

The trait can be implemented on a struct by inheriting from it.

```
struct SomeStruct(SomeTrait):  
    fn required_method(self, x: Int):  
        print("hello traits", x)
```

You can then write a generic functions that accepts any type that conforms to the trait. You do this by creating a parameterized function with a trait-typed parameter:

```
fn fun_with_traits[T: SomeTrait](x: T):  
    x.required_method(42)
```

Which can be invoked with instances of types that conform to the trait:

```
var thing = SomeStruct()  
# Infer the parameter `T`!  
fun_with_traits(thing)
```

Traits can also inherit from other traits, which simply requires that implementers of the child trait also conform to all parent traits.

```
trait Parent:  
    fn parent_func(self): ...  
  
trait Child(Parent):  
    fn child_func(self): ...
```

Then, both child and parent trait methods can be invoked on instances of the trait `Child`. As well, an instance of the child trait can be converted to an instance of the parent trait.

```
fn the_parents[T: Parent](x: T):  
    x.parent_func()  
  
fn the_children[T: Child](x: T):  
    x.child_func()  
    x.parent_func()  
    # Upcast `x` from instance of `Child` to `Parent`.  
    the_parents(x)
```

For more information, see the [Traits page](#) in the Mojo Manual.

- A fundamental `Destructable` trait has been added to the language. This is a core trait that every trait automatically conforms to. This enables destruction of generic types and generic collections.

**Note:** We're aware that this trait might be better spelled `Destructible`. We're planning on removing it in the future and moving its functionality to `AnyType` so that any type that doesn't provide its own destructor will have a default, no-op destructor.

- We've added some traits to the standard library, you can implement these on your own types:
  - [Destructable](#)
  - [Copyable](#)
  - [Movable](#)

- [Stringable](#)
- [Intable](#)
- [Sized](#)
- [CollectionElement](#)
- We added built-in [len\(\)](#), [str\(\)](#), and [int\(\)](#) functions, which work with types that implement the `Sized`, `Stringable`, and `Intable` traits, respectively.
- [DynamicVector](#) is now a proper generic collection that can use any type that implements the `Movable` and `Copyable` traits. This means you can now write, for example, `DynamicVector[String]`. Also, `DynamicVector` now invokes its element destructors upon destruction, so `_del_old` has been deleted.
- `print` now works on any types that implement `Stringable` by invoking their `__str__` method:

```
@value
struct BoxedInt(Stringable):
    var value: Int

    fn __str__(self) -> String:
        return self.value

print(BoxedInt(11), "hello traits!", BoxedInt(42))
```

## ★ New

- The [Mojo Manual](#) is an all-new, complete Mojo user guide. It doesn't include *everything* about Mojo yet, but it includes a lot, and more than the original programming manual (now deprecated).  
  
Plus, the entire Mojo Manual and other Mojo docs are now [open-sourced on GitHub](#), and we'd love to accept contributions to help us improve them!
- Mojo now supports partial automatic parameterization: when a function is declared with an argument of a partially bound type, the unbound parameters of that type are implicitly added to the function's input parameters. For example:

```
@value
struct Fudge[a: Int, b: Int, c: Int = 7]: ...

# These function declarations are roughly equivalent:
fn eat(f: Fudge[5]): ...           # implicitly parameterized
fn eat[_b: Int](f: Fudge[5, _b]): ... # explicitly parameterized
```

In the first signature for `eat()`, the `b` parameter isn't bound, so it's *implicitly* added as an input parameter on the function.

In the second signature for `eat()`, the author has explicitly defined an input parameter (`_b`), which is bound to the second parameter on the argument type (which happens to be `b`).

Both functions can be called like this:

```
eat(Fudge[5, 8]())
```

Mojo infers the value of the `b` parameter from the argument (in this case, 8).

With the second signature, you can also pass the `_b` parameter value explicitly:

```
eat[3](Fudge[5, 3]())
```

Moreover, Mojo now allows you to explicitly mark parameters as unbound using the `_` as syntax meaning "placeholder for an unbound parameter." For example:

```
# These function declarations are roughly equivalent:
fn eat(f: Fudge[5, _, c=_]): ...           # implicitly parameterized
fn eat(f: Fudge[c=_, a=5, b=_]): ...       # implicitly parameterized
fn eat[_b: Int, _c: Int](f: Fudge[5, _b, _c]): ... # explicitly parameterized
```

The first two signatures explicitly unbind the `b` and `c` parameters.

In the last signature, the `_b` and `_c` parameters are explicitly declared by the author, and bound to the `b` and `c` parameters in the argument type.

Any of these signatures can be called like this:

```
eat(Fudge[5, 8]())
eat(Fudge[5, 8, 9]())
```

Note that the default parameter values of struct parameters are bound, unless explicitly unbound by the user.

For more information, see the [Mojo Manual](#).

- Parametric types can now be partially bound in certain contexts. For example, a new `Scalar` type alias has been added defined as:

```
alias Scalar = SIMD[size=1]
```

Which creates a parametric type alias `Scalar` with a single parameter of type `DType`. Types can also be partially or fully bound in other contexts. For instance, `alias` declarations of type values inside functions now work properly:

```
fn type_aliases():
  alias T = SIMD
  print(T[DType.float32, 1]())
  alias Partial = T[type=DType.int32]
  print(Partial[2]())
```

- The `__mlir_op` feature now supports operations that return multiple results. To use them, you write the `_type` field as a `Tuple` of types. For example:

```
# The `ret` variable has type `Tuple[Int, Int]`.
let ret = __mlir_op.`multi_result_op`[_type=(Int, Int)]()
```

- Mojo now has the ability to read raw bytes from a file using the [read\\_bytes\(\)](#) method. For example:

```
with open("file.binary", "r") as f:
    data = f.read_bytes()
```

- A size argument was added to the [read\(\)](#) and [read\\_bytes\(\)](#) methods on the builtin `file.FileHandle`. The size argument defaults to -1 and maintains the previous "read to EOF" behavior when size is negative.

```
with open("file.binary", "r") as f:
    data1 = f.read_bytes(1024)
    data2 = f.read_bytes(256)
```

- [Path](#) now has `read_bytes()` and `read_text()` methods to read file contents from a path:

```
let text_path = Path("file.txt")
let text = text_path.read_text()

let binary_path = Path("file.binary")
let data = binary_path.read_bytes()
```

- [Tensor](#) has new `save()` and `load()` methods to save and load to file. These methods preserve shape and datatype information. For example:

```
let tensor = Tensor[DType.float32]()
tensor.save(path)

let tensor_from_file = Tensor[DType.float32].load(path)
```

- Subscripting added to [DTypePointer](#) and [Pointer](#):

```
let p = DTypePointer[DType.float16].alloc(4)
for i in range(4):
    p[i] = i
    print(p[i])
```

- `file.FileHandle` now has a `seek()` method.
- [String](#) now has an [rfind\(\)](#) method analogous to Python's `str.rfind()`.

- `String` now has an [`split\(\)`](#) method analogous to Python's `str.split()`.
- [`Path`](#) now has a [`suffix\(\)`](#) method analogous to Python's `pathlib.Path.suffix`.
- The Mojo REPL now supports indented expressions, making it a bit easier to execute expressions copied from an indented block (such as a doc string).
- The Mojo Language Server now implements the Document Symbols request. IDEs use this to provide support for **Outline View** and **Go to Symbol**. This addresses [Issue #960](#).
- The Mojo Language Server now shows documentation when code completing modules or packages in `import` statements.
- The Mojo Language Server now supports processing code examples, defined as markdown Mojo code blocks, inside of doc strings. This enables IDE features while writing examples in API documentation.
- The Mojo Language Server now provides semantic token information, providing better highlighting for symbols whose semantics are not statically analyzable.
- The Mojo Language Server now classifies doc strings as folding ranges, making them easier to collapse, reducing vertical space while editing.
- Command line options for the `mojo` driver that take arguments can now be written in either of two ways: both `--foo F00` and `--foo=F00`. Previously, only the former was valid.

## Changed

- Variadic list types [`VariadicList`](#) and [`VariadicListMem`](#) are now iterable. Variadic arguments are automatically projected into one of these types inside the function body, so `var` args can be iterated:

```
fn print_ints(*nums: Int):
    for num in nums:
        print(num)
    print(len(nums))
```

- The assert functions in the [`testing`](#) package now raise an `Error` when the assertion fails instead of returning a `Bool` for whether the assertion succeeded or not.
- Parameters of [`AnyType`](#) type are no longer (implicitly) assumed to be register-passable. A new `AnyRegType` type is used to represent generic types that are register passable.
- Changing the units in a [`benchmark`](#) report is now an argument instead of a parameter:

```
let report = benchmark.run[timer]()
report.print(Unit.ms)
```

- Default values on `inout` arguments are no longer permitted, i.e. the following will now raise an error:



```
fn inout_default(inout x: Int = 2): ...
```

- The `to_string()` function has been removed from [PythonObject](#) in favor of the new `__str__()` function. This composes better with traits so it can be used with the generic `str()` function.

## Fixed

- [#734](#) - Consumption of struct works only for types with a `__del__` method.
- [#910](#) - Parser crash when using memory-only generic type as return of function that `raise s`.
- [#1060](#) - Mojo happily parses code that has messed up indentation
- [#1159](#) - The language server doesn't warn about bad return type.
- [#1166](#) - warning: unreachable code after return statement with context manager
- [#1098](#) - The language server doesn't highlight properties of PythonObjects correctly.
- [#1153](#) - The language server crashes when parsing an invalid multi-nested module import.
- [#1236](#) - The language server doesn't show autocomplete in if statements.
- [#1246](#) - Warning diagnostics are transient in the presence of caching.

## Known Issue

- There is an issue affecting Jupyter notebooks that use autotuning and traits. This issue only manifests on macOS, and the same code runs without issue outside of the notebooks. This issue affects the *Matrix multiplication in Mojo* notebook.

## v0.5.0 (2023-11-2)

## New

- The [SIMD](#) type now defaults to the architectural SIMD width of the type. This means you can write `SIMD[DType.float32]` which is equivalent to `SIMD[DType.float32, simdwidthof[DType.float32]()]`.
- The [SIMD](#) type now contains a `join()` function that allows you to concatenate two `SIMD` values together and produce a new `SIMD` value.
- Mojo now supports compile-time *keyword parameters*, in addition to existing support for [keyword arguments](#). For example:

```
fn foo[a: Int, b: Int = 42]():  
    print(a, "+", b)
```

```
foo[a=5]()          # prints '5 + 42'
foo[a=7, b=13]()    # prints '7 + 13'
foo[b=20, a=6]()    # prints '6 + 20'
```

Keyword parameters are also supported in structs:

```
struct KwParamStruct[a: Int, msg: String = "🔥mojo🔥"]:
  fn __init__(inout self):
    print(msg, a)

fn use_kw_params():
  KwParamStruct[a=42]()          # prints '🔥mojo🔥 42'
  KwParamStruct[5, msg="hello"]() # prints 'hello 5'
  KwParamStruct[msg="hello", a=42]() # prints 'hello 42'
```

For more detail, see the [Mojo Manual](#).

For the time being, the following notable limitations apply:

- Keyword-only parameters are **not supported** yet:

```
fn baz[*args: Int, b: Int](): pass # fails
fn baz[a: Int, *, b: Int](): pass # fails
```

(The analogous keyword-only arguments in Python are described in [PEP 3102](#).)

- Variadic keyword parameters are **not supported** yet:

```
fn baz[a: Int, **kwargs: Int](): pass # fails
```

- Mojo now supports "automatic" parameterization of functions. What this means is that if a function argument type is parametric but has no bound parameters, they are automatically added as input parameters on the function. This works with existing features to allow you to write parametric functions with less boilerplate.

```
@value
struct Thing[x: Int, y: Int]:
  pass

fn foo(v: Thing):
  print(v.x)
  print(v.y)

fn main():
  let v = Thing[2, 3]()
  foo(v)
```

However, partial autoperparameterization is **not supported** yet:

```
fn foo(v: Thing[y=7]): # Partially bound type not allowed yet.  
...
```

- Keyword argument passing is supported when invoking `__getitem__` using the bracket syntax:

```
@value  
struct MyStruct:  
    fn __getitem__(self, x: Int, y: Int, z: Int) -> Int:  
        return x * y + z  
  
MyStruct()[z=7, x=3, y=5] # returns 22
```

However, keyword argument passing to `__setitem__` using the bracket syntax is **not supported** yet:

```
@value  
struct OtherStruct:  
    fn __setitem__(self, x: Int, y: Int): pass  
  
OtherStruct()[x=1] = 4 # fails
```

- Function argument input parameters can now be referenced within the signature of the function:

```
fn foo(x: SIMD, y: SIMD[x.type, x.size]):  
    pass
```

- The [benchmark](#) module has been simplified and improved so you can now run:

```
import benchmark  
from time import sleep  
  
fn sleeper():  
    sleep(.01)  
  
fn main():  
    let report = benchmark.run[sleeper]()  
    print(report.mean())
```

It no longer requires a capturing `fn` so can benchmark functions outside the same scope.

You can print a report with:

```
report.print()
```

```
-----  
Benchmark Report (s)
```

```
-----
Mean: 0.012314264957264957
Total: 1.440769
Iters: 117
Warmup Mean: 0.0119335
Warmup Total: 0.023866999999999999
Warmup Iters: 2
Fastest Mean: 0.012227958333333334
Slowest Mean: 0.012442699999999999
```

Units for all functions default to seconds, but can be changed with:

```
from benchmark import Unit

report.print[Unit.ms]()
```

- Mojo now supports struct parameter deduction (a.k.a. class template argument deduction, or CTAD) for partially bound types. Struct parameter deduction is also possible from static methods. For example:

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

struct CtdStructWithDefault[a: Int, b: Int, c: Int = 8]:
  fn __init__(inout self, x: Thing[a]):
    print("hello", a, b, c)

  @staticmethod
  fn foo(x: Thing[a]):
    print("🔥", a, b, c)

fn main():
  _ = CtdStructWithDefault[b=7](Thing[6]()) # prints 'hello 6 7 8'
  CtdStructWithDefault[b=7].foo(Thing[6]()) # prints '🔥 6 7 8'
```

- [Tensor](#) has new `fromfile()` and `tofile()` methods to save and load as bytes from a file.
- The built-in `print()` function now works on the [Tensor](#) type.
- [TensorShape](#) and [TensorSpec](#) now have constructors that take [DynamicVector\[Int\]](#) and [StaticIntTuple](#) to initialize shapes.
- The [String](#) type now has the `count()` and `find()` methods to enable counting the number of occurrences or finding the offset index of a substring in a string.
- The `String` type now has a `replace()` method which allows you to replace a substring with another string.



Changed

- [VariadicList](#) and [VariadicListMem](#) moved under builtins, and no longer need to be imported.
- Variadic arguments are now automatically projected into a `VariadicList` or `VariadicListMem` inside the function body. This allows for more flexibility in using var args. For example:

```
fn print_ints(*nums: Int):
    let len = len(nums)
    for i in range(len):
        print(nums[i])
    print(len)
```

- The parameters for [InlinedFixedVector](#) have been switched. The parameters are now `[type, size]` instead of `[size, type]`. The `InlinedFixedVector` now has a default size which means that one can just use `InlinedFixedVector` as `InlinedFixedVector[Float32]` and the default size is used.
- `write_file()` method in [Buffer](#) and [NDBuffer](#) is renamed to `tofile()` to match the Python naming.
- Mojo will now utilize all available cores across all NUMA sockets on the host machine by default. The prior default behavior was to use all the cores on the first socket.

## Removed

- The `math.numerics` module is now private, because its types (`FPUtils` and `FlushDenormals`) should not be used externally.

## Fixed

- [#532](#) - Compiler optimizing while True loop away
- [#760](#) - Compilation error: 'hlcf.foryield' op specifies 0 branch inputs but target expected 1 along control-flow edge from here
- [#849](#) - The `Tensor` type is now initialized with zeros at construction time.
- [#912](#) - Invalid load for `__get_address_as_lvalue`.
- [#916](#) - Parser crash when specifying default values for `inout` arguments.
- [#943](#) - Mojo hangs if you use `continue` in the nested loop
- [#957](#) - Parser crash when a function call with variadic arguments of a memory-only type is evaluated at compile time.
- [#990](#) - Fixes rounding issue with floor division with negative numerator.
- [#1018](#) - In some cases the sort function was returning invalid results. This release fixes some of these corner cases.
- [#1010](#) - Initializing tensor in alias declaration results in crash.
- [#1110](#) - The `time.now()` function now returns nanoseconds across all operating systems.
- [#1115](#) - cannot load non-register passable type into SSA register.

# 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[0]().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 __takeinit__(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.

For more information, see the Mojo Manual section on [move constructors](#).

- 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()`.

## 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

- 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`, `IO`, `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[DType.float32](height, width, channels)

# Declare the grayscale image.
var gray_scale_image = Tensor[DType.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__.🔥` 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 fn`s 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 `simdwidthof`, `bitwidthof`, 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 `simdwidthof`.
- The `dtype_` prefix has been dropped from `alignof`, `sizeof`, and `bitwidthof`. 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 Mojo Manual section on [parameters](#).

- 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 `SI32`, 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 of 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:

- fn `__moveinit__(inout self, owned existing: Self):` Traditional Rust style moving constructors that shuffles data around while taking ownership of the source binding.
- 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 `__setitem__` and `__getitem__` 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 `_CompilerBuiltin`. 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&`:
  - i. A future borrow checker will make sure there are no mutable aliases with an immutable borrow.
  - ii. `@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 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 be 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 `def`s). 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:

- i. 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 QoI for unless you're debugging `Int` itself. We need something like `__always_inline__`, `__nodebug__` attributes that clang uses in headers like `xmmintrin.h`.
- ii. Similarly, these "operators" should be treated by users as primitives: they don't want to know about MLIR or internal implementation details of `Int`.
- iii. 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.
- iv. 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 `DType`. 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. `FPUutils`, `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:
  - i. Added Integer and SIMD structs to bootstrap the standard library.
  - ii. 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:
  - i. It's simpler.
  - ii. 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.
  - iii. 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:

- i. 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`.
- ii. This doesn't "check to see if we can invoke `__new__`" it force applies it on a mismatch, which leads to poor QoI.
- iii. 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.

Was this page helpful?



Edit this page