

# Mojo roadmap & sharp edges

This document captures the broad plan about how we plan to implement things in Mojo, and some early thoughts about key design decisions. This is not a full design spec for any of these features, but it can provide a "big picture" view of what to expect over time. It is also an acknowledgement of major missing components that we plan to add.

## Overall priorities

Mojo is still in early development and many language features will arrive in the coming months. We are highly focused on building Mojo the right way (for the long-term), so we want to fully build-out the core Mojo language features before we work on other dependent features and enhancements.

Currently, that means we are focused on the core system programming features that are essential to [Mojo's mission](#), and as outlined in the following sections of this roadmap.

In the near-term, we will **not** prioritize "general goodness" work such as:

- Adding syntactic sugar and short-hands for Python.
- Adding features from other languages that are missing from Python (such as public/private declarations).
- Tackling broad Python ecosystem challenges like packaging.

If you have encountered any bugs with current Mojo behavior, please [submit an issue on GitHub](#).

If you have ideas about how to improve the core Mojo features, we prefer that you first look for similar topics or start a new conversation about it on [Discord](#).

We also consider Mojo to be a new member of the Python family, so if you have suggestions to improve the experience with Python, we encourage you to propose these "general goodness" enhancements through the formal [PEP process](#).

## Why not add syntactic sugar or other minor new features?

We are frequently asked whether Mojo will add minor features that people love in other languages but that are missing in Python, such as "implicit return" at the end of a function, public/private access control, fixing Python packaging, and various syntactic shorthands. As mentioned above, we are intentionally *not* adding these kinds of features to Mojo right now. There are three major reasons for this:

- First, Mojo is still young: we are still "building a house" by laying down major bricks in the type system and adding system programming features that Python lacks. We know we need to implement support for many existing Python features (compatibility is a massive and important goal of Mojo) and this work is not done yet. We have limited engineering bandwidth and want focus on building essential functionality, and we will not debate whether certain syntactic sugar is important or not.
- Second, syntactic sugar is like mortar in a building—its best use is to hold the building together by filling in usability gaps. Sugar (and mortar) is problematic to add early into a system: you can run into problems with laying the next bricks because the sugar gets in the way. We have experience building other languages (such as Swift) that added sugar early, which could have been subsumed by more general features if time and care were given to broader evaluation.
- Third, the Python community should tackle some of these ideas first. It is important to us that Mojo be a good member of the Python family, not just a language with Pythonic syntax. As such, we don't want to needlessly diverge from Python evolution: adding a bunch of features could lead to problems down the road if Python makes incompatible decisions. Such a future would fracture the community which would cause massively more harm than any minor language feature could offset.

For all these reasons, "nice to have" syntactic sugar is not a priority, and we will quickly close such proposals to avoid cluttering the issue tracker. If you'd like to propose a "general goodness" syntactic feature, please do so with the existing [Python PEP process](#). If/when Python adopts a feature, Mojo will also add it, because Mojo's goal is to be a superset. We are happy with this approach because the Python community is better equipped to evaluate these features, they have mature code bases to evaluate them with, and they have processes and infrastructure for making structured language evolution features.

## Small independent features

There are a number of features that are missing that are important to round out the language fully, but which don't depend strongly on other features. These include things like:

- Improved package management support.
- Many standard library features, including copy-on-write data structures.
- Support for "top level code" at file scope.
- Algebraic data types like `enum` in Swift/Rust, and pattern matching.
- Many standard library types need refinement, including `Optional[T]` and `Result[T, Error]`.

## Ownership and Lifetimes

The ownership system is partially implemented, and is expected to get built out in the next couple of months. The basic support for ownership includes features like:

- Capture declarations in closures.
- Lifetime checker: complain about invalid mutable references.

Mojo has support for a safe `Reference` type, and it is used in the standard library, but it is still under active development and not very pretty or nice to use right now.

## Traits support

Mojo has basic support for [traits](#). Traits allow you to specify a set of requirements for types to implement. Types can implement those requirements to *conform to* the trait. Traits allow you to write generic functions and generic containers, which can work with any type that conforms to a given trait, instead of being hard-coded to work with a specific type.

Currently, the only kind of requirements supported by traits are required method signatures. The trait can't provide a default implementation for its required methods, so each conforming type must implement all of the required methods.

A number of [built-in traits](#) are already implemented in the standard library.

We plan to expand traits support in future releases. Planned features include:

- Support for default implementations of required methods.
- Support for a feature like Swift's extensions, allowing you to add a trait to a preexisting type.
- Add support for conditional conformance.

## Classes

Mojo still doesn't support classes, the primary thing Python programmers use pervasively! This isn't because we hate dynamism - quite the opposite. It is because we need to get the core language semantics nailed down before adding them. We expect to provide full support for all the dynamic features in Python classes, and want the right framework to hang that off of.

When we get here, we will discuss what the right default is: for example, is full Python hash-table dynamism the default? Or do we use a more efficient model by default (e.g. vtable-based dispatch and explicitly declared stored properties) and allow opt'ing into dynamism with a `@dynamic` decorator on the class. More discussion is [in this proposal](#).

## C/C++ Interop

Integration to transparently import Clang C/C++ modules. Mojo's type system and C++'s are very compatible, so we should be able to have something pretty nice here. Mojo can leverage Clang to transparently generate a foreign function interface between C/C++ and Mojo, with the ability to directly import functions:

```
from "math.h" import cos

print(cos(0))
```

## Calling Mojo from Python

Currently you can call Python code from Mojo, but not the reverse: you can't pass a Mojo callback to a Python function, or build a Python extension in Mojo. We want to support calling Mojo from Python, but we want to do it right and we need the core language to be more mature first.

## Full MLIR decorator reflection

All decorators in Mojo have hard-coded behavior in the parser. In time, we will move these decorators to being compile-time metaprograms that use MLIR integration. This may depend on C++ interop for talking to MLIR. This completely opens up the compiler to programmers. Static decorators are functions executed at compile-time with the capability to inspect and modify the IR of functions and types.

```
fn value(t: TypeSpec):
    t.__copyinit__ = # synthesize dunder copyinit automatically

@value
struct TrivialType: pass

fn full_unroll(loop: mlir.Operation):
    # unrolling of structured loop

fn main():
    @full_unroll
    for i in range(10):
        print(i)
```

## Sharp Edges

The entire Modular kernel library is written in Mojo, and its development has been prioritized based on the internal needs of those users. Given that Mojo is still a young language, there are a litany of missing small features that many Python and systems programmers may expect from their language, as well as features that don't quite work the way we want to yet, and in ways that can be surprising or unexpected. This section of the document describes a variety of "sharp edges" in Mojo, and potentially how to work around them if needed. We expect all of these to be resolved in time, but in the meantime, they are documented here.

## No list or dict comprehensions

Mojo does not yet support Python list or dictionary comprehension expressions, like `[x for x in range(10)]`.

## No `lambda` syntax

Mojo does not yet support defining anonymous functions with the `lambda` keyword.

## Parametric aliases

Mojo aliases can refer to parametric values but cannot themselves have parameter lists. As of v0.6.0, you can create a parametric alias by aliasing an unbound or partially-bound type. For example, the new `Scalar` type is defined as:

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

This creates a parametric alias that you can use like this:

```
var i = Scalar[DType.int8]
```

Parametric aliases with an explicit parameter list aren't yet supported:

```
alias mul2[x: Int] = x * 2
# Error!
```

## Exception is actually called Error

In Python, programmers expect that exceptions all subclass the `Exception` builtin class. The only available type for Mojo "exceptions" is `Error`:

```
fn raise_an_error() raises:
    raise Error("I'm an error!")
```

The reason we call this type `Error` instead of `Exception` is because it's not really an exception. It's not an exception, because raising an error does not cause stack unwinding, but most importantly it does not have a stack trace. And without polymorphism, the `Error` type is the only kind of error that can be raised in Mojo right now.

## No Python-style generator functions

Mojo does not yet support Python-style generator functions (`yield` syntax). These are "synchronous co-routines" -- functions with multiple suspend points.

## No `async for` or `async with`

Although Mojo has support for async functions with `async fn` and `async def`, Mojo does not yet support the `async for` and `async with` statements.

## The `rebind` builtin

One of the consequences of Mojo not performing function instantiation in the parser like C++ is that Mojo cannot always figure out whether some parametric types are equal and complain about an invalid conversion. This typically occurs in static dispatch patterns, like:

```
fn take_simd8(x: SIMD[DType.float32, 8]): pass

fn generic_simd[nelts: Int](x: SIMD[DType.float32, nelts]):
    @parameter
    if nelts == 8:
        take_simd8(x)
```

The parser will complain,

```
error: invalid call to 'take_simd8': argument #0 cannot be converted from
'SIMD[f32, nelts]' to 'SIMD[f32, 8]'
    take_simd8(x)
    ~~~~~^~~~~
```

This is because the parser fully type-checks the function without instantiation, and the type of `x` is still `SIMD[f32, nelts]`, and not `SIMD[f32, 8]`, despite the static conditional. The remedy is to manually "rebind" the type of `x`,

using the `rebind` builtin, which inserts a compile-time assert that the input and result types resolve to the same type after function instantiation.

```
fn generic_simd[nelts: Int](x: SIMD[DType.float32, nelts]):  
  @parameter  
  if nelts == 8:  
    take_simd8(rebind[SIMD[DType.float32, 8]](x))
```

## Scoping and mutability of statement variables

Python programmers understand that local variables are implicitly declared and scoped at the function level. As the Mojo Manual explains, this is supported in Mojo for [implicitly-declared variables](#). However, there are some nuances to Python's implicit declaration rules that Mojo does not match 1-to-1.

For example, the scope of `for` loop iteration variables and caught exceptions in `except` statements is limited to the next indentation block, for both `def` and `fn` functions. Python programmers will expect the following program to print "2":

```
for i in range(3): pass  
print(i)
```

However, Mojo will complain that `print(i)` is a use of an unknown declaration. This is because whether `i` is defined at this line is dynamic in Python. For instance the following Python program will fail:

```
for i in range(0): pass  
print(i)
```

With `NameError: name 'i' is not defined`, because the definition of `i` is a dynamic characteristic of the function. Mojo's lifetime tracker is intentionally simple (so lifetimes are easy to use!), and cannot reason that `i` would be defined even when the loop bounds are constant.

## Name scoping of nested function declarations

In Python, nested function declarations produce dynamic values. They are essentially syntactic sugar for `bar = lambda ....`

```
def foo():  
  def bar(): # creates a function bound to the dynamic value 'bar'  
    pass  
  bar() # indirect call
```

In Mojo, nested function declarations are static, so calls to them are direct unless made otherwise.

```
fn foo():  
    fn bar(): # static function definition bound to 'bar'  
        pass  
    bar() # direct call  
    var f = bar # materialize 'bar' as a dynamic value  
    f() # indirect call
```

Currently, this means you cannot declare two nested functions with the same name. For instance, the following example does not work in Mojo:

```
def pick_func(cond) -> def() capturing:  
    if cond:  
        def bar(): return 42  
    else:  
        def bar(): return 3 # error: redeclaration of 'bar'  
    return bar
```

The functions in each conditional must be explicitly materialized as dynamic values.

```
def pick_func(cond) -> def() capturing:  
    var result: def() capturing # Mojo function type  
    if cond:  
        def bar0(): return 42  
        result = bar0  
    else:  
        def bar1(): return 3  
        result = bar1  
    return result
```

We hope to sort out these oddities with nested function naming as our model of closures in Mojo develops further.

## Limited polymorphism

Mojo has implemented static polymorphism through traits, as noted above. We plan to implement dynamic polymorphism through classes and MLIR reflection in the future.

Python programmers are used to implementing special dunder methods on their classes to interface with generic methods like `print()` and `len()`. For instance, one expects that implementing `__repr__()` or `__str__()` on a class will enable that class to be printed using `print()`.

```
class One:  
    def __init__(self): pass
```



```
def __repr__(self): return '1'
```

```
print(One()) # prints '1'
```

Mojo currently supports similar functionality through the [Formattable](#) trait, so that `print()` works on all `Formattable` types. Similar support exists for the [int\(\)](#) and [len\(\)](#) functions. We'll continue to add traits support to the standard library to enable common use cases like this.

## Parameter closure captures are unsafe references

You may have seen nested functions, or "closures", annotated with the `@parameter` decorator. This creates a "parameter closure", which behaves differently than a normal "stateful" closure. A parameter closure declares a compile-time value, similar to an `alias` declaration. That means parameter closures can be passed as parameters:

```
fn take_func[f: fn() capturing -> Int]():
    pass

fn call_it(a: Int):
    @parameter
    fn inner() -> Int:
        return a # capture 'a'

    take_func[inner]() # pass 'inner' as a parameter
```

Parameter closures can even be parametric and capturing:

```
fn take_func[f: fn[a: Int]() capturing -> Int]():
    pass

fn call_it(a: Int):
    @parameter
    fn inner[b: Int]() -> Int:
        return a + b # capture 'a'

    take_func[inner]() # pass 'inner' as a parameter
```

However, note that parameter closures are always capture by *unsafe* reference. Mojo's lifetime tracking is not yet sophisticated enough to form safe references to objects (see above section). This means that variable lifetimes need to be manually extended according to the lifetime of the parametric closure:

```
fn print_it[f: fn() capturing -> String]():
    print(f())

fn call_it():
```

```

var s: String = "hello world"
@parameter
fn inner() -> String:
    return s # 's' captured by reference, so a copy is made here
# lifetime tracker destroys 's' here

print_it[inner]() # crash! 's' has been destroyed

```

The lifetime of the variable can be manually extended by discarding it explicitly.

```

fn call_it():
    var s: String = "hello world"
    @parameter
    fn inner() -> String:
        return s

    print_it[inner]()
    _ = s^ # discard 's' explicitly

```

## The standard library has limited exceptions use

For historic and performance reasons, core standard library types typically do not use exceptions. For instance, `List` will not raise an out-of-bounds access (it will crash), and `Int` does not throw on divide by zero. In other words, most standard library types are considered "unsafe".

```

var l = List[Int](capacity=0)
print(l[1]) # could crash or print garbage values (undefined behavior)

print(1//0) # does not raise and could print anything (undefined behavior)

```

This is clearly unacceptable given the strong memory safety goals of Mojo. We will circle back to this when more language features and language-level optimizations are available.

## Nested functions cannot be recursive

Nested functions (any function that is not a top-level function) cannot be recursive in any way. Nested functions are considered "parameters", and although parameter values do not have to obey lexical order, their uses and definitions cannot form a cycle. Current limitations in Mojo mean that nested functions, which are considered parameter values, cannot be cyclic.

```

fn try_recursion():
    fn bar(x: Int): # error: circular reference :<

```

```
if x < 10:
    bar(x + 1)
```

## Only certain loaded MLIR dialects can be accessed

Although Mojo provides features to access the full power of MLIR, in reality only a certain number of loaded MLIR dialects can be accessed in the Playground at the moment.

The upstream dialects available in the Playground are the [index](#) dialect and the [LLVM](#) dialect.

## or expression is statically typed

Because Mojo has static typing, the `or` expression can't currently mimic the behavior of Python. In Python, the result type of the `or` expression is dynamic, based on the runtime values:

```
i: int = 0
s: str = "hello"
print(type(i or s)) # prints <class 'str'>
i = 5
print(type(i or s)) # prints <class 'int'>
```

In Mojo, given the expression `(a or b)`, the compiler needs to statically determine a result type that the types of `a` and `b` can both be **converted** to.

For example, currently an `Int` can be implicitly converted to a `String`, but a `String` can't be implicitly converted to an `Int`. So given an integer value `i` and a string value `s`, the value of `(i or s)` will *always* be a `String`.

## StringLiteral behaves differently than String

String literals behave differently than `String` values in Mojo code. For example:

```
fn main():
    var g: Int = 0
    var h: String = "hello"
    print(g or h) # prints `hello`
    print(g or "hello") # prints `True`
```


While the `IntLiteral` and `FloatLiteral` types convert or *materialize* at runtime into `Int` and `Float64` values, respectively, string literals continue to exist at runtime as `StringLiteral` values. This can result in surprising behavior because `StringLiteral` has a more restricted API than `String`.

In the example above, because the `or` expression is statically typed, and `Int` cannot be implicitly converted to a `StringLiteral`, the compiler chooses a result type that both `Int` and `StringLiteral` can be converted to—in this case, `Bool`.

We plan to address this issue in the future, but in the near term, you can avoid the inconsistency between `StringLiteral` and `String` problems by explicitly converting string literals to `String` values. For example:

```
var h: String = "hello"  
# or  
print(g or str("hello"))
```

Was this page helpful?  

 Edit this page