

Ownership and borrowing

A challenge you might face when using some programming languages is that you must manually allocate and deallocate memory. When multiple parts of the program need access to the same memory, it becomes difficult to keep track of who "owns" a value and determine when is the right time to deallocate it. If you make a mistake, it can result in a "use-after-free" error, a "double free" error, or a "leaked memory" error, any one of which can be catastrophic.

Mojo helps avoid these errors by ensuring there is only one variable that owns each value at a time, while still allowing you to share references with other functions. When the life span of the owner ends, Mojo [destroys the value](#). Programmers are still responsible for making sure any type that allocates resources (including memory) also deallocates those resources in its destructor. Mojo's ownership system ensures that destructors are called promptly.

On this page, we'll explain the rules that govern this ownership model, and how to specify different argument conventions that define how values are passed into functions.

Argument conventions

In all programming languages, code quality and performance is heavily dependent upon how functions treat argument values. That is, whether a value received by a function is a unique value or a reference, and whether it's mutable or immutable, has a series of consequences that define the readability, performance, and safety of the language.

In Mojo, we want to provide full [value semantics](#) by default, which provides consistent and predictable behavior. But as a systems programming language, we also need to offer full control over memory optimizations, which generally requires reference semantics. The trick is to introduce reference semantics in a way that ensures all code is memory safe by tracking the lifetime of every value and destroying each one at the right time (and only once). All of this is made possible in Mojo through the use of argument conventions that ensure every value has only one owner at a time.

An argument convention specifies whether an argument is mutable or immutable, and whether the function owns the value. Each convention is defined by a keyword at the beginning of an argument declaration:

- `borrowed`: The function receives an **immutable reference**. This means the function can read the original value (it is *not* a copy), but it cannot mutate (modify) it. `def` functions treat this differently, as described below.
- `inout`: The function receives a **mutable reference**. This means the function can read and mutate the original value (it is *not* a copy).

- `owned`: The function takes **ownership**. This means the function has exclusive ownership of the argument. Often, this also implies that the caller should transfer ownership to this function, but that's not always what happens and this might instead be a copy (as you'll learn below).
- `ref`: The function gets a reference with an associated lifetime. The reference can be either mutable or immutable. You can think of `ref` arguments as a generalization of the `borrowed` and `inout` conventions. `ref` arguments are an advanced topic, and they're described in more detail in [Lifetimes and references](#).

For example, this function has one argument that's a mutable reference and one that's immutable:

```
fn add(inout x: Int, borrowed y: Int):
  x += y

fn main():
  var a = 1
  var b = 2
  add(a, b)
  print(a) # Prints 3
```

You've probably already seen some function arguments that don't declare a convention. By default, all arguments are `borrowed`. But `def` and `fn` functions treat `borrowed` arguments somewhat differently:

- In an [fn function](#), the function always receives an immutable reference. If you want a mutable copy, you can assign it to a local variable:

```
var my_copy = borrowed_arg
```

- In a [def function](#), if the function mutates the value, the function receives a mutable copy of the argument. Otherwise, it receives an immutable reference. This allows you to treat arguments as mutable, but avoid the overhead of making extra copies when they're not needed.

The difference between `borrowed` and `owned` in a `def` function may be a little subtle:

- In a `def` function, a `borrowed` argument is received as an immutable reference, unless it's mutated in the body of the function. This eliminates unneeded copies, but maintains the Python expectation that arguments are mutable.
- The `borrowed` argument always gets an immutable reference or a local copy. You can't transfer a value into a `borrowed` argument.
- The `owned` argument always gets a uniquely owned value, which may have been copied or transferred from the callee. Using `owned` arguments without the transfer sigil (`^`) usually results in values being copied.

In the following sections, we'll explain each of these argument conventions in more detail.

Ownership summary

The fundamental rules that make Mojo's ownership model work are the following:

- Every value has only one owner at a time.
- When the lifetime of the owner ends, Mojo destroys the value.
- If there are outstanding references to a value, Mojo keeps the value alive.

In the future, the Mojo lifetime checker will enforce reference exclusivity, so that only one mutable reference to a value can exist at a time. **This is not currently enforced.**

Borrowed arguments (borrowed)

The `borrowed` convention is the default for all arguments.

In `fn` functions, a `borrowed` argument is received as an immutable reference.

In `def` functions, you can treat a `borrowed` argument as mutable or immutable. If you mutate the argument in the body of the function, you get a mutable copy of the original value. If you don't mutate the argument, you get an immutable reference, as in an `fn` function.

For example:

```
from collections import List

def print_list(list: List[Int]):
    print(list.__str__())

var list = List(1, 2, 3, 4)
print_list(list)
```

```
[1, 2, 3, 4]
```

Here the `list` argument to `print_list()` is borrowed and not mutated, so the `print_list()` function gets an immutable reference to the original `List`, and doesn't do any copying.

In general, passing an immutable reference is much more efficient when handling large or expensive-to-copy values, because the copy constructor and destructor are not invoked for a borrow.

To avoid expensive copies, types should only be implicitly copyable if the copy operation is inexpensive.

Compared to C++ and Rust

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

- The Mojo compiler implements a lifetime checker that ensures that values are not destroyed when there are outstanding references to those values.
- Small values like `Int`, `Float`, and `SIMD` are passed directly in machine registers instead of through an extra indirection (this is because they are declared with the `@register_passable` decorator). This is a [significant performance enhancement](#) when compared to languages like C++ and Rust, and moves this optimization from every call site to a declaration on the type definition.

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

Mutable arguments (`inout`)

If you'd like your function to receive a **mutable reference**, add the `inout` keyword in front of the argument name. You can think of `inout` like this: it means any changes to the value *inside* the function are visible *outside* the function.

For example, this `mutate()` function updates the original `list` value:

```
from collections import List

def mutate(inout l: List[Int]):
    l.append(5)

var list = List(1, 2, 3, 4)

mutate(list)
print_list(list)

[1, 2, 3, 4, 5]
```

That behaves like an optimized replacement for this:

```
from collections import List
```

```
def mutate_copy(l: List[Int]) -> List[Int]:
    l.append(5)
    return l

var list = List(1, 2, 3, 4)
list = mutate_copy(list)
print_list(list)
```

```
[1, 2, 3, 4, 5]
```

Although the code using `inout` isn't that much shorter, it's more memory efficient because it does not make a copy of the value.

However, remember that the values passed as `inout` must already be mutable. For example, if you try to take a borrowed value and pass it to another function as `inout`, you'll get a compiler error because Mojo can't form a mutable reference from an immutable reference.

Note

You cannot define default values for `inout` arguments.

Argument exclusivity

Mojo enforces *argument exclusivity* for mutable references. This means that if a function receives a mutable reference to a value (such as an `inout` argument), it can't receive any other references to the same value—mutable or immutable. That is, a mutable reference can't have any other references that *alias* it.

For example, consider the following code example:

```
fn append_twice(inout s: String, other: String):
    # Mojo knows 's' and 'other' cannot be the same string.
    s += other
    s += other

fn invalid_access():
    var my_string = str("o")

    # warning: passing `my_string` inout is invalid since it is also passed
    # borrowed.
    append_twice(my_string, my_string)
    print(my_string)
```

This code is confusing because the user might expect the output to be `ooo`, but since the first addition mutates both `s` and `other`, the actual output would be `oooo`. Enforcing exclusivity of mutable references not only prevents

coding errors, it also allows the Mojo compiler to optimize code in some cases.

One way to avoid this issue when you do need both a mutable and an immutable reference (or need to pass the same value to two arguments) is to make a copy:

```
fn valid_access():  
    var my_string = str("o")  
    var other_string = str(my_string)  
    append_twice(my_string, other_string)  
    print(my_string)
```

Only a warning

Aliasing a mutable reference produces a warning in v24.5. This will change to an error in a subsequent release.

Transfer arguments (owned and ^)

And finally, if you'd like your function to receive value **ownership**, add the `owned` keyword in front of the argument name.

This convention is often combined with use of the postfix `^` "transfer" sigil on the variable that is passed into the function, which ends the lifetime of that variable.

Technically, the `owned` keyword does not guarantee that the received value is *the original value*—it guarantees only that the function gets unique ownership of a value. This happens in one of three ways:

- The caller passes the argument with the `^` transfer sigil, which ends the lifetime of that variable (the variable becomes uninitialized) and ownership is transferred into the function without making a copy of any heap-allocated data.
- The caller **does not** use the `^` transfer sigil, in which case, the value is copied into the function argument and the original variable remains valid. (If the original value is not used again, the compiler may optimize away the copy and transfer the value).
- The caller passes in a newly-created "owned" value, such as a value returned from a function. In this case, no variable owns the value and it can be transferred directly to the callee. For example:

```
def take(owned s: String):  
    pass  
  
take(str("A brand-new String!"))
```

For example, the following code works by making a copy of the string, because—although `take_text()` uses the `owned` convention—the caller does not include the transfer sigil:

```
fn take_text(owned text: String):
    text += "!"
    print(text)
```

```
fn my_function():
    var message: String = "Hello"
    take_text(message)
    print(message)
```

```
my_function()
```

```
Hello!
Hello
```

However, if you add the `^` transfer sigil when calling `take_text()`, the compiler complains about `print(message)`, because at that point, the `message` variable is no longer initialized. That is, this version does not compile:

```
fn my_function():
    var message: String = "Hello"
    take_text(message^)
    print(message) # ERROR: The `message` variable is uninitialized
```

This is a critical feature of Mojo's lifetime checker, because it ensures that no two variables can have ownership of the same value. To fix the error, you must not use the `message` variable after you end its lifetime with the `^` transfer operator. So here is the corrected code:

```
fn my_function():
    var message: String = "Hello"
    take_text(message^)
```

```
my_function()
```

```
Hello!
```

Regardless of how it receives the value, when the function declares an argument as `owned`, it can be certain that it has unique mutable access to that value. Because the value is owned, the value is destroyed when the function exits—unless the function transfers the value elsewhere.

For example, in the following example, `add_to_list()` takes a string and appends it to the list. Ownership of the string is transferred to the list, so it's not destroyed when the function exits. On the other hand, `consume_string()`

doesn't transfer its `owned` value out, so the value is destroyed at the end of the function.

```
from collections import List

def add_to_list(owned name: String, inout list: List[String]):
    list.append(name^)
    # name is uninitialized, nothing to destroy

def consume_string(owned s: String):
    print(s)
    # s is destroyed here
```

Note

Value lifetimes are not fully implemented for top-level code in Mojo's REPL, so the transfer sigil currently works as intended only when used inside a function.

Transfer implementation details

In Mojo, you shouldn't conflate "ownership transfer" with a "move operation"—these are not strictly the same thing.

There are multiple ways that Mojo can transfer ownership of a value:

- If a type implements the [move constructor](#), `__moveinit__()`, Mojo may invoke this method *if* a value of that type is transferred into a function as an `owned` argument, *and* the original variable's lifetime ends at the same point (with or without use of the `^` transfer operator).
- If a type implements the [copy constructor](#), `__copyinit__()` and not `__moveinit__()`, Mojo may copy the value and destroy the old value.
- In some cases, Mojo can optimize away the move operation entirely, leaving the value in the same memory location but updating its ownership. In these cases, a value can be transferred without invoking either the `__copyinit__()` or `__moveinit__()` constructors.

In order for the `owned` convention to work *without* the transfer sigil, the value type must be copyable (via `__copyinit__()`).

Comparing `def` and `fn` argument conventions

As mentioned in the section about [functions](#), `def` and `fn` functions are interchangeable, as far as a caller is concerned, and they can both accomplish the same things. It's only the inside that differs, and Mojo's `def` function is essentially just sugaring for the `fn` function:


- A `def` argument without a type annotation defaults to [object](#) type (whereas as `fn` requires all types be explicitly declared).
- A `def` function can treat a `borrowed` argument as mutable (in which case it receives a mutable copy). An `fn` function must make this copy explicitly.

For example, these two functions have the exact same behavior.

```
def def_example(a: Int, inout b: Int, owned c):  
    pass  
  
fn fn_example(a_in: Int, inout b: Int, owned c: object):  
    var a = a_in  
    pass
```

This shadow copy typically adds no overhead, because small types like `object` are cheap to copy. However, copying large types that allocate heap storage can be expensive. (For example, copying `List` or `Dict` types, or copying large numbers of strings.)

Was this page helpful?  

 Edit this page