Value semantics

Mojo doesn't enforce value semantics or reference semantics. It supports them both and allows each type to define how it is created, copied, and moved (if at all). So, if you're building your own type, you can implement it to support value semantics, reference semantics, or a bit of both. That said, Mojo is designed with argument behaviors that default to value semantics, and it provides tight controls for reference semantics that avoid memory errors.

The controls over reference semantics are provided by the value ownership model, but before we get into the syntax and rules for that, it's important that you understand the principles of value semantics. Generally, it means that each variable has unique access to a value, and any code outside the scope of that variable cannot modify its value.

Intro to value semantics

In the most basic situation, sharing a value-semantic type means that you create a copy of the value. This is also known as "pass by value." For example, consider this code:

```
x = 1
y = x
y += 1

print(x)
print(y)
```

```
1
2
```

We assigned the value of `x` to `y`, which creates the value for `y` by making a copy of `x`. When we increment `y`, the value of `x` doesn't change. Each variable has exclusive ownership of a value.

Whereas, if a type instead uses reference semantics, then `y` would point to the same value as `x`, and incrementing either one would affect the value for both. Neither `x` nor `y` would "own" the value, and any variable would be allowed to reference it and mutate it.

Numeric values in Mojo are value semantic because they're trivial types, which are cheap to copy.

Here's another example with a function:

```
def add_one(y: Int):
    y += 1
    print(y)

x = 1
add_one(x)
print(x)
```

```
2
1
```

Again, the `y` value is a copy and the function cannot modify the original `x` value.

If you're familiar with Python, this is probably familiar so far, because the code above behaves the same in Python. However, Python is not value semantic.

It gets complicated, but let's consider a situation in which you call a Python function and pass an object with a pointer to a heap-allocated value. Python actually gives that function a reference to your object, which allows the function to mutate the heap-allocated value. This can cause nasty bugs if you're not careful, because the function might incorrectly assume it has unique ownership of that object.

In Mojo, the default behavior for all function arguments is to use value semantics. If the function wants to modify the value of an incoming argument, then it must explicitly declare so, which avoids accidental mutations of the original value.

All Mojo types passed to a `def` function can be treated as mutable, which maintains the expected mutability behavior from Python. But by default, it is mutating a uniquely-owned value, not the original value.

For example, when you pass an instance of a `SIMD` vector to a `def` function it creates a unique copy of all values. Thus, if we modify the argument in the function, the original value is unchanged:

```
def update_simd(t: SIMD[DType.int32, 4]):
    t[0] = 9
    print(t)

v = SIMD[DType.int32, 4](1, 2, 3, 4)
update_simd(v)
print(v)
```

```
[9, 2, 3, 4]
[1, 2, 3, 4]
```

If this were Python code, the function would modify the original object, because Python shares a reference to the original object.

However, not all types are inexpensive to copy. Copying a `String` or `List` requires allocating heap memory, so we want to avoid copying one by accident. When designing a type like this, ideally you want to prevent *implicit* copies, and only make a copy when it's explicitly requested.

Value semantics in `def` vs `fn`

The arguments above are mutable because a `def` function has special treatment for the default `borrowed` argument convention.

Whereas, `fn` functions always receive `borrowed` arguments as immutable references. This is a memory optimization to avoid making unnecessary copies.

For example, let's create another function with the `fn` declaration. In this case, the `y` argument is immutable by default, so if the function wants to modify the value in the local scope, it needs to make a local copy:

```
fn add_two(y: Int):
    # y += 2 # This will cause a compiler error because `y` is immutable
    # We can instead make an explicit copy:
    var z = y
    z += 2
    print(z)

x = 1
add_two(x)
print(x)
```

```
3
1
```

This is all consistent with value semantics because each variable maintains unique ownership of its value.

The way the `fn` function receives the `y` value is a "look but don't touch" approach to value semantics. This is also a more memory-efficient approach when dealing with memory-intensive arguments, because Mojo doesn't make any copies unless we explicitly make the copies ourselves.

Thus, the default behavior for `def` and `fn` arguments is fully value semantic: arguments are either copies or immutable references, and any living variable from the callee is not affected by the function.

But we must also allow reference semantics (mutable references) because it's how we build performant and memory-efficient programs (making copies of everything gets really expensive). The challenge is to introduce reference semantics in a way that does not disturb the predictability and safety of value semantics.

The way we do that in Mojo is, instead of enforcing that every variable have "exclusive access" to a value, we ensure that every value has an "exclusive owner," and destroy each value when the lifetime of its owner ends.

On the next page about value ownership, you'll learn how to modify the default argument conventions, and safely use reference semantics so every value has only one owner at a time.

Python-style reference semantics

> **Note**
>
> If you will always use strict type declarations, you can skip this section because it only applies to Mojo code using `def` functions without type declarations (or values declared as `object`).

As we said at the top of this page, Mojo doesn't enforce value semantics or reference semantics. It's up to each type author to decide how an instance of their type should be created, copied, and moved (see Value lifecycle). Thus, in order to provide compatibility with Python, Mojo's `object` type is designed to support Python's style of argument passing for functions, which is different from the other types in Mojo.

Python's argument-passing convention is called "pass by object reference." This means when you pass a variable to a Python function, you actually pass a reference to the object, as a value (so it's not strictly reference semantics).

Passing the object reference "as a value" means that the argument name is just a container that acts like an alias to the original object. If you reassign the argument inside the function, it does not affect the caller's original value. However, if you modify the object itself (such as call `append()` on a list), the change is visible to the original object outside the function.

For example, here's a Python function that receives a list and modifies it:

```
%%python
def modify_list(l):
    l.append(3)
    print("func:", l)

ar = [1, 2]
modify_list(ar)
print("orig:", ar)
```

```
func: [1, 2, 3]
orig: [1, 2, 3]
```

In this example, it looks like the list is "passed by reference" because `l` modifies the original value.

However, if the Python function instead *assigns* a value to `l`, it does not affect the original value:

```
%%python
def change_list(l):
    l = [3, 4]
    print("func:", l)

ar = [1, 2]
change_list(ar)
print("orig:", ar)
```

```
func: [3, 4]
orig: [1, 2]
```

This demonstrates how a Python argument holds the object reference *as a value*: the function can mutate the original value, but it can also assign a new object to the argument name.

Pass by object reference in Mojo

Although we haven't finished implementing the `object` type to represent any Mojo type, our intention is to do so, and enable "pass by object reference" as described above for all dynamic types in a `def` function.

That means you can have dynamic typing and "pass by object reference" behavior by simply writing your Mojo code like Python:

1. Use `def` function declarations.
2. Don't declare argument types.

TODO

Mojo is not a complete superset of Python yet, and there is a lot to do in this department before Mojo supports all of Python's types and behaviors. As such, this is a topic that also still needs a lot of documentation.