Functions

As mentioned in Language basics, Mojo supports two types of functions: `def` and `fn` functions. You can use either declaration with any function, including the `main()` function, but they have different default behaviors, as described on this page.

We believe both `def` and `fn` have good use cases and don't consider either to be better than the other. Deciding which to use is a matter of personal taste as to which style best fits a given task.

We believe Mojo's flexibility in this regard is a superpower that allows you to write code in the manner that's best for your project.

Note

Functions declared inside a `struct` are called "methods," but they have all the same qualities as "functions" described here.

`fn` functions

The `fn` function has somewhat stricter rules than the `def` function.

Here's an example of an `fn` function:

```
fn greet(name: String) -> String:
    var greeting = "Hello, " + name + "!"
    return greeting
```

As far as a function caller is concerned, `def` and `fn` functions are interchangeable. That is, there's nothing a `def` can do that an `fn` can't (and vice versa). The difference is that, compared to a `def` function, an `fn` function is more strict on the inside.

Here's everything to know about `fn`:

- Arguments must specify a type (except for the `self` argument in struct methods).
- Return values must specify a type, unless the function doesn't return a value.

  If you don't specify a return type, it defaults to `None` (meaning no return value).

- By default, arguments are received as an immutable reference (values are read-only, using the `borrowed` argument convention).

  This prevents accidental mutations, and permits the use of non-copyable types as arguments.

  If you want a local copy, you can simply assign the value to a local variable. Or, you can get a mutable reference to the value by declaring the `inout` argument convention).
- If the function raises an exception, it must be explicitly declared with the `raises` keyword. (A `def` function does not need to declare exceptions.)

By enforcing these type checks, using the `fn` function helps avoid a variety of runtime errors.

`def` functions

Compared to an `fn` function, a `def` function has fewer restrictions. The `def` function works more like a Python `def` function. For example, this function works the same in Python and Mojo:

```
def greet(name):
    greeting = "Hello, " + name + "!"
    return greeting
```

In a Mojo `def` function, you have the option to specify the argument type and the return type. You can also declare variables with `var`, with or without explicit typing. So you can write a `def` function that looks almost exactly like the `fn` function shown earlier:

```
def greet(name: String) -> String:
    var greeting = "Hello, " + name + "!"
    return greeting
```

This way, the compiler ensures that `name` is a string, and the return type is a string.

Here's everything to know about `def`:

- Arguments don't require a declared type.

  Undeclared arguments are actually passed as an `object`, which allows the function to receive any type (Mojo infers the type at runtime).
- Return types don't need to be declared, and also default to `object`. (If a `def` function doesn't declare a return type of `None`, it's considered to return an `object` by default.)
- Arguments are mutable. Arguments default to using the `borrowed` argument convention) like an `fn` function, with a special addition: if the function mutates the argument, it makes a mutable copy.

  If an argument is an `object` type, it's received as a reference, following object reference semantics.

If an argument is any other declared type, it's received as a value.

The `object` type

If you don't declare the type for an argument or return value in a `def`, it becomes an `object`, which is unlike any other type in the standard library.

The `object` type allows for dynamic typing because it can actually represent any type in the Mojo standard library, and the actual type is inferred at runtime. (Actually, there's still more to do before it can represent all Mojo types.) This is great for compatibility with Python and all of the flexibility that it provides with dynamic types. However, this lack of type enforcement can lead to runtime errors when a function receives or returns an unexpected type.

For compatibility with Python, `object` values are passed using object reference semantics. As such, the `object` type is not compatible with the argument conventions that enforce value semantics. So, be careful if using `object` values alongside other strongly-typed values—their behavior might be inconsistent because `object` is the only type in the standard library that does not conform to full value semantics.

> **TODO**
>
> The `object` type is still a work in progress. It doesn't support all of the possible underlying types, for example.

Function arguments

As noted in the previous sections, there are a few differences between how `def` and `fn` functions treat arguments. But most of the time they are the same.

As noted, there are some differences in *argument conventions*. Argument conventions are discussed in much more detail in the page on Ownership.

The other difference is that `def` functions don't need to specify an argument's type. If no type is specified, the argument is passed as an `object`.

The remaining rules for arguments described in this section apply to both `def` and `fn` functions.

Optional arguments

An optional argument is one that includes a default value, such as the `exp` argument here:

```
fn my_pow(base: Int, exp: Int = 2) -> Int:
    return base ** exp
```

```
fn use_defaults():
    # Uses the default value for `exp`
    var z = my_pow(3)
    print(z)
```

However, you cannot define a default value for an argument that's declared as `inout`.

Any optional arguments must appear after any required arguments. Keyword-only arguments, discussed later, can also be either required or optional.

Keyword arguments

You can also use keyword arguments when calling a function. Keyword arguments are specified using the format

```
argument_name = argument_value
```

. You can pass keyword arguments in any order:

```
fn my_pow(base: Int, exp: Int = 2) -> Int:
    return base ** exp

fn use_keywords():
    # Uses keyword argument names (with order reversed)
    var z = my_pow(exp=3, base=2)
    print(z)
```

Variadic arguments

Variadic arguments let a function accept a variable number of arguments. To define a function that takes a variadic argument, use the variadic argument syntax

```
*argument_name
```

:

```
fn sum(*values: Int) -> Int:
  var sum: Int = 0
  for value in values:
    sum = sum + value
  return sum
```

The variadic argument `values` here is a placeholder that accepts any number of passed positional arguments.

You can define zero or more arguments before the variadic argument. When calling the function, any remaining positional arguments are assigned to the variadic argument, so any arguments declared **after** the variadic argument can only be specified by keyword (see Positional-only and keyword-only arguments).

Variadic arguments can be divided into two categories:

- Homogeneous variadic arguments, where all of the passed arguments are the same type—all `Int`, or all `String`, for example.
- Heterogeneous variadic arguments, which can accept a set of different argument types.

The following sections describe how to work with homogeneous and heterogeneous variadic arguments.

> **Variadic parameters**
>
> Mojo parameters are distinct from arguments (parameters are used for compile-time metaprogramming). Variadic parameters are supported, but with some limitations—for details see variadic parameters.

Homogeneous variadic arguments

When defining a homogeneous variadic argument, use

```
*argument_name: argument_type
```

:

```
def greet(*names: String):
    ...
```

Inside the function body, the variadic argument is available as an iterable list for ease of use. Currently there are some differences in handling the list depending on whether the arguments are register-passable types (such as `Int`) or memory-only types (such as `String`). TODO: We hope to remove these differences in the future.

Register-passable types, such as `Int`, are available as a `VariadicList` type. As shown in the previous example, you can iterate over the values using a `for..in` loop.

```
fn sum(*values: Int) -> Int:
  var sum: Int = 0
  for value in values:
    sum = sum+value
  return sum
```

Memory-only types, such as `String`, are available as a `VariadicListMem`. Iterating over this list directly with a `for..in` loop currently produces a `Reference` for each value instead of the value itself. You must add an empty subscript operator `[]` to dereference the reference and retrieve the value:

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

Alternately, subscripting into a `VariadicListMem` returns the argument value, and doesn't require any dereferencing:

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

Heterogeneous variadic arguments

Implementing heterogeneous variadic arguments is somewhat more complicated than homogeneous variadic arguments. Writing generic code to handle multiple argument types requires traits and parameters. So the syntax may look a little unfamiliar if you haven't worked with those features. The signature for a function with a heterogeneous variadic argument looks like this:

```
def count_many_things[*ArgTypes: Intable](*args: *ArgTypes):
    ...
```

The parameter list, `[*ArgTypes: Intable]` specifies that the function takes an `ArgTypes` parameter, which is a list of types, all of which conform to the `Intable` trait. The argument list, `(*args: *ArgTypes)` has the familiar `*args` for the variadic argument, but instead of a single type, its type is defined as *list* of types, `*ArgTypes`.

This means that each argument in `args` has a corresponding type in `ArgTypes`, so

```
args[n]
```

is of type

```
ArgTypes[n]
```

.

Inside the function, `args` is available as a `VariadicPack`. The easiest way to work with the arguments is to use the `each()` method to iterate through the `VariadicPack`:

```
fn count_many_things[*ArgTypes: Intable](*args: *ArgTypes) -> Int:
    var total = 0

    @parameter
    fn add[Type: Intable](value: Type):
        total += int(value)

    args.each[add]()
    return total

print(count_many_things(5, 11.7, 12))
```

```
28
```

In the example above, the `add()` function is called for each argument in turn, with the appropriate `value` and `Type` values. For instance, `add()` is first called with `value=5` and `Type=Int`, then with `value=11.7` and `Type=Float64`.

Also, note that when calling `count_many_things()`, you don't actually pass in a list of argument types. You only need to pass in the arguments, and Mojo generates the `ArgTypes` list itself.

As a small optimization, if your function is likely to be called with a single argument frequently, you can define your function with a single argument followed by a variadic argument. This lets the simple case bypass populating and iterating through the `VariadicPack`.

For example, given a `print_string()` function that prints a single string, you could re-implement the variadic `print()` function with code like this:

```
fn print_string(s: String):
    print(s, end="")

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

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

```
Bob
```

If you call `print_many()` with a single argument, it calls `print_string()` directly. The `VariadicPack` is empty, so `each()` returns immediately without calling the `print_elt()` function.

Variadic keyword arguments

Mojo functions also support variadic keyword arguments (`**kwargs`). Variadic keyword arguments allow the user to pass an arbitrary number of keyword arguments. To define a function that takes a variadic keyword argument, use the variadic keyword argument syntax

```
**kw_argument_name
```

:

```
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)
```

In this example, the argument name `kwargs` is a placeholder that accepts any number of keyword arguments. Inside the body of the function, you can access the arguments as a dictionary of keywords and argument values (specifically, an instance of [`OwnedKwargsDict`](#)).

There are currently a few limitations:

- Variadic keyword arguments are always implicitly treated as if they were declared with the `owned` [argument convention](#), and can't be declared otherwise:

  ```
  # Not supported yet.
  fn borrowed_var_kwargs(borrowed **kwargs: Int): ...
  ```

- All the variadic keyword arguments must have the same type, and this determines the type of the argument dictionary. For example, if the argument is `**kwargs: Float64` then the argument dictionary will be a `OwnedKwargsDict[Float64]`.
- The argument type must conform to the [`CollectionElement`](#) trait. That is, the type must be both [`Movable`](#) and [`Copyable`](#).
- Dictionary unpacking is not supported yet:

  ```
  fn takes_dict(d: Dict[String, Int]):
    print_nicely(**d)  # Not supported yet.
  ```

- Variadic keyword *parameters* are not supported yet:

```
# Not supported yet.
fn var_kwparams[**kwparams: Int](): ...
```

Positional-only and keyword-only arguments

When defining a function, you can restrict some arguments so that they can only be passed as positional arguments, or they can only be passed as keyword arguments.

To define positional-only arguments, add a slash character (`/`) to the argument list. Any arguments before the `/` are positional-only: they can't be passed as keyword arguments. For example:

```
fn min(a: Int, b: Int, /) -> Int:
    return a if a < b else b
```

This `min()` function can be called with `min(1, 2)` but can't be called using keywords, like `min(a=1, b=2)`.

There are several reasons you might want to write a function with positional-only arguments:

- The argument names aren't meaningful for the the caller.
- You want the freedom to change the argument names later on without breaking backward compatibility.

For example, in the `min()` function, the argument names don't add any real information, and there's no reason to specify arguments by keyword.

For more information on positional-only arguments, see PEP 570 – Python Positional-Only Parameters.

Keyword-only arguments are the inverse of positional-only arguments: they can only be specified by keyword. If a function accepts variadic arguments, any arguments defined *after* the variadic arguments are treated as keyword-only. For example:

```
fn sort(*values: Float64, ascending: Bool = True): ...
```

In this example, the user can pass any number of `Float64` values, optionally followed by the keyword `ascending` argument:

```
var a = sort(1.1, 6.5, 4.3, ascending=False)
```

If the function doesn't accept variadic arguments, you can add a single star (`*`) to the argument list to separate the keyword-only arguments:

```
fn kw_only_args(a1: Int, a2: Int, *, double: Bool) -> Int:
    var product = a1 * a2
    if double:
        return product * 2
    else:
        return product
```

Keyword-only arguments often have default values, but this is not required. If a keyword-only argument doesn't have a default value, it is a *required keyword-only argument*. It must be specified, and it must be specified by keyword.

Any required keyword-only arguments must appear in the signature before any optional keyword-only arguments. That is, arguments appear in the following sequence a function signature:

- Required positional arguments.
- Optional positional arguments.
- Variadic arguments.
- Required keyword-only arguments.
- Optional keyword-only arguments.
- Variadic keyword arguments.

For more information on keyword-only arguments, see PEP 3102 – Keyword-Only Arguments.

Overloaded functions

If a `def` function does not specify argument types, then it can accept any data type and decide how to handle each type internally. This is nice when you want expressive APIs that just work by accepting arbitrary inputs, so there's usually no need to write function overloads for a `def` function.

On the other hand, all `fn` functions must specify argument types, so if you want a function to work with different data types, you need to implement separate versions of the function that each specify different argument types. This is called "overloading" a function.

For example, here's an overloaded `add()` function that can accept either `Int` or `String` types:

```
fn add(x: Int, y: Int) -> Int:
    return x + y

fn add(x: String, y: String) -> String:
    return x + y
```

If you pass anything other than `Int` or `String` to the `add()` function, you'll get a compiler error. That is, unless `Int` or `String` can implicitly cast the type into their own type. For example, `String` includes an overloaded version of its constructor (`__init__()`) that accepts a `StringLiteral` value. Thus, you can also pass a `StringLiteral` to a function that expects a `String`.

When resolving an overloaded function call, the Mojo compiler tries each candidate function and uses the one that works (if only one version works), or it picks the closest match (if it can determine a close match), or it reports that the call is ambiguous (if it can't figure out which one to pick).

If the compiler can't figure out which function to use, you can resolve the ambiguity by explicitly casting your value to a supported argument type. For example, in the following code, we want to call the overloaded `foo()` function, but both implementations accept an argument that supports implicit conversion from `StringLiteral`. So, the call to `foo(string)` is ambiguous and creates a compiler error. We can fix it by casting the value to the type we really want:

```
@value
struct MyString:
    fn __init__(inout self, string: StringLiteral):
        pass

fn foo(name: String):
    print("String")

fn foo(name: MyString):
    print("MyString")

fn call_foo():
    alias string: StringLiteral = "Hello"
    # foo(string) # This call is ambiguous because two `foo` functions match it
    foo(MyString(string))
```

When resolving an overloaded function, Mojo does not consider the return type or other contextual information at the call site—only the argument types affect which function is selected.

Overloading also works with combinations of both `fn` and `def` functions. For example, you could define multiple `fn` function overloads and then one or more `def` versions that don't specify all argument types, as a fallback.

Note

Although we haven't discussed parameters yet (they're different from function arguments, and used for compile-time metaprogramming), you can also overload functions based on parameter types.

Was this page helpful?