Introduction to Mojo

At this point, you should have already set up the Mojo SDK and run "Hello world". Now let's talk about how to write Mojo code.

You probably already know that Mojo is designed as a superset of Python. So if you know Python, then a lot of Mojo code will look familiar. However, Mojo is—first and foremost—designed for high-performance systems programming, with features like strong type checking, memory safety, next-generation compiler technologies, and more. As such, Mojo also has a lot in common with languages like C++ and Rust.

Yet, we've designed Mojo to be flexible, so you can incrementally adopt systems-programming features like strong type checking as you see fit—Mojo does not *require* strong type checking.

On this page, we'll introduce the essential Mojo syntax, so you can start coding quickly and understand other Mojo code you encounter. Subsequent sections in the Mojo Manual dive deeper into these topics, and links are provided below as appropriate.

Let's get started! 🔥

Note

Mojo is a young language and there are many features still missing. As such, Mojo is currently **not** meant for beginners. Even this basics section assumes some programming experience. However, throughout the Mojo Manual, we try not to assume experience with any particular language.

Functions

Mojo functions can be declared with either `fn` or `def`.

The `fn` declaration enforces type-checking and memory-safe behaviors (Rust style), while `def` allows no type declarations and dynamic behaviors (Python style).

For example, this `def` function doesn't require declaration of argument types or the return type:

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

While the same thing as an `fn` function requires that you specify the argument type and the return type like this:

```
fn greet2(name: String) -> String:
    return "Hello, " + name + "!"
```

Both functions have the same result, but the `fn` function provides compile-time checks to ensure the function receives and returns the correct types. Whereas, the `def` function might fail at runtime if it receives the wrong type.

Currently, Mojo doesn't support top-level code in a `.mojo` (or `.🔥`) file, so every program must include a function named `main()` as the entry point. You can declare it with either `def` or `fn`:

```
def main():
   print("Hello, world!")
```

Note

You don't need a `main()` function when coding in the REPL or in a Jupyter notebook.

For more details, see the page about functions.

Value ownership and argument mutability

If you're wondering whether function arguments are passed by value or passed by reference, the short answer is: `def` functions receive arguments "by value" and `fn` functions receive arguments "by immutable reference."

The longer short answer is that Mojo allows you to specify for each argument whether it should be passed by value (as `owned`), or whether it should be passed by reference (as `borrowed` for an immutable reference, or as `inout` for a mutable reference).

This feature is entwined with Mojo's value ownership model, which protects you from memory errors by ensuring that only one variable "owns" a value at any given time (but allowing other variables to receive a reference to it). Ownership then ensures that the value is destroyed when the lifetime of the owner ends (and there are no outstanding references).

But that's still a short answer, because going much further is a slippery slope into complexity that is out of scope for this section. For the complete answer, see the section about value ownership.

Variables

You can declare variables with the `var` keyword. Or, if your code is in a `def` function, you can omit the `var` (in an `fn` function, you must include the `var` keyword).

For example:

```
def do_math(x):
    var y = x + x
    y = y * y
    print(y)
```

Optionally, you can also declare a variable type like this:

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

Even in an `fn` function, declaring the variable type is optional (only the argument and return types must be declared in `fn` functions).

For more details, see the page about variables.

Structs

You can build high-level abstractions for types (or "objects") as a `struct`.

A `struct` in Mojo is similar to a `class` in Python: they both support methods, fields, operator overloading, decorators for metaprogramming, and so on. However, Mojo structs are completely static—they are bound at compile-time, so they do not allow dynamic dispatch or any runtime changes to the structure. (Mojo will also support Python-style classes in the future.)

For example, here's a basic struct:

```
struct MyPair:
    var first: Int
    var second: Int

    fn __init__(inout self, first: Int, second: Int):
        self.first = first
        self.second = second

    fn dump(self):
        print(self.first, self.second)
```

And here's how you can use it:

```
fn use_mypair():
    var mine = MyPair(2, 4)
    mine.dump()
```

For more details, see the page about structs.

Traits

A trait is like a template of characteristics for a struct. If you want to create a struct with the characteristics defined in a trait, you must implement each characteristic (such as each method). Each characteristic in a trait is a "requirement" for the struct, and when your struct implements each requirement, it's said to "conform" to the trait.

Currently, the only characteristics that traits can define are method signatures. Also, traits currently cannot implement default behaviors for methods.

Using traits allows you to write generic functions that can accept any type that conforms to a trait, rather than accept only specific types.

For example, here's how you can create a trait (notice the function is not implemented):

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

And here's how to create a struct that conforms to the trait:

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

Then, here's a function that uses the trait as an argument type (instead of the struct type):

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

fn use_trait_function():
    var thing = SomeStruct()
    fun_with_traits(thing)
```

Note

You're probably wondering about the square brackets on `fun_with_traits()`. These aren't function *arguments* (which go in parentheses); these are function *parameters*, which we'll explain next.

Without traits, the `x` argument in `fun_with_traits()` would have to declare a specific type that implements `required_method()`, such as `SomeStruct` (but then the function would accept only that type). With traits, the function can accept any type for `x` as long as it conforms to (it "implements") `SomeTrait`. Thus, `fun_with_traits()` is known as a "generic function" because it accepts a *generalized* type instead of a specific type.

For more details, see the page about traits.

Parameterization

In Mojo, a parameter is a compile-time variable that becomes a runtime constant, and it's declared in square brackets on a function or struct. Parameters allow for compile-time metaprogramming, which means you can generate or modify code at compile time.

Many other languages use "parameter" and "argument" interchangeably, so be aware that when we say things like "parameter" and "parametric function," we're talking about these compile-time parameters. Whereas, a function "argument" is a runtime value that's declared in parentheses.

Parameterization is a complex topic that's covered in much more detail in the Metaprogramming section, but we want to break the ice just a little bit here. To get you started, let's look at a parametric function:

```
fn repeat[count: Int](msg: String):
    for i in range(count):
        print(msg)
```

This function has one parameter of type `Int` and one argument of type `String`. To call the function, you need to specify both the parameter and the argument:

```
fn call_repeat():
    repeat[3]("Hello")
    # Prints "Hello" 3 times
```

By specifying `count` as a parameter, the Mojo compiler is able to optimize the function because this value is guaranteed to not change at runtime. The compiler effectively generates a unique version of the `repeat()` function that repeats the message only 3 times. This makes the code more performant because there's less to compute at runtime.

Similarly, you can define a struct with parameters, which effectively allows you to define variants of that type at compile-time, depending on the parameter values.

For more detail on parameters, see the section on Metaprogramming.

Blocks and statements

Code blocks such as functions, conditions, and loops are defined with a colon followed by indented lines. For example:

```
def loop():
    for x in range(5):
        if x % 2 == 0:
            print(x)
```

You can use any number of spaces or tabs for your indentation (we prefer 4 spaces).

All code statements in Mojo end with a newline. However, statements can span multiple lines if you indent the following lines. For example, this long string spans two lines:

```
def print_line():
    long_text = "This is a long line of text that is a lot easier to read if"
                " it is broken up across two lines instead of one long line."
    print(long_text)
```

And you can chain function calls across lines:

```
def print_hello():
    text = String(",")
          .join("Hello", " world!")
    print(text)
```

Code comments

You can create a one-line comment using the hash # symbol:

```
# This is a comment. The Mojo compiler ignores this line.
```

Comments may also follow some code:

```
var message = "Hello, World!" # This is also a valid comment
```

You can instead write longer comments across many lines using triple quotes:

```
"""
This is also a comment, but it's easier to write across
many lines, because each line doesn't need the # symbol.
"""
```

Triple quotes is the preferred method of writing API documentation. For example:

```
fn print(x: String):
    """Prints a string.

    Args:
        x: The string to print.
    """
    ...
```

Documenting your code with these kinds of comments (known as "docstrings") is a topic we've yet to fully specify, but you can generate an API reference from docstrings using the `mojo doc` command.

Python integration

Mojo is not yet a full superset of Python, but we've built a mechanism to import Python modules as-is, so you can leverage existing Python code right away.

For example, here's how you can import and use NumPy (you must have Python `numpy` installed):

```
from python import Python

fn use_numpy() raises:
    var np = Python.import_module("numpy")
    var ar = np.arange(15).reshape(3, 5)
    print(ar)
    print(ar.shape)
```

Note

Note: You must have the Python module (such as `numpy`) installed already.

For more details, see the page about Python integration.

Next steps

Hopefully this page has given you enough information to start experimenting with Mojo, but this is only touching the surface of what's available in Mojo.

If you're in the mood to read more, continue through each page of this Mojo Manual using the buttons at the bottom of each page—the next page from here is Functions.

Otherwise, here are some other resources to check out:

- If you want to experiment with some code, clone the Mojo repo to try our code examples:

  ```
  $ git clone https://github.com/modularml/mojo.git
  ```

  In addition to several `.mojo` examples, the repo includes Jupyter notebooks that teach advanced Mojo features.
- To see all the available Mojo APIs, check out the Mojo standard library reference.