HelloMojo

August 9, 2023

1 Hello, Mojo

We're excited to introduce you to Mojo with this interactive notebook!

Mojo is designed as a superset of Python, so a lot of language features and concepts you might know in Python translate directly to Mojo. For example, a "Hello World" program in Mojo looks exactly like Python:

[1]:
```
print("Hello Mojo!")
```

```
Hello Mojo!
```

You can also import existing Python packages and use them as if you're programming in Python, but we'll get to that later.

However, it's important to know that Mojo is an entirely new language on its own, not just a new implementation of Python with extra sugar. As you learn more about Mojo, you'll see that it has more in common with languages like Rust and C++, except it uses Python syntax and fully supports imported Python packages.

So let's get started! This notebook introduces the basics of the Mojo language, and requires only a little programming experience.

If you want much more detail about the language, check out the Mojo programming manual.

Mojo is a work in progress: Please send us bug reports, suggestions, and questions through our Mojo community channels. And see what's new in the Mojo changelog.

Note: Mojo Playground is designed only for testing the Mojo language. The cloud environment is not always stable and performance varies, so it is not an appropriate environment for performance benchmarking. However, we believe it can still demonstrate the magnitude of performance gains provided by Mojo, as shown in the Matmul.ipynb notebook. For more information about the compute power in the Mojo Playground, see the Mojo FAQ.

1.1 Language basics

First and foremost, Mojo is a compiled language and a lot of its performance and memory-safety features are derived from that fact. Mojo code can be ahead-of-time (AOT) or just-in-time (JIT) compiled. Mojo also supports REPL environments such as the one that runs the code in this Jupyter notebook (and command-line REPL is coming soon).

Like other compiled languages, Mojo requires a `main()` function as the entry point to a program. For example:

[2]:
```
fn main():
    var x: Int = 1
    x += 1
    print(x)
```

If you know Python, you might have expected `def main()` instead of `fn main()`. Both actually work in Mojo, but using `fn` behaves a bit differently, as we'll discuss below.

Of course, the `main()` function isn't required in a REPL environment, as shown above when we printed "hello world" without a `main()` function. But the `main()` function is required when you want to write your own `.mojo` programs.

Note: Local development with `.mojo` files is coming soon—currently, using the Mojo Playground is the only way you can run Mojo code.

Now let's discuss the code shown in this `main()` function.

1.1.1 Syntax and semantics

This is simple: Mojo uses all of Python's syntax and semantics. (If you're not familiar with Python syntax, there are a ton of great resources online that can teach you.)

For example, like Python, Mojo uses line breaks and indentation to define code blocks (not curly braces), and Mojo supports all of Python's control-flow syntax such as `if` conditions and `for` loops.

However, Mojo is still a work in progress, so there are some things from Python that aren't implemented in Mojo yet (see the Mojo roadmap). All the missing Python features will arrive in time, but Mojo already includes many features and capabilities beyond what's available in Python.

As such, the following sections will focus on some of the language features that are unique to Mojo (compared to Python).

1.1.2 Functions

Mojo functions can be declared with either `fn` (shown above) or `def` (as in Python). The `fn` declaration enforces strongly-typed and memory-safe behaviors, while `def` provides Python-style dynamic behaviors.

Both `fn` and `def` functions have their value, and it's important that you learn them both. However, for the purposes of this introduction, we're going to focus on `fn` functions only. For much more detail about both, see the programming manual.

In the following sections, you'll learn how `fn` functions enforce strongly-typed and memory-safe behaviors in your code.

1.1.3 Variables

You can declare variables, such as `x` in the above `main()` function, with `var` to create a mutable value or with `let` to create an immutable value.

Go ahead and change `var` to `let` in the `main()` function above and run it. You'll get a compiler error like this:

```
error: Expression [15]:7:5: expression must be mutable for in-place operator destination
    x += 1
    ^
```

That's because `let` makes it immutable, so you can't increment the value.

And if you delete `var` completely, you'll get an error because `fn` functions require explicit variable declarations (unlike `def` functions).

Finally, notice that the `x` variable has an explicit `Int` type specification. Declaring the type is not required for variables in `fn`, but it is desirable sometimes. If you omit it, Mojo infers the type, as shown here:

[3]:
```
fn do_math():
    let x: Int = 1
    let y = 2
    print(x + y)

do_math()
```

```
3
```

1.1.4 Function arguments and returns

Unlike local variables, arguments in an `fn` function must specify a type.

To return a value from an `fn` function, you must declare the return type with an arrow `->` at the end of the signature.

For example:

[4]:
```
fn add(x: Int, y: Int) -> Int:
    return x + y

z = add(1, 2)
print(z)
```

```
3
```

Argument mutability and ownership Now let's explore how argument values are shared with a function.

Notice that, above, `add()` doesn't modify `x` or `y`, it only reads the values. In fact, as written, the function *cannot* modify them because `fn` arguments are **immutable references**, by default.

In terms of argument conventions, this is called "borrowing," and although it's the default for `fn` functions, you can make it explicit with the `borrowed` declaration like this (this behaves exactly the same as the `add()` above):

[5]:
```
fn add(borrowed x: Int, borrowed y: Int) -> Int:
    return x + y
```

If you want the arguments to be mutable, you need to declare the argument convention is `inout`. This means that changes made to the arguments *in*side the function are visible *out*side the function.

For example, this function is able to modify the original variables:

[6]:
```
fn add_inout(inout x: Int, inout y: Int) -> Int:
    x += 1
    y += 1
    return x + y

var a = 1
var b = 2
c = add_inout(a, b)
print(a)
print(b)
print(c)
```

```
2
3
5
```

Another option is to declare the argument as `owned`, which provides the function full ownership of the value (it's mutable and guaranteed unique). This way, the function can modify the value and not worry about affecting variables outside the function. For example:

[7]:
```
from String import String

fn set_fire(owned text: String) -> String:
    text += " "
    return text

fn mojo():
    let a: String = "mojo"
    let b = set_fire(a)
    print(a)
    print(b)

mojo()
```

```
mojo
mojo
```

In this case, Mojo makes a copy of `a` and passes it as the `text` argument. The original `a` string is still alive and well.

However, if you want to give the function ownership of the value and **do not** want to make a copy (which can be an expensive operation for some types), then you can add the `^` "transfer" operator when you pass `a` to the function. The transfer operator effectively destroys the local variable name—any attempt to call upon it later causes a compiler error.

Try it above by changing the call to `set_fire()` to look like this:

```
let b = set_fire(a^)
```

You'll now get an error because the transfer operator effectively destroys the `a` variable, so when the following `print()` function tries to use `a`, that variable isn't initialized anymore.

If you delete `print(a)`, then it works fine.

Note: Currently, Mojo always makes a copy when a function returns a value.

1.2 Structures

You can build high-level abstractions for types (or "objects") in a `struct`. A `struct` in Mojo is similar to a `class` in Python: they both support methods, fields, operator overloading, decorators for metaprogramming, etc. 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 classes in the future.)

For example, here's a basic struct:

[8]:
```
struct MyPair:
    var first: Int
    var second: Int

    # This "initializer" behaves like a constructor in other languages
    fn __init__(inout self, first: Int, second: Int):
        self.first = first
        self.second = second

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

And here's how you can use it:

[9]:
```
def pair_test() -> Bool:
    let p = MyPair(1, 2)
    # Uncomment to see an error:
    # return p < 4 # gives a compile time error
    return True
```

If you're familiar with Python, then the `__init__()` method and the `self` argument should be familiar to you. If you're *not* familiar with Python, then notice that, when we call `dump()`, we don't actually pass a value for the `self` argument. The value for `self` is automatically provided with the current instance of the struct (similar to the `this` name used in some other languages).

For much more detail about structs and other special methods like `__init__()` (also known as "dunder" methods), see the programming manual.

1.3 Python integration

Although Mojo is still a work in progress and is not a full superset of Python yet, we've built a mechanism to import Python modules as-is, so you can leverage existing Python code right away. Under the hood, this mechanism uses the CPython interpreter to run Python code, and thus it works seamlessly with all Python modules today.

For example, here's how you can import and use NumPy (you must have Python `numpy` installed, but in this case, it's already installed in the Mojo Playground):

[10]:
```
from PythonInterface import Python

let np = Python.import_module("numpy")

ar = np.arange(15).reshape(3, 5)
print(ar)
print(ar.shape)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
(3, 5)
```

Note: Mojo is not a feature-complete superset of Python yet, so some language patterns or features from Python currently do not work. So, you can't always copy-paste Python code and run it in Mojo. Please report any issues you find on GitHub.

1.4 Next steps

We hope this notebook covered enough of the basics to get you started. It's intentionally brief, so if you want more details, check out the Mojo programming manual. Also see the other Mojo notebooks for more interesting and complex code examples.

And to see all the available Mojo APIs, check out the Mojo standard library reference.