

Variables

A variable is a name that holds a value or object. All variables in Mojo are mutable—their value can be changed. (If you want to define a constant value that can't change at runtime, see the [alias keyword](#).)

Mojo has two kinds of variables:

- Explicitly-declared variables are created with the `var` keyword, and may include [type annotations](#).

```
var a = 5
var b: Float64 = 3.14
```

- Implicitly-declared variables are created with an assignment statement:

```
a = 5
b = 3.14
```

Both types of variables are strongly-typed: the variable receives a type when it's created, and the type never changes. You can't assign a variable a value of a different type:

```
count = 8 # count is type Int
count = "Nine?" # Error: can't implicitly convert 'StringLiteral' to 'Int'
```

Some types support [implicit conversions](#) from other types. For example, an integer value can implicitly convert to a floating-point value:

```
var temperature: Float64 = 99
print(temperature)
```

```
99.0
```

In this example, the `temperature` variable is explicitly typed as `Float64`, but assigned an integer value, so the value is implicitly converted to a `Float64`.

Implicitly-declared variables

You can create a variable with just a name and a value. For example:

```
name = String("Sam")
user_id = 0
```

Implicitly-declared variables are strongly typed: they take the type from the first value assigned to them. For example, the `user_id` variable above is type `Int`, while the `name` variable is type `String`. You can't assign a string to `user_id` or an integer to `name`.

Implicitly-declared variables are scoped at the function level. You create an implicitly-declared variable the first time you assign a value to a given name inside a function. Any subsequent references to that name inside the function refer to the same variable. For more information, see [Variable scopes](#), which describes how variable scoping differs between explicitly- and implicitly-declared variables.

Explicitly-declared variables

You can declare a variable with the `var` keyword. For example:

```
var name = String("Sam")
var user_id: Int
```

The `name` variable is initialized to the string "Sam". The `user_id` variable is uninitialized, but it has a declared type, `Int` for an integer value. All explicitly-declared variables are typed—either explicitly with a [type annotation](#) or implicitly when they're initialized with a value.

Since variables are strongly typed, you can't assign a variable a value of a different type, unless those types can be [implicitly converted](#). For example, this code will not compile:

```
var user_id: Int = "Sam"
```

There are several main differences between explicitly-declared variables and implicitly-declared variables:

- An explicitly-declared variable can be declared without initializing it:

```
var value: Float64
```

- Explicitly-declared variables follow [lexical scoping](#), unlike implicitly-declared variables.

Type annotations

Although Mojo can infer a variable type from the first value assigned to a variable, it also supports static type annotations on variables. Type annotations provide a more explicit way of specifying the variable's type.

To specify the type for a variable, add a colon followed by the type name:

```
var name: String = get_name()
```

This makes it clear that `name` is type `String`, without knowing what the `get_name()` function returns. The `get_name()` function may return a `String`, or a value that's implicitly convertible to a `String`.

Note

You must declare a variable with `var` to use type annotations.

If a type has a constructor with just one argument, you can initialize it in two ways:

```
var name1: String = "Sam"
var name2 = String("Sam")
```

Both of these lines invoke the same constructor to create a `String` from a `StringLiteral`.

Late initialization

Using type annotations allows for late initialization. For example, notice here that the `z` variable is first declared with just a type, and the value is assigned later:

```
fn my_function(x: Int):
    var z: Float32
    if x != 0:
        z = 1.0
    else:
        z = foo()
    print(z)

fn foo() -> Float32:
    return 3.14
```

If you try to pass an uninitialized variable to a function or use it on the right-hand side of an assignment statement, compilation fails.

```
var z: Float32
```

```
var y = z # Error: use of uninitialized value 'z'
```



Note

Late initialization works only if the variable is declared with a type.

Implicit type conversion

Some types include built-in type conversion (type casting) from one type into its own type. For example, if you assign an integer to a variable that has a floating-point type, it converts the value instead of giving a compiler error:

```
var number: Float64 = 1
```

1

As shown above, value assignment can be converted into a constructor call if the target type has a constructor that takes a single argument that matches the value being assigned. So, this code uses the `Float64` constructor that takes an integer: `__init__(inout self, value: Int)`.

In general, implicit conversions should only be supported where the conversion is lossless.

Implicit conversion follows the logic of [overloaded functions](#). If the destination type has a single-argument constructor that takes an argument of the source type, it can be invoked for implicit conversion.

So assigning an integer to a `Float64` variable is exactly the same as this:

```
var number = Float64(1)
```

Similarly, if you call a function that requires an argument of a certain type (such as `Float64`), you can pass in any value as long as that value type can implicitly convert to the required type (using one of the type's overloaded constructors).

For example, you can pass an `Int` to a function that expects a `Float64`, because `Float64` includes a constructor that takes an `Int`:

```
fn take_float(value: Float64):  
    print(value)
```

```
fn pass_integer():  
    var value: Int = 1  
    take_float(value)
```

For more details on implicit conversion, see [Constructors and implicit conversion](#).

Variable scopes

Variables declared with `var` are bound by *lexical scoping*. This means that nested code blocks can read and modify variables defined in an outer scope. But an outer scope **cannot** read variables defined in an inner scope at all.

For example, the `if` code block shown here creates an inner scope where outer variables are accessible to read/write, but any new variables do not live beyond the scope of the `if` block:

```
def lexical_scopes():
    var num = 1
    var dig = 1
    if num == 1:
        print("num:", num) # Reads the outer-scope "num"
        var num = 2        # Creates new inner-scope "num"
        print("num:", num) # Reads the inner-scope "num"
        dig = 2            # Updates the outer-scope "dig"
    print("num:", num)     # Reads the outer-scope "num"
    print("dig:", dig)     # Reads the outer-scope "dig"
```

```
lexical_scopes()
```

```
num: 1
num: 2
num: 1
dig: 2
```

Note that the `var` statement inside the `if` creates a **new** variable with the same name as the outer variable. This prevents the inner loop from accessing the outer `num` variable. (This is called "variable shadowing," where the inner scope variable hides or "shadows" a variable from an outer scope.)

The lifetime of the inner `num` ends exactly where the `if` code block ends, because that's the scope in which the variable was defined.

This is in contrast to implicitly-declared variables (those without the `var` keyword), which use **function-level scoping** (consistent with Python variable behavior). That means, when you change the value of an implicitly-declared variable inside the `if` block, it actually changes the value for the entire function.

For example, here's the same code but *without* the `var` declarations:

```
def function_scopes():
    num = 1
    if num == 1:
```


```
print(num)    # Reads the function-scope "num"
num = 2       # Updates the function-scope variable
print(num)    # Reads the function-scope "num"
print(num)    # Reads the function-scope "num"
```

```
function_scopes()
```

```
1
2
2
```

Now, the last `print()` function sees the updated `num` value from the inner scope, because implicitly-declared variables (Python-style variables) use function-level scope (instead of lexical scope).

Was this page helpful?  

 Edit this page