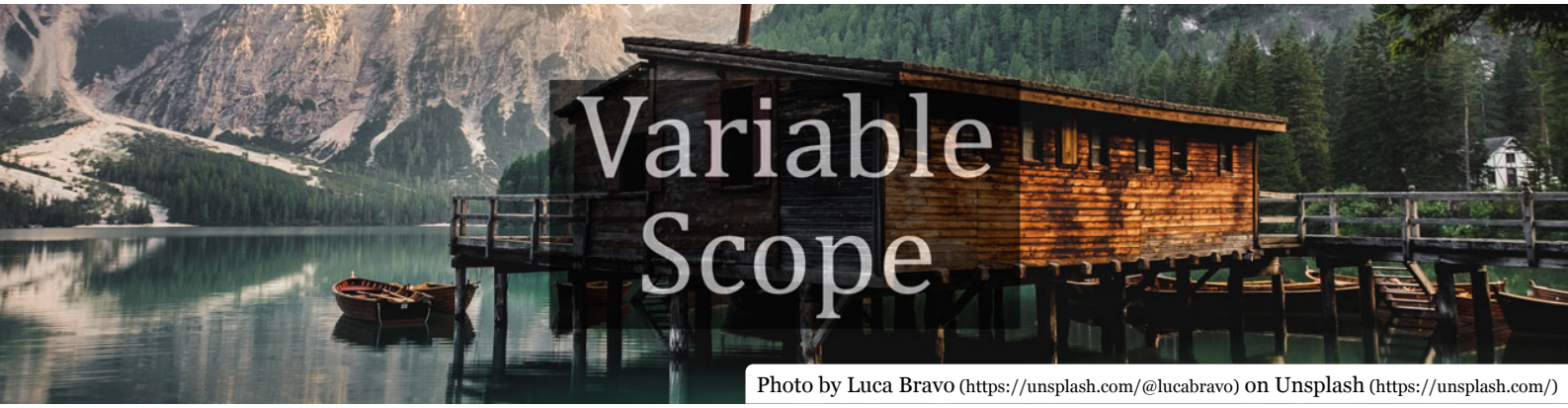

Photo by Luca Bravo (https://unsplash.com/@lucabravo) on Unsplash (https://unsplash.com/)

Variable Scope

6 minute read

In this lesson we will learn what is the **scope** of a variable and how scopes can be used to rule when a variable should be accessible in our program.

The **scope** of a variable is the region of a program where the variable is known and accessible. A variable may live in two kind of scopes: the **global scope** or a **local scope**.

Scope

A variable in the **global scope** is accessible everywhere in the program and can be modified by any part of the code. When we define a variable in the REPL or outside of a function, for example, we create a global variable.

A variable in a **local scope** is only accessible in that scope and in other scopes eventually defined inside it.

In Julia there are several constructs which introduces a scope:

Construct	Scope type	Scope blocks it may be nested in
`module` (https://docs.julialang.org/en/v1/base/base/#module), `baremodule` (https://docs.julialang.org/en/v1/base/base/#baremodule)	global	global
interactive prompt (REPL)	global	global
(mutable) `struct` (https://docs.julialang.org/en/v1/base/base/#struct), `macro` (https://docs.julialang.org/en/v1/base/base/#macro)	local	global
`for` (https://docs.julialang.org/en/v1/base/base/#for), `while` (https://docs.julialang.org/en/v1/base/base/#while), `try-catch-finally` (https://docs.julialang.org/en/v1/base/base/#try), `let` (https://docs.julialang.org/en/v1/base/base/#let)	local	global or local
functions (either syntax, anonymous & do-blocks)	local	global or local
comprehensions, broadcast-fusing	local	global or local

As you can see, some constructs introduce a global scope (for example each module has its separate global scope) and others introduce a local scope (for example `functions`, `for` loops and `let` blocks).

Let's now see in more details how to work with scopes and in which scope it is better to define a variable, depending on its usage.

Local Scope

Let's start with a construct we are already familiar with: a function. A function declaration introduce a scope, which means that each variable declared inside a function lives inside the function: it can be accessed freely inside the function but cannot be accessed outside the function, which is good! Thanks to this property we can use the names most suitable for our variables (x, y, z, etc.) without the risk of clashing with the declaration of other functions.

If a value computed inside a function is needed outside the function it is a good idea to return that value instead of modifying a global constant external to the function. As a rule of thumb, a function should rely only on its input parameters and return only the variable which may be useful for further computations.

Let's see an example of a variable which exist inside a function (local scope) but doesn't exist in the global scope:

```
function example1()
    z = 42
    return
end

>>> z
ERROR: UndefVarError: z not defined
```

Although it is not advisable, it is possible to specify that a variable should be accessed in the global scope through:

```
function example2()
    global z = 42
    return
end

>>> example2()
>>> z
42
```

A better approach is instead to return z and let the user perform the allocation of z:

```
function example3()
    z = 42
    return z
end

>>> z = example3()
>>> z
42
```

In the case where it is necessary to distinguish between a variable which exists both in the local and global scope, it is possible to indicate the one that needs to be used through the `global` or `local` annotation before the desired variable (for more details see this page (https://docs.julialang.org/en/v1/manual/variables-and-scoping/index.html) of the Julia documentation).

`let` construct

The `let` construct can be used to introduce a new local scope. It is useful, for example, when you want to perform some computations but you don't want the intermediate results/variables to pollute your current scope.

The let block will be able to access all the local (or global) variables available in its parent scope and will have its own set of local variables. It is also possible to specify some initial values to mimic the execution of a function once.

```
a = let
    i=3
    i+=5
    i # the value returned from the computation
end

>>>a
8

b = let i=5
    i+=42
    i
end

>>>b
47

c = let i=10
    i+=42
    i
end

>>>c
52

>>>i
UndefVarError: i not defined
```

As you can see, `let` blocks are pretty convenient when it comes to splitting computations over several lines, but there are other possible uses as shown here (https://docs.julialang.org/en/v1/manual/variables-and-scoping/index.html#Local-Scope-1).

`let` blocks are somewhat similar to `begin` blocks, although `begin` blocks don't introduce a local scope:

```
d = begin
    i=41
    i+=1
    i
end

>>>d
42

>>>i
42
```

Global scope

Whenever we define a variable in the REPL or in general outside a construct which introduces a local scope, we place a variable in the **global scope**. The global scope is accessible everywhere in the program and a variable in the global scope can be modified by any part of the code. As such, it is generally advisable to **avoid using global variables** as much as possible, in fact since global variables can change their type and value at any time, they cannot be properly optimised by the compiler.

Constants

One way to mitigate this performance issue is to define global variables as constants through the `const` annotation. When we think of a constant we generally imagine a variable which is immutable, for example the speed of light: the speed of light is a number and so it could be expressed by a `Float64`, there is no need to change its type, once it has been defined, to a `String`, in this case.

A constant in Julia is a variable which cannot change its type once it is defined (Julia will throw an error if there is an attempt to modify the type of a constant) and Julia will show a warning message if we try to modify the value of a constant. Since a constant is "type immutable" it can be properly optimised by the compiler and there are fewer performance issues.

```
>>> const C = 299792458 # m / s, this is an Int
299792458

>>> C = 300000000 # change the value of C
WARNING: redefining constant C

>>> C = 2.998 * 1e8 # change the type of C, not permitted
invalid redefinition of constant C
```

Modules

> If you don't already know what a module is you don't have to worry, we will talk about modules in the next lessons and you can come back to this part later!

As we have anticipated, modules introduce separate global scope, which means that a variable which is known inside a module will not be accessible outside of the module unless it is exported or it is accessed through the `ModuleName.varName` notation.

```
1  module ScopeTestModule
2  export a1
3  a1 = 25
4  b1 = 42
5  end # end of module
6
7  using .ScopeTestModule
8
9  >>>a1
10 25
11
12 >>>b1
13 UndefVarError: b1 not defined
14
15 >>>ScopeTestModule.b1
16 42
17
18 >>>ScopeTestModule.b1=26
19 cannot assign variables in other modules
```

At line 9-10 we can see that the `a1` variable, which is exported by the module, can be accessed directly without specifying the scope of the variable, while on line 12-13 and 15-16 we can see that `b1` can only be accessed by specifying where the variable lives (i.e. inside `ScopeTestModule`). At line 18-19 we can see that it is not possible to directly modify a variable which is defined inside another module.

Conclusions

When you are writing some quick computations in the REPL or you are writing a simple script, you don't need to care about what goes in the global scope and what not, but if you are writing a piece of code that has to be reusable (like a library for example) and needs to be fast, there are some guidelines which should be followed:

- **Avoid using global variables** as much as possible, if global variables are needed **define them as `const`**
- Pass all the required parameters to a function instead of defining them as global variables, a function should be able to ideally operate only on its input.
- Use functions and let blocks to introduce local scopes where you can define as many variables as you desire without incurring in the risk of overlapping variable names with those used in other parts of your code.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: December 20, 2019

LEAVE A COMMENT

4 Comments **techytok** **Privacy Policy** **Login**

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS**

Name

bocc • 7 months ago

question: can you create a constant struct if you use the new() method in it's constructor?

• Reply • Share ›

aureamerio Mod bocc • 6 months ago • edited

Yes, you can! Mutable and immutable structures differ only in the behavior after creation.
For example you can write an immutable Person structure in this way:

```
struct Person
height::Float64
weight::Float64
BMI::Float64
function Person(heightInMeters::Float64,
weightInKilos::Float64)
BMI = weightInKilos / heightInMeters^2
new(heightInMeters, weightInKilos, BMI)
end
end
```

PS: apparently the indentation in comments is messed up, but you should be able to understand anyway...

• Reply • Share ›

bocc aureamerio • 6 months ago

Thanks for you time, this is definitely doable, but I would prefer to use something like:

n = new()
n.field1. = ...
n.field2 = ...
return n

In order to avoid mixing up the field orders (as they would be of identical type). This of course can only be done with an 'internal' constructor, but if I want to do this, I need to declare this struct as mutable.

 • Reply • Share ›

aureamerio Mod ➚ bocc • 6 months ago • edited

I'm not sure what you mean... If the problem is the order of the arguments in the constructor (i.e function Person) you could use multiple dispatch to add a constructor which uses **positional arguments** and one which uses **keyword arguments** like in the example below:

```
struct Person
height::Float64
weight::Float64
BMI::Float64

# positional arguments
function Person(heightInMeters::Float64,
weightInKilos::Float64)
BMI = weightInKilos / heightInMeters^2
new(heightInMeters, weightInKilos, BMI)
end
```

/