3d. Variable Scope & Relevance of Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

Variable scope refers to the code block in which a variable is accessible. The concept allows us to distinguish between **global variables**, which are accessible in any part of the code, and **local variables**, which are confined to specific blocks like functions or loops. The existence of scopes determines that the same variable `x` could refer to different objects, depending on where it's called.

When it comes to functions, Julia adheres to specific rules for variable scope. Specifically, given a variable `x` defined outside a function:

- if a new variable `x` is defined inside a function or is passed to a function as an argument, then `x` is considered *local* to that function. This means that any reference to `x` within the function refers to the local variable, without any relation to the variable `x` defined outside the function,

- if a function doesn't define a new `x` nor `x` is a function argument, then `x` refers to the variable defined outside the function (i.e., the global variable).

In this section, we'll show how these rules work in practice.

GLOBAL AND LOCAL VARIABLES

A variable that is local to a function exists solely within that function's scope. This means that these variables cease to exist once the function finishes executing. Consequently, any attempt to reference local variables outside the function will result in an error.

Variables local to a function encompass:

1. the function arguments,
2. the variables defined in the function body.

Any other variable included in a function that's not *i)* or *ii)* necessarily refers to a global variable.

Understanding which variables are local or global is essential for predicting a program’s behavior. This is because a local variable may share the same name as a global one, without them being related. The following examples help clarify the differences between global and local variables.

```
x = "hello"

function foo(x)                         # 'x' is local, ur
    y = x + 2                           # 'y' is local, 'x

    return x,y
end
```

```
julia> foo(1)
1                            # local x
3                            # local y

julia> x
"hello"

julia> y
ERROR: UndefVarError: y not defined
```

```
z = 2

function foo(x)
    y = x + z                   # 'x' refers to t

    return x,y,z
end
```

```
julia> foo(1)
1                        # local x
3                        # local y
2                        # global z

julia> x
ERROR: UndefVarError: x not defined

julia> z
2
```

THE ROLE OF FUNCTIONS

In programming, **functions** can be understood **as self-contained mini-programs to represent specific tasks**. Under this interpretation, local variables simply act as labels that help articulate the mechanics of the task. Consequently, their inaccessibility outside the function emerges naturally. [1]

To explain this view of functions, consider a variable `x`, along with another variable `y` computed by transforming `x` through a function `f`. In particular, assume a transformation that doubles `x`, so that `y = 2 * x`. The following are two approaches to calculating `y`.

```
x = 3

double() = 2 * x
y        = double()
```

```
x = 3

double(x) = 2 * x
y         = double(x)
```

```
x = 3

double(🐒) = 2 * 🐒
y          = double(x)
```

The function in Approach 1 relies on the global variable `x`. This practice is highly discouraged for several reasons. Firstly, it prevents the reusability of the function, as it's specifically designed to double the global variable `x`, rather than acting as a mini-program that doubles *any* variable.

Second, the inclusion of the global variable `x` compromises the function's self-containment, as the function's output depends on the value of `x` at the moment of execution. If you work on a long project, this will turn the code prone to bugs.

Lastly, global variables have a detrimental impact on performance, a topic we'll study later on the website. In fact, global variables in Julia are directly a performance killer.

In contrast, Approach 2 refers to `x` as a local variable. This `x` is unrelated to the global variable `x`—it simply serves as a label to identify the variable to be doubled. Indeed, we could've replaced `x` with any other label, as demonstrated in Approach 3 through the monkey emoji, `🐒`.

By avoiding referencing any variable outside its scope, Approach 2 makes the function self-contained. This allows users to easily anticipate the consequence of executing `double` through a simply inspection of the function, eliminating the need to review the entire codebase. Thus, Approach 2 aligns with the interpretation of a function as a self-contained mini-program: the function embodies the task of doubling a variable, turning the function reusable and applicable to any variable. In this context, applying `double` to the global variable `x` becomes just one possible application.

RECOMMENDATIONS FOR THE USE OF FUNCTIONS

Structuring code around functions offers numerous advantages. However, to fully realize these benefits, users must adhere to certain principles when writing code. This section outlines a few of them and should be considered as a mere introduction to the subject. The topic will be investigated further, when we explore high performance.

AVOID GLOBAL VARIABLES IN FUNCTIONS

Global variables are strongly discouraged. This is not only due to the reasons mentioned previously, but also because they can have a devastating impact on performance. The easiest solution to this issue is to pass global variables as function arguments. This practice will actually become second nature once you start viewing functions as self-contained mini-programs. Specifically, by adopting this perspective, you'll conceive local variables as labels to describe a task, rather than references to global variables. This shift in mindset can help you write more efficient and maintainable code.

AVOID REDEFINING VARIABLES WITHIN FUNCTIONS

The suggestion applies to both local variables and function arguments. Redefining these variables can have several disadvantages, including reduced code readability and potential performance degradation. Therefore, it's recommended that you define new variables instead of redefining existing ones. This approach is demonstrated in the following example.

```
function foo(x)
   x      = 2 + x          # redefines the argum

   y      = 2 * x
   y      = x + y          # redefines a local v
end
```

```
function foo(x)
   z      = 2 + x          # new variable

   y      = 2 * x
   output = z + y          # new variable
end
```

(OPTIONAL) - Another Issue of Redefining Variables

×

ANOTHER ISSUE OF REDEFINING VARIABLES

Within a function, Julia will throw an error if you perform a computation using a global variable `x` and then redefine `x`. For example:

▼ Code

```julia
x = 2

function foo()
    y = x + 2
    x = x + 4

    return x
end
```

```
julia> foo()
ERROR: UndefVarError: x not defined
```

The error arises because Julia reads the entire function before its execution: the function's second line assigns a value to `x`, causing Julia to consider `x` a local variable. The consequence is that, when the function is run, `x` in `y = x + 2` is interpreted as a local variable that hasn't been defined yet.

MODULARITY

We've emphasized the importance of viewing functions as self-contained mini-programs, designed to perform specific tasks. This perspective leads us to highlight the importance of **modularity**: the practice of breaking down a program into multiple small functions, each with its own distinct purpose, inputs, and outputs.

The primary benefit of modularity is the ability to work with independent code blocks. By keeping these blocks separate, we can decompose complex problems into multiple manageable tasks, making it easier to test and debug code. Additionally, modularity makes it possible to eventually improve or substitute parts of the code, without breaking the entire program.

A helpful way to understand this principle is by considering the analogy of building a Lego minifigure. In the first step, multiple blocks are created independently, each representing a specific part of the figure, such as the legs, torso, arms, and head. Then, in the second stage, these individual blocks are brought together and assembled into an integrated minifigure.

This two-step approach offers several advantages. By focusing on each block individually, we can concentrate and refine each part without worrying about the entire structure. Additionally, it provides great flexibility: since each block is created independently, we can modify specific blocks without having to rebuild the entire figure. For instance, if we want to change the figure's head, we can simply swap out the corresponding block, without starting from scratch.

The principle of modularity is closely tied to the suggestion of writing short functions. Some proponents even argue that functions should be limited to fewer than five lines of code Indeed, entire books have been written based on this principle. Although this viewpoint may be considered rather extreme, it clearly emphasizes the advantages of avoiding lengthy functions.

(OPTIONAL) - Example of Modularity

×

EXAMPLE OF MODULARITY

Coming up with mock scenarios illustrating the advantages of modularity can be tricky. This occurs because the same function could be deemed modular enough, depending on the context and your goals. Moreover, modularity may become detrimental if it impairs code readability. On top of this, note that making code more modular may not be justified if there aren't plans to reuse it.

With these challenges in mind, we'll present an example that showcases the potential benefits of modularity. The example focuses on calculating the cost of purchasing a product, when this is subject to a percentage tax over the total value. Specifically, consider the following two scripts to compute this.

```
expenditure(price, quantity, tax_rate) = price
```

```
value_before_taxes(price, quantity)       = pri
valueAdded_tax(price, quantity, tax_rate) = pri

expenditure(price, quantity, tax_paid) = value_
```

Con
1,0(
min
purchasing two iPhones.

```
#functions to compute expenditure
expenditure(price, quantity, tax_rate) = price





#information
price    = 1000
quantity = 2
tax_rate = 5 / 100

#computation
expenditure_iPhones = expenditure(price, quanti
```

```
julia> expenditure_iPhones
2100.0
```

Approach 2 is more verbose, but also more readable, allowing the user to quickly grasp the code's purpose. In contrast, Approach 1 requires the reader to carefully examine each term of `expenditure` to identify its functionality.

Furthermore, **Approach 2 is more modular**, as it breaks
dow
mo
paid
imm
app
reas

Firs
App
mul
reco
App
incl
com

```
#functions to compute expenditure
value_before_taxes(price, quantity)        = pri
valueAdded_tax(price, quantity, tax_rate) = pri

expenditure(gross_value, tax_paid) = gross_valu

#information
price    = 1000
quantity = 2
tax_rate = 5 / 100

#computation
gross_value          = value_before_taxes(price,
tax_paid             = valueAdded_tax(price, qua

expenditure_iPhones = expenditure(gross_value,
```

```
julia> expenditure_iPhones
2100.0
```

testing code blocks separately. This feature is critical for ensuring proper code functioning, and can additionally simplify code debugging and the identification of performance bottlenecks.

FOOTNOTES

1. Local variables play a similar role to integration variables in math. Formally, \(\mathrm{d}t \) in \(\int f\left(t\right)\ \) for some function \(f\) is simply a symbol indicating over which variable we're integrating. The integral could be equivalently expressed using any other integration variable, such as \(x\) in \(\int f\left(x\right)\,\mathrm{d}x \).