Lesson 6, Week 2: Scope I (functions)

AIM

— To introduce the concept of scope as it applies in Julia programs

After this lesson, you will be able to

* Say what the scope is of a block of code
* Say what it means for a block of code to have local scope
* Explain the difference between local scope and global scope
* Explain how a function in Julia treats variables from the global scope
* Use the keyword `global` to make global variables accessible to writing inside a function
* Explain why, even though they're global, the values inside an array can be modified inside a function

What is scope, in the sense of programming?

The term *scope* refers to which variables can be accessed in a particular block of code.

So what is a block of code? Well, any portion of a program that doesn't leave out any lines between the first and last lines in the block. So the program as a whole is a code block, and the variables that are accessible to the whole program are called the *global* variables[1].

However, some code blocks introduce *local scope*. Variables that are local to that scope cannot be accessed by code outside the code block that introduces the local scope[2].

You have already seen one way to make a local scope: comprehensions.

DEMO:
```
xyz = "abcd"
vecxyz = [locvar^4 for locvar in xyz]
```

We see that the variable `locvar` is available inside the comprehension, but not outside it.

[1] Roughly speaking, the global variables are those in the namespace of the program as a whole. If there is only one namespace, then all its variables are global.

[2] Whether global variables are accessible in the local scope is a delicate matter, as we will see, and the rules are not the same for all ways to create local scope.

Please be aware that a variable may be accessible for reading or for writing or for both. On this course, all code blocks that do not use local scope are part of the global scope[3].

Which keywords introduce a new scope?

For us at the moment, the important point is that the keyword `function` creates a local scope[4]. All variables that are created inside the function body are local to that scope and for all practical purposes are invisible to any code outside the function body.

Note that even if you re-use the name of a global variable to create a local variable with the same name, then you have created two different variables, one local, one global. This is obviously a conflict[5], and should be avoided.

Global variables are accessible for reading inside a function, but not for writing—with the exception that the keyword `global` can be used to make a global variable accessible for writing in a block with local scope.

Some perhaps unexpected effects of Julia's scoping rules for functions

Consider the functions `eg1` and `eg2` defined below

```
eg1(x) = x+y
```

```
eg2(x, y) = x+y
```

DEMO: define `x,y = 2,3` and explain what happens in the cases `eg1(x)`, `eg2(x, y)`, `eg1(6)` and `eg2(6, 7)`

We see that in `eg1`, the value of `y` is the global value, while in `eg2`, the value is whatever is passed to the function in the second position. In other words, it is not necessary to use the variable names `x` or `y` when writing a function call[6]. It should be clear that, by putting the name `y` in the argument list of `eg2` when we write the code that defines the function, we make `y` a *local* variable, visible only in the local scope of `eg2`.

Let us now try to change the value of the global variable `y` in these functions. We have to use the keyword `function` in order to get multiline bodies for both functions:

[3] This is because we work only in the REPL, which has only one namespace.
[4] We'll see some of the others later; there are 10 such keywords (as of Julia 1.1).
[5] Alas, it happens very easily and can lead to bugs that are very hard to find.
[6] Reminder: a function call is code where you tell Julia to execute the function.

```
function eg1(x)
    z = x+y
    y = 11
    return z
end
```

```
function eg2(x, y)
    z = x+y
    y = 11
    return z
end
```

[DEMO: test the same values as above, but also test whether the global value of `y` changes. Discuss.]

[DEMO: for `eg1`, replace `y = 11` with `global y = 1`, repeat test, discuss.]

[DEMO: explain why the same change doesn't work for `eg2`.]

We see that specifying that `y` as `global` insde the function body works only if it hasn't already been specified as local in the argument list. And by the way, we don't have to use combine `global` with assignment as we did there, we can have the line `global y` anywhere in the function body and just use `y=11` as before.

The special case of arrays

We have said that global variables cannot be modified inside a function (at least, not without using the keyword `global`). But arrays may seem to violate that rule:

```
function modaa()
  aa = 7
end
```

```
function modbb()
  bb[2] = 7
end
```

DEMO:
```
aa, bb = 123, [1, 2]
modaa(); modbb()
aa, bb
```

We see that as expected `modaa` cannot modify the scalar variable `aa`, but `modbb` succeeds in modifying one of the elements of `bb`.

However, in this case the variable `bb` as a vector doesn't change: it remains two `Int64` numbers. In that sense, it satisfies the rule. It is the values inside the array that are not subject to the rule.

This is not behaviour peculiar to functions; it is general. When you create an array, Julia tries to not to have to allocate new memory. Compare the code on the left to the code on the right:

```
aa = 2
bb = aa
bb = 3
println("$aa, \t$bb")
```

```
aa = [1, 2]
bb = aa
bb[2] = 3
println("$aa, \t$bb")
```

On the left, when `bb=aa` is executed, the name `bb` binds to same bit of memory as the name `aa`. When `bb=3` is executed, the name `bb` simply binds to a new bit of memory. So the result is that `aa` and `bb` end up with different values.

On the right, when `bb=aa` is executed, it also binds to the same memory as `aa`. But this `aa` is an array, it is potentially enormous, because there is almost no limit to the size an array can have in Julia. The designers of Julia favour performance, so they wish to avoid copying arrays whenever they can. This means that when `bb[2]=3` is executed, no new memory is used. Instead, the memory that `aa` and `bb` share is changed. That is why changing an element in `bb` changes the corresponding element `aa`, so that they remain equal[7].

Review and summary

* Any part of code can be a block, all the is needed is that it doesn't leave out any lines from its first to its last line
* The scope of a block of code is all the variables that are visible inside that code
* Variables can be visible for reading only or for reading and writing
* A block with local scope means that variables created in that block are not visible outside it
* Comprehension creates local scope
* The `function` keyword creates local scope that ends with its `end` keyword[8]
* Global variables are visible inside local scope for reading only
* Any variables that are mentioned in the input to a function call are in the local scope of that function
* The keyword `global` can be applied to a variable name inside the function body[9], and makes the global variable of that name available to writing inside the function body
* Values inside arrays that are in global scope can be modified in the local scope of a function
* When an array-valued variable is assigned to a second names, both names always bind to the same value; changing elements in that value changes both variables.

[7] This is one of the ways in which Julia differs from many other languages.
[8] In other words, the code in the function body has local scope
[9] Provided the function body does not have a local variable of that name.