Functions and Scoping

Programming Structures

Gaston Sanchez

STAT 33B, Fall 2025

About

In this part we discuss **scoping**, the act of finding the value of a variable.

The basic rules of scoping (how R find values of objects) are quite intuitive, although there are special situations that seem mind-boggling for the untrained useR.

You'll learn the formal rules of scoping as well as some of its more subtle details, which have important implications when creating and using functions.

Let's make some tacos

Super Basic Taco Examples

Example 1

```
tortilla = 8

carne = 7

taco = tortilla + carne

taco
```

You should have no trouble with this code and guessing its output.

Example 2

```
tortilla = 8
carne = 7

make_taco = function(ingred1, ingred2) {
  ingred1 + ingred2
}

make_taco(tortilla, carne)
```

You should have no trouble with this code and guessing its output.

Example 3

```
tortilla = 8
carne = 7

make_taco = function(tortilla, carne) {
  tortilla + carne
}

make_taco(tortilla = 2, carne = 3)
```

You should have no trouble with this code and guessing its output.

Global and Local Variables (example 1)

```
# example 1
tortilla = 8
carne = 7
taco = carne + tortilla
taco
```

- there are three variables: `tortilla`, `carne`, and `taco`
- all three variables are "global" *(assuming you run the code on the console, or in a code chunk)*
- technically speaking, these variables are in the so-called *Global Environment*

Global and Local Variables (example 2)

```
# example 2
tortilla = 8
carne = 7

make_taco = function(ingred1, ingred2) {
  ingred1 + ingred2
}
make_taco(tortilla, carne)
```

- there are two global variables: `tortilla` and `carne`
- there is one global function: `make_taco()`
- `ingred1` and `ingred2` are local variables—only accessible when `make_taco()` is called

Global and Local Variables (example 3)

```
# example 3
tortilla = 8
carne = 7

make_taco = function(tortilla, carne) {
  tortilla + carne
}
make_taco(tortilla = 2, carne = 3)
```

- there are two global variables: `tortilla = 8` and `carne = 7`
- there is one global function: `make_taco()`
- there are also two local variables `tortilla` and `carne`—arguments of `make_taco()`

Less Basic
Taco Examples

Example 4

```
tortilla = 8
carne = 7

make_taco = function(carne) {
  tortilla + carne
}

make_taco(2)
```

Most useRs have no trouble with this code and guessing its output.

Example 5

```
tortilla = 8
carne = 7

make_taco = function(carne) {
  tortilla = 4
  make_quesadilla = function(queso) {
    tortilla + queso
  }
  make_quesadilla(carne)
}

make_taco(2)
```

What is the output of `make_taco(2)`?

Example 6

```
tortilla = 8
carne = 7

make_taco = function(carne) {
  tortilla = 4
  make_quesadilla(carne)
}

make_quesadilla = function(queso) {
  tortilla + queso
}

make_taco(2)
```

What is the output of `make_taco(2)`?

Names and Values

When R encounters the **name** of a variable, how does it know what **value** to use for that name?

For example, when R sees the name `tortilla`, how does it decide between using `tortilla = 4` or `tortilla = 8` or something else?

Lexical Scoping

Lexical Scoping

- R uses **lexical scoping**: it looks up the values of names based on how a function is defined, not how it is invoked.
- "Lexical" here is not the English adjective that means relating to words or a vocabulary.
- Here "lexical" is a technical CS term that tells us that the scoping rules use a parse-time, rather than a run-time structure.

Scope and Scoping

- Scoping is the act of finding the value of a variable.
- A variable's **scope** is the section of code where it exists and is accessible.
- In R, when you create a function, you create a new scope.

Lexical Scoping Rules

R's lexical scoping follows four primary principles:

- Name masking
- Functions versus variables
- A fresh start
- Dynamic lookup

Scoping Principle:
Name Masking

Names defined inside a function **mask** names defined outside a function.

Name Masking: Local and Non-Local Variables

Names defined inside a function **mask** names defined outside a function

```
tortilla = 8
carne = 7

make_taco = function() {
  tortilla = 4
  carne = 2
  tortilla + carne
}

make_taco()
```

when calling `make_taco()` the local `tortilla = 4` and `carne = 2` mask the global `tortilla = 8` and `carne = 7`

Name Masking: Local and Non-Local Variables

The same applies to R functions since they are ordinary objects. This means that a function defined inside a function **masks** a function defined outside a function.

```
tortilla = function(x) x * 2

make_taco = function(carne) {
  tortilla = function(x) x + 10
  tortilla(4)
}

make_taco(2)
```

`tortilla` takes on 2 different values (*residing in different environments*): one is a global function, the other one is a local variable.

Name Masking and Lookup

If a name isn't defined inside a function, R looks one level up.

```
tortilla = 8
carne = 7

make_taco = function(carne) {
  tortilla + carne
}

make_taco(2)
```

`tortilla` is not defined inside `make_taco()`, so when this function is called, R looks one level up—and finds `tortilla = 8`.

Name Masking and Lookup

The same rules apply if a function is defined inside another function.

```
tortilla = 8

make_taco = function(carne) {
  tortilla = 4
  make_quesadilla = function(queso) {
    queso + tortilla
  }
  make_quesadilla(carne)
}

make_taco(2)
```

First, R looks inside the current function. Then, it looks where that function was defined (and so on, all the way up to the global environment). Finally, it looks in other loaded packages.

Name Masking: Local Variables are Private

Local variables cannot be accessed from outside:

```
make_burrito = function(ingredient1, ingredient2) {
  spread = ingredient1 * runif(1)
  ingredient2 / spread
}

spread
```

```
Error: object 'spread' not found
```

```
# exists(spread)
```

Note: The `exists()` function checks whether a variable is in scope.

Name Masking

A function can use variables defined outside (non-local), but only if those variables are in scope **where the function was defined**.

Here, `guacamole` is **not** within the scope of `make_supertaco()`

```
tortilla = 8

make_supertaco = function(carne) {
  tortilla = 4
  make_quesadilla(carne) + guacamole
}

make_quesadilla = function(queso) {
  guacamole = 3
  queso + tortilla
}

make_supertaco(2)
```

```
Error in make_supertaco(2): object 'guacamole' not found
```

Scoping Principle:
Functions versus Variables

In R, functions are ordinary objects. This means the scoping rules described above also apply to functions.

Functions versus Variables

When a function and a non-function share the same name (they must reside in different environments) R knows how to distinguish them.

```
tortilla = function(x) x + 1

make_taco = function(carne) {
  tortilla = 4
  tortilla(tortilla)
}

make_taco(2)
```

`tortilla` takes on 2 different values (*residing in different environments*): one is a global function, the other one is a local variable.

Note that using the same name for different things is confusing and best avoided!

Scoping Principle:
Fresh Start or Reset

Local variables are reset each time the function is called.

Fresh Start or Reset

What will happen the first time you run the following function? What will happen the second time?

```
make_snack = function() {
  if (!exists("nachos")) {
    nachos = 33
  } else {
    nachos = nachos + 5
  }
  nachos
}

make_snack()
```

```
[1] 33
```

```
make_snack()
```

```
[1] 33
```

Fresh Start or Reset

```
make_snack = function() {
  if (!exists("nachos")) {
    nachos = 33
  } else {
    nachos = nachos + 5
  }
  nachos
}

make_snack()
make_snack()
```

`make_snack()` always return the same value because every time a function is called a new environment is created to host its execution. This means that a function has no way to tell what happened the last time it was run.

Scoping Principle:
Dynamic Lookup

Dynamic lookup is what R uses to look for values when a function is run, not when it is created.

Dynamic Lookup

R looks for values when the function is run, not when the function is created.

```
make_taco = function(carne) {
  carne + tortilla
}

tortilla = 6
make_taco(2)
```

```
[1] 8
```

```
tortilla = 8
make_taco(2)
```

```
[1] 10
```

Summary

- Function definitions create a new scope.
- Local variables
  - Are private
  - Get reset for each call
  - Mask non-local variables (exception: function calls)
- *Lexical scoping*: where a function is **defined** determines which non-local variables are in scope.
- *Dynamic lookup*: when a function is **called** determines values of non-local variables.

My recommendation

Whenever possible, try to avoid using global variables.

Based on previous examples, I would write code like so:

```
make_taco = function(carne, tortilla) {
  carne + tortilla
}

make_quesadilla = function(queso, tortilla) {
  queso + tortilla
}

tortilla = 8
carne = 2
make_taco(carne, tortilla)
make_quesadilla(carne, tortilla)
```

My recommendation

I would also write code like this:

```
make_quesadilla = function(queso, tortilla) {
  queso + tortilla
}

make_taco = function(carne, tortilla) {
  make_quesadilla(carne, tortilla)
}

tortilla = 8
carne = 2
make_taco(carne, tortilla)
```

Writing functions that don't use global variables makes it easy to inspect, and also to debug.

Some Comments

- In R, the behavior of functions defined at the top-level (what I'm calling "top-global" functions) is different from the behavior of functions defined inside other functions (what I'm calling locally defined functions)
- In R, it matters **where** to look for values, and also **when** to look for values.
- Likewise, it matters where a function is created (the function environment), and also where a function is called (the execution environment). These two environments dictate the scoping mechanism used by R to find the values associated to variables.