Video 13: Scoping Rules

Stats 102A

Miles Chen, PhD

Related Reading

https://adv-r.hadley.nz/names-values.html#binding-basics

https://adv-r.hadley.nz/functions.html#lexical-scoping

Binding and Scoping

Introduction

Name Binding

Consider the code:

```
1 x <- c(1, 2, 3)
```

Informally, we often think of this command as creating an object called `x` which contains the vector of values 1, 2, and 3.

Technically, the assignment (`<-`) operator associates, or **binds**, the name `x` to the vector `c(1,2,3)`.

The command above actually does two things:

- Creates a vector object `c(1,2,3)` in memory.
- Binds the object `c(1,2,3)` to a name `x`.

The name, or **variable**, `x` is just a reference, or **pointer**, to the vector, i.e., `x` points to the memory location on your computer that contains the values `c(1,2,3)`.

Name Binding

Consider the following commands:

```
1 x <- c(1, 2, 3)
2 y <- x
3 z <- c(1, 2, 3)
```

What is the difference between `y` and `z`?

Variables in R are references to memory locations of objects, so `x` and `y` actually point to the same location. The `y` variable is another binding to the same object as `x`.

The `z` variable points to a different copy of `c(1,2,3)`.

While memory allocation issues are typically managed by R automatically, basic understanding of memory management can help in writing cleaner and faster code.

Copy-on-modify

```
1 x <- c(1, 2, 3)
2 y <- x
3 y[[3]] <- 4
4 x
```

```
[1] 1 2 3
```

Modifying y did not modify x. At first the names y and x pointed to the same object in memory.

When R saw that we were changing the value of y, it created a copy and modified the copy. The name x still points to the original object. The name y now points to a different object where the third value has been changed to 4.

Scope

Assignment is the act of binding a name to a value.

Scoping is act of finding the value associated with a name. The **scope** of a variable is the region of the code where the variable is defined.

The basic scoping rules are fairly intuitive. Consider the following code:

```
y <- 1
g <- function(x) {
  y <- 2
  x + y
}
```

Think: What is the output of `g(3)`?

Name Masking

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

```
y <- 1
g <- function(x) {
  y <- 2
  x + y  # finds y in the current environment
}
```

```
g(3)
```

```
[1] 5
```

`g(3)` returns 5. The value 3 is passed in as the argument `x`. The function returns the value `x + y`. `x` has the value 3. It searches for the value of `y` and finds 2 in the execution environment. The value of `y` in the global environment is irrelevant.

Scoping

The main scoping rule (known as **lexical scoping**): If R cannot find a variable in the function body's scope, it will look for it in the next higher scope, and so on.

While it is considered poor technique to reference objects not defined inside a function, R does its best to find the value rather than return an error.

Scoping

```
y <- 1
f <- function(x) {
  x + y  # can't find y in this environment, searches higher one
}
```

What is `f(3)`?

```
f(3)
```

```
[1] 4
```

`f(3)` returns 4. The value 3 is passed in as the argument `x`. The function returns the value `x + y`. `x` has the value 3. It searches for the value of `y` and can't find it in the execution environment. It searches for `y` in the global environment and finds the value 1.

More Scope

Variables defined within a scope only exist in that scope and only for the duration of that scope. If variables exist with the same name outside the scope, those are separate and do not interact with the variables inside the scope, nor are they overwritten.

A function is its own separate environment that only communicates to the outside world via the arguments going in and the returned object going out.

For example, if you execute the command `rm(list = ls())` inside a function (a terrible idea, btw), you would only delete the objects that have been defined inside the function.

The exception is the **super assignment** (`<<-`) operator which searches a higher scope and the global environment. This is a dangerous operator and should not be used regularly.

Scoping

```
x <- 1
y <- 1
z <- 1
f <- function() {
    y <- 2  # creates y inside the scope of f()
    g <- function() {  # this function is created inside f()
      z <- 3  # creates z inside the scope for g()
      return(x + y + z) # R uses scope rules to search for these va
      }
    return(g())
}
```

think: If I run `f()`, what will it return? After running `f()`, what are the values of x, y, z in the global environment?

Answers

```
1 f()
```

```
[1] 6
```

```
1 c(x, y, z)
```

```
[1] 1 1 1
```

Super Assignment

Super Assignment

The regular assignment `<-` operator always creates variables within the current environment.

The **super assignment** `<<-` operator never creates a variable in the current environment but instead **modifies** an existing variable found in the parent environment.

Warning: If (`<<-`) does not find an existing variable in the parent environment, it will climb the scope ladder until it finds the variable it is looking for. If it reaches the global environment without finding the variable, it **creates** the variable in the global environment.

Super assignment should generally be avoided.

The Super assignment operator

```
x <- 1; y <- 1; z <- 1
f <- function(){
  y <<- 3
  return(y)
}
```

```
c(x, y, z) # value of x, y, z before running f()
```

```
[1] 1 1 1
```

```
f()
```

```
[1] 3
```

```
c(x, y, z) # value of x, y, z after running f()
```

```
[1] 1 3 1
```

The output of `f()` is no surprise. However, after running the supper assignment operator, we see that the value of `y` in the global environment has changed to 3.

The Super assignment operator

```
x <- 1; y <- 1; z <- 1 # created in the global environment
f <- function() {
    y <<- 2 # modifies the value of y in the scope higher up (globa
    g <- function() { # g() defined inside the environment of f()
      z <<- 3  # modifies the value of z to 3 in the higher scope:
               # but does not find z in f, so climbs higher: global
      return(x + y + z) # R uses scope rules to search for these va
      }
    return(g())
}
```

think: If I run `f()`, what will it return? After running `f()`, what are the values of x, y, z in the global environment?

Answers

```
1 f()
```

```
[1] 6
```

```
1 c(x, y, z)
```

```
[1] 1 2 3
```

The super assignment operators have modified the values of `x`, `y`, and `z` in the global environment to 1, 2, and 3 respectively.

When we call `f()`, it returns 6. It searches for the values of `x`, `y`, and `z` which only exist in the global environment.

Super assignment can be tricky

```
x <- 1; y <- 1; z <- 1
f <- function(){
    y <- 2  # creates y in the current scope (inside f)
    y <<- 4 # modifies y in the higher scope (global)
    g <- function(){
      z <<- 3  # modifies z in the scope higher up f,
               # but does not find z, so goes to the higher scope (
      return(x + y + z) # searches for y in the current scope.
               # does not find y inside g, but does in f. Uses that
      }
    return(g())
}
```

think: If I run `f()`, what will it return? After running `f()`, what are the values of x, y, z in the global environment?

Answers

```
f() # because x = 1, y = 2 (in the scope of f), and z = 3
```

```
[1] 6
```

```
c(x, y, z) # the values of x y and z in the global environment
```

```
[1] 1 4 3
```

More super assignment

```
x <- 1; y <- 1; z <- 1
f <- function(){
    y <- 2
    z <- 10
    y <<- 4 # modifies y in the higher scope (global)
    g <- function(){
      z <<- 3 # modifies z in the scope higher up: f
              # does not touch the z in the global environment
      return(x + y + z) # uses the scoping rules when searching
      }
    return(g())
}
```

think: If I run `f()`, what will it return? After running `f()`, what are the values of x, y, z in the global environment?

Answers

```
1 f()
```

```
[1] 6
```

```
1 c(x, y, z)
```

```
[1] 1 4 1
```

Avoid Super Assignment

You should avoid using super assignment.

Let's say you want to update some object `foo` by appending the value of x-squared to it

```
# BAD:
add_sq_bad <- function(foo, x){
  foo <<- c(foo, x ^ 2) # combines foo with x^2 and super assigns b
}

foo <- 2
add_sq_bad(foo, 5)
foo # so this seemed to work
```

```
[1]  2 25
```

Dangers of super assignment

```
# Let's try the function with a different object, bar

bar <- 10
add_sq_bad(bar, 6)
```

What is the value of `bar` in the global environment? What is the value of `foo` in the global environment?

Dangers of super assignment

```
bar # bar in unchaged
```

```
[1] 10
```

```
foo  # foo changed. probably not what you wanted
```

```
[1] 10 36
```

A better way to change values in the global environment

```
add_sq_good <- function(baz, x){
  c(baz, x^2)
}

foo <- 2
foo <- add_sq_good(foo, 5)  # the assignment of the output to the c
foo
```

```
[1]  2 25
```

```
bar <- 10
bar <- add_sq_good(bar, 5)
bar
```

```
[1] 10 25
```