asitarrives

This WordPress.com site is the cat's pajamas

Understanding Lexical Scoping in R – Great guidance by community TA in Coursera

October 18, 2014 computer science, learning data science

Gregory D. HorneCommunity TA· 5 days ago

I will show you the output from the sample functions to illustrate their behaviour. From the output you should be able to discern the logic within the R code. The functions we are expected to implement to handle invertible matrices follows the exact same logic. The sample run show below can be adapted to the new functions for matrices and the behaviour will be the same.

```
> a <- makeVector(c(1,2,3,4))
> a$get()
[1] 1 2 3 4
> a$getmean()
NULL
> cachemean(a)
[1] 2.5
> a$getmean()  # this is only to show you that the mean has been stored and does not affect anything
[1] 2.5
> cachemean(a)
getting cached data
[1] 2.5
> a$set(c(10,20,30,40))
> a$getmean()
NULL
> cachemean(a)
[1] 25
> cachemean(a)
getting cached data
[1] 25
> a$get()
[1] 10 20 30 40
> a$setmean(0)  # do NOT call setmean() directly despite it being accessible for the reason you will see next
> a$getmean()
[1] 0     # obviously non-sense since…
> a$get()
[1] 10 20 30 40
>cachemean(a)
[1] 0    # as you can see the call to setmean() effectively corrupted the functioning of the code
```

```
> a <- makeVector(c(5, 25, 125, 625))
> a$get()
[1] 5 25 125 625
> cachemean(a)
[1] 195
> cachemean(a)
getting cached data
[1] 195
```

Understanding the concepts in terms of the behaviour of the two functions takes more than a few minutes because you have to not only read the existing exemplar functions but take some time to play with the functions. Often seeing the code in action is more helpful than reading a chapter about lexical scoping. Variable m is declared uniquely in both functions and are allocated separate addresses in memory. In function makeVector() you'll notice variable m is declared immediately and assigned the value NULL using the standard assignment operator (<-). However, the "set" functions defined within the containing makeVector() function require the special assignment operator (<–) to update the value of variable m; it is important to remember variable m was declared and initialised by makeVector(). Had functions set() and setmean() not used the special assignment operator, these functions would have allocated memory to store the value and labelled the address as m. The variables named m would effectively be isolated and distinct variables.

<- Operator Versus <<- Operator

assignment operator: <-
superassignment operator: <<-

```
crazy <- function() {
 x <<- 3.14              # variable x in the containing environment (global in this case) is updated to be
3.14
 print(x)                # since no local variable 'x' exists within function 'crazy' R searches the
containing environments
{ print(x);              # this is to demonstrate the function, not a code block, is the smallest
environment in R
  x <- 42; print(x)      # local variable 'x' is declared (created) and assigned the value 42; overrides the
variable 'x' in
}                        # the containing environment
 print(x)                # since local variable 'x' now exists within the function there is no need to search
the containing
}                        # environment (global in this case)
> x <- 0
> crazy()
3.14
3.14
42
42
> x   # variable 'x' outside of the function its updated value after the first statement within function
'crazy()'
[1] 3.14
```

The first two print() statements use the variable 'x' in the containing environment, as no local variable 'x' exists at the moment, which has been updated from x <- 0 to x <- 3.14 via x <<- 3.14 inside function 'crazy()'.

The third print() statement uses the variable 'x' just created by the preceding assignment statement x <- 42 which causes the containing environment not to be searched unlike the first and second print()

statements.

The fourth print() statement uses the variable 'x' which exists within the function because the x <- 42 now masks access, at least for anything other than the super-assignment operator, to the containing environment's variable 'x'.

I added a call to variable 'x' after the function 'crazy()' returns to show it keeps the new value assigned to it by the super-assignment operator inside function 'crazy()'.

The super-assignment operator does not update a variable of the same name inside an inner function but the innermost environment inherits any changes unless a local variable of the same name exists within the inner function as demonstrated by x <- 42; print(x) and print(x).

Furthermore, if a variable named 'x' had existed inside function 'crazy()' and preceded the call to the super-assignment operator, the results would be as shown in the next example.

```
crazy <- function() {
 x <- 42
 x <<- 3.14
 print(x)
}
>  x <- 0
> x
[1] 0
> crazy()
42
> x
[1] 3.14
```

To reiterate the concept the next section explains in more detail the role and behaviour of the "superassignment" operator which allows the programmer to modify a variable declared outside of the current function in which the reference to the variable is made.

In the simplest example consider how variable 'x' changes when the crazy() function is called.

```
# Declare and define a function named crazy()
crazy <- function() {          # create a new environment with a local variable 'x' and access to another variable 'x'
                               # declared somewhere outside this function
 x <- 3.14                     # assign the numeric value 3.14 to local variable 'x'
 print(x)                      # output the current value of local variable 'x' (1)
  { print(x);                  # output the current value of local variable 'x' (2)
    x <<- 42;                  # assign the numeric value 42 to variable 'x' declared outside this function (3)
    print(x)                   # output the current value of local variable 'x' (4)
  }
 print(x)                      # output the current value of local variable 'x' (5)
}
> x <- 0                      # Declare and define a local variable named 'x'
> x                           # output the current value of local variable 'x'
0
> crazy()                     # Call function crazy()
3.14                          # (1) inner variable 'x'
3.14                          # (2) inner variable 'x'
3.14                          # (4) inner variable 'x'
3.14                          # (5) inner variable 'x'
> x                           # (3) containing environment variable 'x'
42
```

The curly braces (brackets) are intended to highlight the fact in R the smallest unit of lexical scoping

is the function. Unlike some programming languages such as C that allow block-level variable scope, R treats a block enclosed in brackets as part of the nearest function.

```
x <- 3.14
{ x <<- 42 }
```

is treated as though it is as shown below.

```
x <- 3.24  # assigns the value 3.14 to local variable 'x' not the variable 'x' in the containing environment
x <<- 42  # assigns the value 42 to variable 'x' in the containing environment
```

Perhaps the crude graphic can illuminate the effects of lexical scoping on variables a better than mere words.

```
+————————————————+
| (1a) x <- 0  becomes 42 |  outer function / by default the global lexical scope is an anonymous function
|       (1b) x <- 42 by      |
|           (3b) x <<- 42    |
| +————————————+ |
| |   (2) x <- 3.14        | |  inner function
| |   (3a) x <<- 42 does | |
| |          not affect     | |
| |          (2)            | |
| +————————————+ |
+————————————————+
```

Flow of execution is (1a) -> [(2) & (3a)] -> [(3b) & (1b)] -> (1a). I used the labels "inner function" and "outer function" because the same rules apply to nested functions.

Unit tests (with expected output) for Programming Assignment 2

```
### Example
>  source("cachematrix.R")
>  amatrix = makeCacheMatrix(matrix(c(1,2,3,4), nrow=2, ncol=2))
>  amatrix$get()       # Returns original matrix
   [,1] [,2]
[1,]   1   3
[2,]   2   4
 >  cacheSolve(amatrix)  # Computes, caches, and returns   matrix inverse
   [,1] [,2]
[1,]  -2  1.5
[2,]   1 -0.5
 > amatrix$getinverse()  # Returns matrix inverse
   [,1] [,2]
[1,]  -2  1.5
[2,]   1 -0.5
 > cacheSolve(amatrix)  # Returns cached matrix inverse using previously computed matrix inverse
getting cached data
   [,1] [,2]
[1,]  -2  1.5
[2,]   1 -0.5
>  amatrix$set(matrix(c(0,5,99,66), nrow=2, ncol=2)) # Modify existing matrix
>  cacheSolve(amatrix)  # Computes, caches, and returns new matrix inverse
        [,1] [,2]
[1,] -0.13333333  0.2
[2,]  0.01010101  0.0
>  amatrix$get()       # Returns matrix
```

```
     [,1] [,2]
[1,]    0   99
[2,]    5   66
>   amatrix$getinverse()  # Returns matrix inverse
          [,1] [,2]
[1,] -0.13333333  0.2
[2,]  0.01010101  0.0
```


Share this: