R Programming Assignment 2: Lexical Scoping

Mihai M. Craiu

2/7/2020

Introduction

This second programming assignment will require you to write an R function is able to cache potentially time-consuming computations. For example, taking the mean of a numeric vector is typically a fast operation. However, for a very long vector, it may take too long to compute the mean, especially if it has to be computed repeatedly (e.g. in a loop). If the contents of a vector are not changing, it may make sense to cache the value of the mean so that when we need it again, it can be looked up in the cache rather than recomputed. In this Programming Assignment will take advantage of the scoping rules of the R language and how they can be manipulated to preserve state inside of an R object.

Example: Caching the Mean of a Vector

In this example we introduce the «- operator which can be used to assign a value to an object in an environment that is different from the current environment. Below are two functions that are used to create a special object that stores a numeric vector and caches its mean.

The first function, makeVector creates a special "vector", which is really a list containing a function to

- set the value of the vector
- get the value of the vector
- set the value of the mean
- get the value of the mean

```
makeVector <- function(x = numeric()) {
        m <- NULL
        set <- function(y) {
                x <<- y
                m <<- NULL
        }
        get <- function() x
        setmean <- function(mean) m <<- mean
        getmean <- function() m
        list(set = set, get = get,
             setmean = setmean,
             getmean = getmean)
}
```

The following function calculates the mean of the special "vector" created with the above function. However, it first checks to see if the mean has already been calculated. If so, it gets the mean from the cache and skips the computation. Otherwise, it calculates the mean of the data and sets the value of the mean in the cache via the setmean function.

```r
cachemean <- function(x, ...) {
        m <- x$getmean()
        if(!is.null(m)) {
                message("getting cached data")
                return(m)
        }
        data <- x$get()
        m <- mean(data, ...)
        x$setmean(m)
        m
}
```

Assignment: Caching the Inverse of a Matrix

Matrix inversion is usually a costly computation and there may be some benefit to caching the inverse of a matrix rather than compute it repeatedly (there are also alternatives to matrix inversion that we will not discuss here). Your assignment is to write a pair of functions that cache the inverse of a matrix.

Write the following functions:

- makeCacheMatrix: This function creates a special "matrix" object that can cache its inverse.
- cacheSolve: This function computes the inverse of the special "matrix" returned by makeCacheMatrix above. If the inverse has already been calculated (and the matrix has not changed), then the cacheSolve should retrieve the inverse from the cache. Computing the inverse of a square matrix can be done with the solve function in R. For example, if X is a square invertible matrix, then solve(X) returns its inverse.

For this assignment, assume that the matrix supplied is always invertible

My Solution

```r
## Caching the Inverse of a Matrix:
## Matrix inversion is usually a costly computation and there may be some
## benefit to caching the inverse of a matrix rather than compute it repeatedly.
## Below are a pair of functions that are used to create a special object that
## stores a matrix and caches its inverse.

## This function creates a special "matrix" object that can cache its inverse.

makeCacheMatrix <- function(x = matrix()) {
        ##Initialize the inverse property
        inv <- NULL
        ## Method to set the matrix
        set <- function(y) {
                x <<- y
                inv <<- NULL
        }
        ##Method to get the matrix
        get <- function() x
        ##Method to set the inverse of the matrix
        setInverse <- function(inverse) inv <<- inverse
        ##Method to get the inverso of the matrix
```

```
        getInverse <- function() inv
        ##Return a list of the methods
        list(set = set,
             get = get,
             setInverse = setInverse,
             getInverse = getInverse)
}


## This function computes the inverse of the special "matrix" created by
## makeCacheMatrix above. If the inverse has already been calculated (and the
## matrix has not changed), then it should retrieve the inverse from the cache.


cacheSolve <- function(x, ...) {
        ## Return a matrix that is the inverse of 'x'
        inv <- x$getInverse()
        ##Just run the inverse if its alredy set
        if (!is.null(inv)) {
                message("getting cached data")
                return(inv)
        }
        ##Get the matrix from our object
        mat <- x$get()
        ##Calculate the inverse using the matrix multiplication
        inv <- solve(mat, ...)
        ##Set the inverse to the object
        x$setInverse(inv)
        ##Return the matrix
        inv
}
```

Testing My Functions

```
my_matrix <- makeCacheMatrix(matrix(1:4, 2, 2))
my_matrix$get()
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
my_matrix$getInverse()
```

```
## NULL
```

```
cacheSolve(my_matrix)
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```r
cacheSolve(my_matrix)
```

```
## getting cached data
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```r
my_matrix$getInverse()
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```r
my_matrix$set(matrix(c(2, 2, 1, 4), 2, 2))
my_matrix$get()
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    2    4
```

```r
my_matrix$getInverse()
```

```
## NULL
```

```r
cacheSolve(my_matrix)
```

```
##            [,1]       [,2]
## [1,]  0.6666667 -0.1666667
## [2,] -0.3333333  0.3333333
```

```r
cacheSolve(my_matrix)
```

```
## getting cached data
```

```
##            [,1]       [,2]
## [1,]  0.6666667 -0.1666667
## [2,] -0.3333333  0.3333333
```

```r
my_matrix$getInverse()
```

```
##            [,1]       [,2]
## [1,]  0.6666667 -0.1666667
## [2,] -0.3333333  0.3333333
```

Another Useful Example

A good example for getting a better understanding of differences amount formal parameters, local variables and free variables. And it is also helpful for learning the «- operator.

```r
open.account <- function(total) {
        list(
                deposit = function(amount) {
                        if(amount <= 0)
                                stop("Deposits must be positive!\n")
                        total <<- total + amount
                        cat(amount, "deposited.  Your balance is", total, "\n\n")
                },
                withdraw = function(amount) {
                        if(amount > total)
                                stop("You don't have that much money!\n")
                        total <<- total - amount
                        cat(amount, "withdrawn.  Your balance is", total, "\n\n")
                },
                balance = function() {
                        cat("Your balance is", total, "\n\n")
                }
        )
}
```

Simple tests for open.account function:

```r
peter <- open.account(100)

mm <- open.account(200)

peter$withdraw(30
                )
```

```
## 30 withdrawn.  Your balance is 70
```

```r
peter$balance()
```

```
## Your balance is 70
```

```r
mm$balance()
```

```
## Your balance is 200
```

```r
peter$deposit(50)
```

```
## 50 deposited.  Your balance is 120
```

```r
peter$balance()
```

```
## Your balance is 120
```

```r
mm$balance()
```

```
## Your balance is 200
```

```r
mm$withdraw(12)
```

```
## 12 withdrawn.  Your balance is 188
```

```r
mm$balance()
```

```
## Your balance is 188
```

```r
peter$balance()
```

```
## Your balance is 120
```

```r
##peter$withdraw(500)
##Error in peter$withdraw(500) : You don't have that much money!
```