

Programming Assignment 2: Lexical Scoping

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 cache's its mean.

The first function, `makeVector` creates a special “vector”, which is really a list containing a function to:

1. set the value of the vector
2. get the value of the vector
3. set the value of the mean
4. 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.

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

```

    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.

```

## 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 two functions are used to cache the inverse of a matrix.

## Assumption - Matrix supplied is always invertible.

## The first function, makeCacheMatrix creates a special "matrix", which is
## really a list containing a function to:
## 1. set the value of the matrix
## 2. get the value of the matrix
## 3. set the value of the inverse of the matrix
## 4. get the value of the inverse of the matrix

makeCacheMatrix <- function(x = matrix()) {
  m <- NULL
  set <- function(y){
    x <- y
    m <- NULL
  }
  get <- function() x
  setinverse <- function(inverse) m <- inverse
  getinverse <- function() m
  list(set = set, get = get,
       setinverse = setinverse,
       getinverse = getinverse)
}

## 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.

```

```

cacheSolve <- function(x, ...) {
  m <- x$getinverse()
  if(!is.null(m)) {
    message("getting cached data")
    return(m)
  }
  data <- x$get()
  m <- solve(data, ...)
  x$setinverse(m)
  m
}

```

```

## Example
x <- matrix(c(1,2,3,4), ncol=2)
z <- makeCacheMatrix(x)
z$get()

```

```

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

```

```

#first run
cacheSolve(z)

```

```

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

```

```

#second run
cacheSolve(z)

```

```

## getting cached data

```

```

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

```