Announcements

WEEK-BY-WEEK

Week 1: Getting Started and R Nuts and Bolts

Week 2: Programming with R

Week 3: Loop Functions and Debugging

Week 4: Simulation and Profiling

LECTURES

Lecture Videos

Lecture Slides (link to GitHub)

EXERCISES

Quizzes

Programming Assignments 1, 3, & swirl

Programming Assignment 2

ABOUT THE COURSE

Syllabus

About the Instructor

COMMUNITY

DSS Community Site

Discussion Forums

Course Wiki

Help Articles

Forums / Programming Assignment 2 (peer assessment): Lexical Scoping

Explanation of the example functions for newbies!

Subscribe for email updates.

No tags yet. + Add Tag

Sort replies by: Oldest first Newest first Most popular

Hussain Boxwala · 5 days ago

Hi All,
As a newbie to R programming myself, I found it very hard to understand each and every line of the example functions and at this stage I think it is absolutely necessary to understand each and every line as the functions are fairly simple compared to what we would encounter in coming courses of the specialization.

So I have prepared this post of what I have understood after reading the documentations and some googling. Hope you find it useful.

Let us examine one function at a time, first the makeVector function.

I have added a few lines to the original example to understand the environments concept.

```
> makeVector <- function(x = numeric()) {
        m <- NULL
        print(environment())
        evn <- environment()
        print(parent.env(evn))
        set <- function(y) {
                x <<- y
                m <<- NULL
        }
        get <- function() x
        setmean <- function(mean) m <<- mean
        getmean <- function() m
        getevn<- function() environment()
        list(set = set, get = get,
             setmean = setmean,
             getmean = getmean,
             getevn = getevn)
 }
```

now lets assign this function to a variable vec

```
> x <- 1:10000
> vec<-makeVector(x)
<environment: 0x2924c28>
<environment: R_GlobalEnv>
```

We can see the two environments being printed from our print statements. We will look into them in some time.
As we know the last statement of a function is returned, here "vec" will contain a list of the functions defined in makeVector as the "list(set = ...)" is our last statement.
Now let us see how the mean is cached. The cached mean is nothing but the variable "m", and we can access it through the functions getmean() and setmean(). Let us see some examples.

```
> vec$getmean()
NULL
> mx <- mean(x)
> vec$setmean(mx)
> vec$getmean()
[1] 5000.5
```

The above code is fairly straightforward. But how do we see the value of "m" i.e. where the mean is stored by the setmean() function. We want to access it directly without the getmean() function. Here is where the environments come in.

```
> vec$getevn()
<environment: 0x353b260>
> ls(vec$getevn())
character(0)
```

The environment returned above is the environment inside the function getenv(), beacuse the statement returning the environment is defined inside this function, but m is defined in the environment immediately outside i.e. the parent environment. So lets try this:

```
> parent.env(vec$getevn())
<environment: 0x3322ab8>
> ls(parent.env(vec$getevn()))
[1] "evn"     "get"     "getevn"  "getmean" "m"       "set"     "setmean"
[8] "x"
```

As we can see "m" is defined in this environment and has the cached value.
We can see this by typing:

```
> parent.env(vec$getevn())$m
[1] 5000.5
```

See! The cachemean() function does nothing but the same things that we did manually. Go ahead and have a look at the cachemean() function and it should be very clear to you now.
Hope this helps all the newbies like me who want to understand each and every line of code from the examples.

↑ 35 ↓ · flag

 Paul T Mielke Signature Track · 4 days ago

Hussain,

Thank you very much for taking the trouble to explain this so clearly! Your post really helped me understand what is going on here.

I know C++ a bit, so I was trying to map the R concept of object orientation to the way C++ works. The mapping is not very clean. I was trying to think of "makeVector" as the definition of the base class and cachemean as a subclass of that. It's a little more twisted than that. cachemean is just a function that operates on makeVector objects. A cleaner way to implement this concept would be to put all the functionality in the makeVector class. The base getmean and setmean "class methods" could do the mean calculations down in makeVector. I.e. to think of the makeVector class as a "automatic mean caching vector" object, rather than splitting the actual mean calculation out into a separate function.

Maybe I'll get motivated enough to implement it the way I described above and post that.

Thanks again for your clear explanations!

Regards,
Paul

↑ 0 ↓ · flag

 Paul T Mielke Signature Track · 4 days ago

Oh, ok. I should have known. As soon as I tried to implement my strategy of burying the mean calculation down inside the makeVector$getmean() "class method", I see why that doesn't work. The problem is that there is no clean way to pass more arguments to the underlying mean() function. Consider the case that your base vector has some NAs in it. Then you can say:

xVec <- makeVector(x)
xmean <- cachemean(xVec, na.rm = TRUE)

But note that the way this works is basically broken, in that once the value is cached, specifying different arguments has no effect. Consider this sequence in the case that the original vector has NAs in it:

xVec <- makeVector(x)
cachemean(xVec)
NA
cachemean(xVec, na.rm = TRUE)
getting cached data
NA

You really need to clear the cache if any of the arguments to cachemean() are different. I guess you'd have to store the argument list as a "private" data list like the cached "m" value and compare. Seems pretty gross, but I guess this is just to teach us how this stuff works.

Regards,
Paul

↑ 0 ↓ · flag

+ Comment

Hussain Boxwala · 4 days ago

Paul,
I'm glad you found this post helpful. Yes you are right that we need to clear the cached value to get a different value, because the example functions are quite trivial to just explain the concept. The way I look at the environments is that of one sphere inside the other and that any sphere and its members can be accessed from any other sphere that you may be in, with the appropriate command.

↑ 0 ↓ · flag

MD. ABU YUSUF ANSARI · 7 hours ago

Mr. Hussain Boxwala can you please help me to understand these things?

```
1. f <- function(){
       g <- function() Do something
}
```

>> what is the parent & global environment of g & parent & global environment of f ?

```
2. set <- function(y) {
               x <<- y
               m <<- NULL
       }
```

>> Which one is the free variable in this nested function? [i suspect y is a formal argument]

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

>> Can you or anyone here explain this piece of code line by line. I will pray for you?

>> The whole thing is looking absurd to me.
>> Thanks in advance.

-1 · flag

+ Comment

Karthik Narayanan Venkatesh Signature Track · 3 days ago

Thanks for details.
Another way to look into the object created. The output is list of objects created.

> x <- 1:4
>x1<- makeVector(x)
>x1
function (y)
{
  x <<- y
  m <<- NULL
}
<environment: 0x00000000099a4c80>

$get
function ()
x
<environment: 0x00000000099a4c80>

$setmean
function (mean)
m <<- mean
<environment: 0x00000000099a4c80>

$getmean
function ()
m
<environment: 0x00000000099a4c80>

0 · flag

+ Comment

Jinshu Fang Signature Track · 3 days ago

This gives a good start to understand the sample for assignment2 !
Thank you!

0 · flag

+ Comment

Ying Jiang Signature Track · 3 days ago

Hi Hussain (and Paul),

Thanks for the detailed post. It helped a lot. Several things I still don't understand with regards to the cachemean() function:

1. Is the input to cachemean() a vector? The example given by Paul seems to suggest that the input is a function (xVec is a list of functions).
2. If cachemean(x) takes a vector x instead of a function, then x$getmean() is invalid. Is that correct?

Regards,
Ying

0 · flag

Paul T Mielke Signature Track · 3 days ago

Hi, Ying

I had the same question until I read Hussain's post and then did some experiments.

1) The input parameter to cachemean is one of the "function list" objects created when you call makeVector. The argument to makeVector is a plain ordinary R numeric vector, but the return value of makeVector is a different thing entirely.
2) If you try passing a plain vector to cachemean, it throws an error since the expression x$getmean is undefined.

This would be a lot more obvious in a strongly typed language. In c or c++ you have to declare the types of your formal parameters, so this type of question would be answered by the function declaration.

Regards,
Paul

0 · flag

Ying Jiang Signature Track · 2 days ago

Hi Paul,

Thank you. I understand that makeVector() returns a "function list". It's clear that cachemean() needs to take a function, not a plain vector. As such, I would appreciate clarification on the assignment's guidance code:

```
cacheSolve <- function(x, ...) {
  ## Return a matrix that is the inverse of 'x'
```

Does the variable x here refer to an actual matrix? Or is it implied that it's not an actual matrix? Would appreciate enlightenment on linguistic conventions as I'm completely new to programming.

Regards,
Ying

0 · flag

Paul T Mielke Signature Track · 2 days ago

Hi, Ying.

That's a good point about the wording in that comment. I agree that it is misleading. Of course I can't speak for Prof Peng, but my guess is that the wording is just sloppy. Maybe he even said it that way intentionally to force us to dig deeper and do the work to understand what he really meant.

We all agree that cachemean takes a list of functions as its argument. I implemented the cacheSolve case as an exact analog of the way cachemean works, taking the return value of makeCacheMatrix as its argument. So cacheSolve returns the inverse of the matrix that is returned by x$get() of course.

I've been writing software for a long time, but am new to R and to the data science area. The one thing I can say about this in general is that computer code needs to be taken extremely literally (since the computer will), but specifications for computer code "not so much".

Paul

0 · flag

Hussain Boxwala · 2 days ago

Hi Ying,

I think the confusion is because the parameters for both the functions have been named 'x':

```
## Write a short comment describing this function

makeCacheMatrix <- function(x = matrix()) {

}


## Write a short comment describing this function

cacheSolve <- function(x, ...) {
        ## Return a matrix that is the inverse of 'x'
```

The 'x' in the statement " ## Return a matrix that is the inverse of 'x' " refers to the 'x' of makeCacheMatrix() and not cachesolve(). So the cachesolve() should return the inverse of the matrix which is passed as an argument to the makeCacheMatrix().

Hope this makes it clear.

↑ 1 ↓ · flag

+ Comment

J.R. Jasperson Signature Track · 2 days ago

I spent a lot of time unwinding this, put on the right track by the original/excellent post by Hussein. A few notes that folks may find helpful; note I reiterate a few concepts Hussein (and/or others) introduced in the flow of my explanation:
-The two functions in the example are meant to work together, really cachemean() is just a "wrapper" to makeVector.

-Usage would be something like this:

```
> myVector <- makeVector(1:10)
> cachemean(myVector)
[1] 5.5
>cachemean(myVector)
getting cached data
[1] 5.5
```

-When makeVector is first called, it creates a unique environment (basically a namespace for variables). This environment persists (stays in memory) because there is a pointer to the instantiation of the function (myVector in the example above). As long as that pointer exists the environment stays in memory.

-If you were to call makeVector again, it would get a new/different environment/namespace.

-The list command at the end of the function returns a named list with the variables (which point to the get/set functions) as values. Thus the names (characters before the equal signs) are arbitrary and are set to the same as the function names just to avoid any confusion. myVector[[2]]() is the same as myVector$get().

-There is nothing magical about this list or unique to the values being functions. The call to makeVector returns a list, indexed by names for convenience, of pointers to (in this case) the functions defined therein. You *could* just as easily add m = m to directly expose the mean, but this kind of defeats some of the point of the example. More on this below...

-If you examine myVector (myVector) you will see it contains a named list of the functions which also prints out the enclosing environment they were defined in (hex number 0X and 15 alpha/numerics, i.e. 0x0000000008bdf400).

-Just printing myVector does not tell the whole story though. Try:

```
ls(environment(myVector$get))
[1] "get"   "getmean" "m"   "set" "setmean" "x"
```

Here we not only see the exposed functions, but also the variable "m" (mean) and "x" (original numeric vector passed as the param) hiding in memory. This is the actual location where the mean is cached - inside the object named m, inside the makeVector's environment (again, this environment is unique to *this* instance of the makeVector call). The mean can just as easily be fetched by calling:

```
> myVector$getmean()
```

```
[1] 5.5
> # --OR--
> environment(myVector$get)$m
[1] 5.5
```

-The environment listed for the functions inside myVector is the environment in which they were defined (i.e. the same as the makeVector environment which persists thanks to the assignment of the myVector object). However, each time these functions (get/set etc.) *run* they spin up their own unique environment which is destroyed upon the end of the execution of the function. This is why you see the "<<-" assignment operator. Any object assigned with <- would be destroyed as soon as the function exited.

-"m" is set to NULL at the top of makeVector function because of the curious way that <<- works. <<- "cause[s] a search to made through parent environments for an existing definition of the variable being assigned. If such a variable is found (and its binding is not locked) then its value is redefined, otherwise assignment takes place in the global environment."[0]. So <<- first looks in its "parent environment", which is to say the environment which *this* instantiation of makeVector points to. If "m" was not initialized/did not exist in this instance of makeVector's environment, it would get popped into the R_GlobalEnv (i.e. the interactive space). You can test this by commenting out the m <- NULL line, running setmean, and then notice that "m" is available at the command line. That would not be very desirable because subsequent calls to setmean() would clobber one another.
Hope this understanding helps you with assignment 2, happy to answer any questions.

Cited:
[0] http://stat.ethz.ch/R-manual/R-devel/library/base/html/assignOps.html
Other material I used to figure this out:
[1] http://adv-r.had.co.nz/Environments.html
[2] http://adv-r.had.co.nz/Functions.html

16 · flag

 Paul T Mielke Signature Track · 2 days ago

JR, Thanks! You have really covered the topic. Worthy of many upvotes!

0 · flag

J.R. Jasperson Signature Track · 2 days ago

Thanks Paul, that is kind.

I'm new to R, but not new to programming. It seems pretty clear to me that the point of this exercise is more about understanding functional environments and how to work in them than caching an expensive result (which could easily be done with a single object assignment in a far less convoluted way). This may be sadistic but I actually kind of like that the instructors gave just the slightest bit of bread crumbs and expect the students to figure out the rest.

It was easy enough to simply "transmute" the examples into working code for the assignment, but I suspect if you don't take the time to understand the "why" now you'll get buried later. With the nudge in the right direction from the OP, some additional reading and testing I was finally able to understand every bit of it - hopefully the posts in this thread can help some other folks stay on track.

0 · flag

Hussain Boxwala · 2 days ago

Thanks JR Jasperson, your post covers whatever explanation was left to be covered in this topic.

0 · flag

+ Comment

Karthik Narayanan Venkatesh Signature Track · 2 days ago

Hi JR,

I tried you command i.e. ls(environment(a$get()), but it looks like 'm' is not present in the environment and I am able to access it

from global environment.

Is there anything I am missing out here?

↑ 0 ↓ · flag

> J.R. Jasperson Signature Track · 2 days ago
>
> In this case you don't want the parentheses () after a$get.
>
> Assuming you are running the example functions, and "a" is an instance of makeVector (a <- makeVector(someVector)) then:
> ls(environment(a$get))
> should work for you.
>
> The reason for no parenthesis (as you would normally see) after the a$get function name is because we are trying to get the enclosing environment of the a$get function itself, not the environment of the *output* of the get function (a$get()).
>
> Hope that makes sense, give it a try and let me know if you're still stuck/have questions.
>
> ↑ 1 ↓ · flag

+ Comment

 Guilherme Pedrosa · 2 days ago

Hussain and J. R.,

Thank you for your thorough explanations, they were of great value to me. Time to put hands on the console once more so everything sinks in!

Cheers

↑ 0 ↓ · flag

+ Comment

 Danny Clarke Signature Track · 2 days ago

I, too, wish to thank everyone on this thread for explanations of the programming assignment. After reading the explanations, it struck me that the make* functions are just another way of defining a class and instantiating it. The inner functions (set, get, etc.) are methods that can be invoked when one is working with a class instance. Am I wrong in thinking this? The cache* functions are merely ways of avoiding repetitive computation, right? Any corrections or further insights would be much appreciated.

↑ 0 ↓ · flag

> Hussain Boxwala · 2 days ago
>
> I don't know much about OOP but the environment can be visualized as a public class with all its methods and variables also defined as public so that if we wanted , we could access the variable "m" directly without the getmean() and setmean() functions. Let me know if this would be a good way of relating it to OOP.
>
> ↑ 0 ↓ · flag
>
> > J.R. Jasperson Signature Track · 2 days ago
> >
> > Danny -
> >
> > What you said is spot on. It doesn't seem like R has the native OOP functionality and semantics that an OO programmer would be used to, but the approach in the example reasonably emulates OO.
> >
> > makeFunction does somewhat act like a class, and the methods within somewhat act like object methods. Instead of public or private declarations, one can expose the names (variables) or functions (methods) via the final list() command of makeVector to emulate 'public' and make it easier to get at what is desired inside the function (in this case the four get/set

functions). As I demonstrated above other names (varaibles, functions) can still be reached via environment() calls so it does not *appear* (at this point anyway) that a true equivalent to "private" exists.

↑ 1 ↓ · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our forum posting policies before posting.

B | I | Link | <code> | Pic | Math | Edit: Rich | Preview

Make this post anonymous to other students

Subscribe to this thread at the same time

Add post