08 APRIL 2013

Package-Wide Variables/Cache in R Packages

It's often beneficial to have a variable shared between all the functions in an R package. One obvious example would be the maintenance of a package-wide cache for all of your functions. I've encountered this situation multiple times and always forget at least one important step in the process, so I thought I'd document it here for myself and anyone else who might encounter this issue. I setup a simple project on GitHub to demonstrate the various attempts you may take to solve the problem and, ultimately, a solution: https://github.com/trestletech/RCache (https://github.com/trestletech/RCache). The rest of this article will presume some knowledge of authoring R packages. If that's not you, check out RStudio's guide to authoring R packages (http://www.rstudio.com/ide/docs/packages/overview).

The fundamental problem comes down to R's management of *environments*. (I'll introduce the basics here, but for a thorough discussion on the topic, be sure to check out Hadley Wickham's writings (https://github.com/hadley/devtools/wiki/Environments) on the topic.) An environment is responsible for mapping data to named objects. When I execute

```
> a <- "hello"
> a
[1] "hello"
```

there was an environment responsible for initially associating the length-one character vector containing the text `hello` with the variable named `a`. Later, when I go to reference a variable by name, an environment is responsible for retrieving the data I had

associated with this variable. Functions have their own environments in which they run, which is why you observe behavior like:

```
> x <- 1
> foo <- function(){
  print (x)
  x <- 2
  print(x)
  x
}
> x
[1] 1
> foo()
[1] 1
[1] 2
[1] 2
> x
[1] 1
```

You can see that the variable named `x`, depending on which environment it's in, can take one of two values in this example. Inside of the `foo` function, the `x` variable initially only existed in the parent environment, so it took on the value of `x` in that environment and was `1`. Later, it was assigned a value of `2` and retained that value when it was returned. Outside of the function, however, `x` was associated with the value `1`, and this binding was maintained despite another variable named `x` being created inside of a function's environment.

This is a powerful feature of R, when properly understood. This allows you to create variables that only exist inside of a function or, more relevant to today's discussion, only within a particular package. So it should be possible to leverage these environments to create a variable named `cache` which is accessible to all the functions in my package, but won't accidentally likely be overwritten or modified by the user or, even more likely, collide with another variable named `cache` used in some other package.

Example Package – rev. a804 (https://github.com/trestletech/RCache/tree/a804a4eaba9f6eae68232561bc85829156eff4fd)

For the rest of the demonstration, we'll use the example of creating an R package which merely downloads files from the Internet. Of course, it might be appropriate to cache data in such a package so that the same URL won't have to be retrieved remotely multiple times. One simple approach would be to use a `list` to associated character strings (such as URLs) with some data (such as the content of the web page). We can first create a function which will download data without using any cache:

```
#' Download a file.
#'
#' @importFrom httr GET
#' @importFrom httr content
#' @export
download <- function(url){
  content(GET(url))
}
```

Example Package – rev. a81e (https://github.com/trestletech/RCache/tree/a81e48a0d31c7baff18cf6dcc68f9293f9ecd1df)

Creating a variable that is available within all functions of your package is as simple as binding a variable to data outside of any functions in a .R file in your package.

```
cache <- list()
```

One might think we could then put data into that cache variable by assigning named elements within to it from packages that can access it. For instance:

```
download <- function(url){
  if (!is.null(cache[[url]])){
    return(cache[[url]])
  }

  file <- content(GET(url))
  cache[[url]] <- file

  file
}
```

Unfortunately, this, like our first example, is assigning data to a separate variable named `cache` which exists only inside of this function. If you were to build the package and run it, you'd find that the code:

```
down <- download("http://www.gutenberg.org/cache/epub/2500/pg2500.txt")
RCache:::cache
```

would successfully download the book, *Siddhartha*, but the package cache would be empty. (If you're unfamiliar with the `:::` operator, it checks inside of the package — named on the left — for a variable — named on the right and returns it. So we can inspect the `cache` variable from code outside of our package.)

Special Assignment – rev. 5548 (https://github.com/trestletech/RCache/tree/55480862a1f9c21c02b9b68f74c089d8dade5956)

In order to alter a variable created in the parent environment, one must use the *special* assignment operator, `<<-`. This will adjust a binding not in the current environment, but in a parent environment. So we can adjust our line in which we assign a value to the cache variable to use this operator:

```
file <- content(GET(url))
  cache[[url]] <<- file
```

However, if we run this, we'll find that we get an interesting error:

```
down <- download("http://www.gutenberg.org/cache/epub/2500/pg2500.txt")
Error in cache[[url]] <<- file :
  cannot change value of locked binding for 'cache'
```

What's the deal? R has a concept of "Locked Bindings" which allow you to ~~forbid~~ *discourage* changes in variables by locking either a particular variable binding or an entire environment. In this case, the `cache` binding has been locked and can't be altered by constituent functions. So we'll need to take a different approach altogether.

Environments – rev. 320e (https://github.com/trestletech/RCache/tree/320e3df3b9078a3d0ad81286a68f9a2e83de93a2)

It seems we can properly access a package-wide variable from within a function of a package, but we're not allowed to overwrite it (or create new variables at that level from within a function). Perhaps we could leverage environments to solve this problem. As it turns out, environments were likely a cleaner solution to our problem all along. Instead of creating a cache `list`, we can create it a cache `environment`:

```
cacheEnv <- new.env()
```

As long as this environment is created in one of our package's `.R` files and not inside of a function, it will be accessible across our entire package. We can do all the things with this environment that we're used to doing in our regular R environment (whether we knew we were using environments or not): create new variables (`assign`), modify existing variables (`assign`), remove variables (`rm`), retrieve data associated with a variable (`get`), list the variables in an environment (`ls`), etc.

All of the functions mentioned above accept an `envir` argument which specifies in which environment you'd like to perform the operation. The default is your current environment, but you could just as easily point these functions at your new environment to do something like `assign(url, file, envir=cacheEnv)` to assign the value currently stored in the `file` variable to a new variable who's name is the value currently contained in the `url` variable within our cache environment. Then we could use `get(url, envir=cacheEnv)` to get the variable who name matches the current value of the `url` variable in the cache environment.

For instance:

```
> url <- "http://mytext.com"
> file <- "This is the content I downloaded"
> cacheEnv <- new.env()
> assign(url, file, envir=cacheEnv)
> get(url, envir=cacheEnv)
[1] "This is the content I downloaded"
```

Now we can incorporate this into our package by changing the `download` function to use these facilities:

```
download <- function(url){
  if (exists(url, envir=cacheEnv)){
    return(get(url, envir=cacheEnv))
  }

  file <- content(GET(url))
  assign(url, file, envir=cacheEnv)

  file
}
```

And we finally have a working cache. When a URL is requested via our package's `download` function, the data will first be stored in this package's `cacheEnv` environment before being returned. The next time that URL is requested, the `cacheEnv` environment will be checked to see if we already downloaded the content of that URL. Because a variable by that name already `exists` in our `cacheEnv` environment, the value will be pulled from there and returned rather than retrieved remotely.

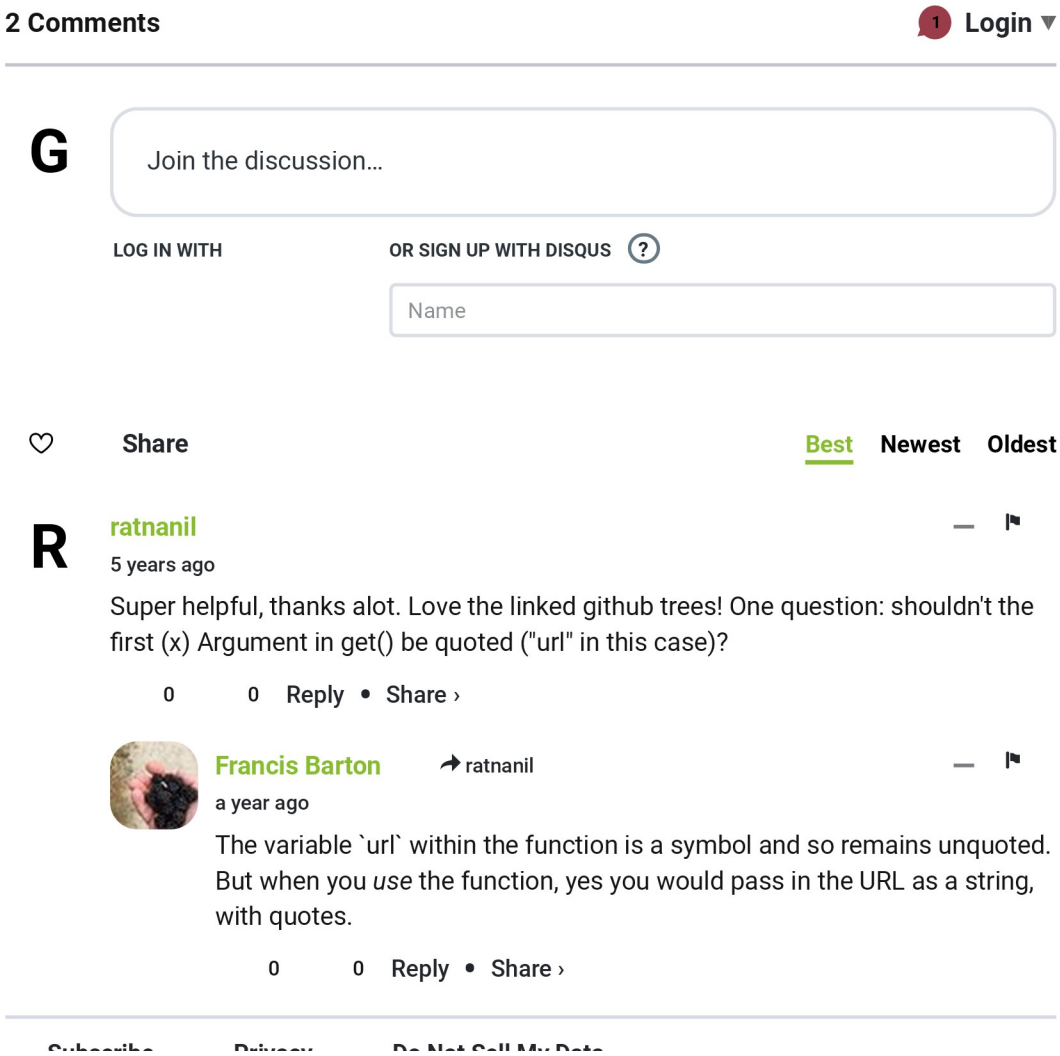
Conclusion

Hopefully you've learned a thing or two about environments in R and how to use them from within R packages. Environments can be very valuable tools for advanced R programmers. They're one of the tried-and-true ways to replicate "pass-by-reference" programming in R (though that's typically unexpected — thus discouraged — behavior for an R object). They also have some unique lookup properties being built on hashes that allow them to more expediently map character strings to data when there is a very large number of possible keys.

Happy packaging!

R (https://trestletech.com/tag/R)

(https://twitter.com
/intent
/tweet?original_refer
//trestletech.com
/2013/04
/package-
wide-
variablescache-
in-
r-package
/&text=Package-
Wide
Variables/Cache
in R
Packages&url=https:
/2013/04
/package-
wide-
variablescache-
in-
r-package/)



‹ Prev (https://trestletech.com/2013/02/graphical-tools-rgl-on-a-headless-shiny-server/)
Next › (https://trestletech.com/2013/06/dallas-r-users-creating-r-packages-this-saturday-622/)

GEORGE DYSON

What if the cost of machines that think is people who don't?

(https://twitter.com/trestlejeff)
(https://www.linkedin.com/company/trestle-technology)