Functions

Paul E. Johnson[1] [2]

[1] Department of Political Science

[2] Center for Research Methods and Data Analysis, University of Kansas

2018

KU CENTER FOR RESEARCH METHODS & DATA ANALYSIS

College of Liberal Arts & Sciences

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

Outline

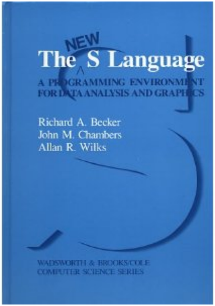
1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

R is comparatively more open

- S started as a programming language for statistical calculations
- The programs S and R (R Core Team, 2017) accept that language
- Because S/R was first a language, it retains many of the programmer-friendly features of a programming language
- In comparison to, for example, SAS or Stata

Generations of S

- The S Language– John Chambers, et al. at Bell Labs, mid 1970s.
- There have been 4 generations of the S language.
- Many packages now were written in S3, but S4 has existed for 10 years.
- New frameworks constantly debated & proposed


NEW
The S Language
A PROGRAMMING ENVIRONMENT FOR DATA ANALYSIS AND GRAPHICS
Richard A. Becker
John M. Chambers
Allan R. Wilks
WADSWORTH & BROOKS/COLE COMPUTER SCIENCE SERIES

S3: *The New S Language* 1988

Is R a Branch from S?

Ross Ihaka and Robert Gentleman. 1996. “R: A language for data analysis and graphics.” *Journal of Computational and Graphical Statistics*, 5(3):299-314.

- R is
  1. a competing dialect of the S language.
  2. a competing software & package management system.

S pioneers now work to advance R.

S4: John Chambers,*Software for Data Analysis: Programming with R*, Springer, 2008

functions

- The R design allows both
  1. inclusion of function collections (packages) prepared by others
  2. easy creation of user functions written during a user's session
- In CRMDA, I notice a pattern.
  1. We work on 1 project, write some functions.
  2. Work on another project, write same/similar functions
  3. We notice the common need, sometimes try to write general purpose functions that
     1. would have worked in past projects
     2. are useful in future projects.
- Many functions in the "rockchalk" package, and all of the function in "kutils", are borne of necessity in that way.

Looking Good is Feeling Good

When your project is finished, I wish your work would look like this

```r
## Functions defined at the top!
myfn1 <- function (arg1, arg2){
   ## lines here using arg1, arg2
 }
myfn2 <- function (arg1, arg2, arg3){
   ## caution: reused arg1, arg2 local varnames
   ## arg1, arg2 different here than in myfn1
 }
## When I check your work, I focus below, not above this line
 a <- 7
 b <- c(4, 4, 4, 4, 2)
 d <- c("New York", "Cincinnati")
 result1 <- myfn1(a, b)
 result2 <- myfn2(result1, d)
```

In a perfect world

- Each function would carry out an understandable purpose that we can believe is done correctly
- After we verify `myfn1` and `myfn2`, we'd never "read through" them again, they are no longer part of the proof-reading exercise. There may be "troubleshooting", but we expect those functions to work dependably.
- Some "art" and "judgment" is needed, to make a function work correctly, with just the right inputs.
  - Novice error: bury input constants inside functions. Should be arguments instead.
- Can relocate functions in a separate file, or into a package, and everything "*just works*"

Anatomy of a function

- R allows us to create functions "on the fly". This is the essential difference between a compiled language like C and an interpreted language like R. While an R session is running, we can add new capabilities to it.
- The artist Escher would like this:

  > *There is a* **function** *named* **function**. *That is to say,* **function** *is a* **function** *that creates functions!*

  Maybe that is more *Dr. Seuss*.

Anatomy of a function

- `somethingGood()` is a new function, created by the `function()` function like so:

```
somethingGood <- function(x, y, z){
    ## code in here
}
```

- We Choose
  1. the function's name, `somethingGood`.
  2. the names of the arguments, which are x, y and z
- To "call" (i.e, "use") that function, we'll write

```
somethingGood(whatever1, whatever2, whatever3)
```

  Built-in R functions have short names like `ls()` `lm()`, `glm()`.

- The terms **arguments** and **parameters** are interchangeable. I often say **inputs**.

Anatomy of a function ...

- In R, we do not use the word “options” for function inputs. That confuses people, who think you are referring to session options and the R function called `options()`.
- `arguments` *may* be specified with default values, as in

```
somethingGood <- function(x1 = 0, x2 = NULL){
```

- After the squiggly brace, any valid R code can be used.
- **What happens in the function stays in the function.** Does not affect same-named variables in the workspace.
- Return results: When when the function’s work is finished, a single object’s name is included on the last line.

```
somethingGood <- function(x1 = 0, x2 = NULL){
   ## suppose really interesting calculations
      create res, a result
   res
   }
```

Anatomy of a function ...

- Please remember.
  1. The return includes one object
  2. That object can be a vector, a matrix, a data frame, or a list including (one or more of) all of the above.
- If a returned value includes a large matrix or data frame, one is wise to NOT PRINT it into the session by default. Wrap your return value inside `invisible()`

```
somethingGood <- function(x1 = 0, x2 = NULL){
## suppose really interesting calculations
   create \texttt{res}, a result
   invisible(res)
   }
```

- Can break out of function by calling `return()`. This offers a pleasant way to use an if/then condition to stop work.

Anatomy of a function ...

```r
somethingGood <- function(x1 = 0, x2 = NULL){
##  suppose you created res
 if (someLogicalCondition)
    return(invisible(res))
## otherwise, go on and revise res further.
 invisible(res)
 }
```

Anatomy of a function ...

- Functions can be nested. If there is a special purpose function that you don't expect to use anywhere else, hide it in the top of the function where you use it.

```
somethingGood <- function(x1 = 0, x2 = NULL){
   chore <- function (z){
   ## calculation about z argument
   ## or x1 or x2 from enclosuring environment
}
   z.candidate <- R calculations involving x1
      and x2
   result <- chore(z.candidate)
   result
}
```

- chore() is available only within somethingGood()

R Functions pass information “by value”

- Users should organize their information “here”, in the current environment
    - the function must not be allowed to damage information.
- Thus, we send info “over there” to a function
- We get back a new something.

```
g <- somethingGood(whatever1, whatever2)
```

spawns a new thing `g`

- Can clobber old things (on purpose?)

```
whatever1 <- somethingGood(whatever1, whatever2)
```

- *Emphasis*. A function **DOES NOT**
    - change variables we give to the function
    - change other variables in the user workspace
- The super assignment `<<-` allows an exception to this, but R Core recommends we avoid it. If you must do this, the `assign()` function is a safer method.

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

Standardize notation about functions

- Programmers (me) often lazy about leaving behind clear documentation.
- They like to write functions, not instructions
- The Literate Programming movement (@1990) began as a way to blend documentation with functions, to encourage programmers to try harder

Standardize notation about functions

The Roxygen style uses text markup like so

```
##’ terse statement of function purpose
##’
##’ paragraph about function
##’
##’ Paragraphs of "Details"
##’ @param x words about x
##’ @param y words about y
##’ @return a description of the function’s return
##’ @author Paul Johnson <pauljohn@@ku.edu>
myfunction <- function(x, y){
    ## imagine code here
 }
```

Roxygen can be turned into package documentation

- Hadley Wickham has provided many useful R packages, including roxygen2
- Write roxygen markup, then run the `roxygenize` function that creates documentation.
- Details about package markup: http://r-pkgs.had.co.nz/man.html#text-formatting

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

reverse a factor's levels

- In many projects, we have "Likert Scales"
- Often, users have factor variables for which the "polarity" must be reversed.
  - high-to-low must become low-to-high
  - However, they usually have some values like "Skip" or "Not Avail" that they want to leave at the end of the output.

If we did not have to worry about the special values, this would be easy as pie!

```
##' Reverse a factor's levels
##'
##' This requires a factor variable
##' @param x A factor variable
##' @return A reversed factor variable
##' @author Paul Johnson <pauljohn@@ku.edu>
revs <- function (x){
     if (!is.factor(x)) stop("your variable is not a factor")
     rlevels <- rev(levels(x))
     factor(x, levels = rlevels)
 }
```

Lets test that

```
x <- c("hot", "hot", "cold", "medium", "medium",
    "hot")
zz1 <- ordered(x, levels = c("hot", "medium",
    "cold"))
x2 <- revs(zz1)
table(x2, zz1, dnn = list("x2", "zz1 is the
    original"))
```

```
         zz1 is the original
x2         hot medium cold
  cold       0      0    1
  medium     0      2    0
  hot        3      0    0
```

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

R uses "lexical scope"

- The highest, available-everywhere "environment" is the user workspace.
- Using a function creates an "closure" within which changes are contained.
- However, in R a function can "look up" for something that it thinks it needs. It can reach "up" to the user workspace and pull in information.

That outward-looking tendency is helpful

- If your functions use the same information, perhaps it is too boring or tedious to name those things as variables in your function

```
x <- 30
aa <- letters[5:10]
getXYZ <- function(m1, m2){
    res1 <- paste(m1, x, sep = "_")
    res2 <- paste(aa, m2, sep = "_what?_")
    list(res1, res2)
 }
getXYZ(m1 = c(1, 2, 3), m2 = c(98, 99))
```

```
[[1]]
[1] "1_30" "2_30" "3_30"

[[2]]
[1] "e_what?_98" "f_what?_99" "g_what?_98" "h_what?_99" "i_what?_98"
    "j_what?_99"
```

That outward-looking tendency is helpful ...

- Notice: The function went and retrieved "`x`" and "`aa`" from the workspace
- They were not passed in as arguments

That outward-looking tendency may be harmful

- Fail! If the x and aa in the workspace are not the same ones you wanted in your function
- That's why I'm very worried about undefined variables in functions.
  - In C or similar language, we would get an error
  - In R, we don't get an error or even a warning if R finds something that *seems* to fit.
- Because commonly used variable names like "`x`", "`y`", "`dat`" are floating about both in the workspace and in functions I write, I'm especially vulnerable to this trouble.
- The package "`codetools`" has a function `checkUsage()` which can help identify undefined variables.

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

A ”Variable Key” example

- The input data set had names like “V1”, “V2”, . . ., “V99”.
- Client provided an Excel sheet with new names like this

oldname	newname
V1	Respondent ID
V2	Respondent Age
V3	city - residence
V4	state - residence

- We want to respect their `newname` choices as much as possible, but
  - we cannot use those as column names (spaces and some minus signs).
- We also want consistency, so we decided to make all of these lower case.
- Can fix by running 4 commands on `newname` before replacing it:

A "Variable Key" example ...

```
newname <- c("Respondent ID", "Respondent Age",
    "city - residence", "state - residence")
## Change space to underscore
newname <- gsub(" ", "_", newname, fixed = TRUE)
## Replace minus with underscore
newname <- gsub("-", "_", newname, fixed = TRUE)
## Replace multiple underscores with one
    underscore
newname <- gsub("(_)\\1+", "_", newname)
## Lower case
newname <- tolower(newname)
newname
```

```
[1] "respondent_id"   "respondent_age"   "city_residence"
    "state_residence"
```

```
##colnames(dat) <- newname
```

A "Variable Key" example

- If we import 10 data frames with that same issue, then we have to have 40 lines of code to fix their names.
- I'd rather sequester those commands in a function,

```
##' Remove spaces, minus signs, and change
   letters to lower case
##'
##' Cleans up a character string. Does not do
   comprehensive
##' cleanup, just minus, spaces and capitals.
   Could extend to other flaws
##' @param x A vector of character string.
##' @return cleaned vector of strings
##' @author pauljohn@@ku.edu
cleanVarName <- function(x){
     x <- gsub(" ", "_", x, fixed = TRUE)
     x <- gsub("-", "_", x, fixed = TRUE)
```

A "Variable Key" example ...

```
     x <- gsub("(_)\\1+", "_", x)
     x <- tolower(x)
     x
 }
```

- And then run one line per data frame

```
colnames(dat1) <- cleanVarName(colnames(dat1))
colnames(dat2) <- cleanVarName(colnames(dat2))
## ...
colnames(dat10) <- cleanVarNames(colname(dat10))
```

- Possibly even a for loop that saves so much typing. If we had the data.frame names within a vector, or if we were importing files from a list, we could automate this.

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

browser() and debug()

There are 3 things to try to get a handle on what your function does.

1. Type the function's name, check out the way R looks at your code.

   ```
   getXYZ
   ```

   ```
   function(m1, m2){
      res1 <- paste(m1, x, sep = "_")
      res2 <- paste(aa, m2, sep = "_what?_")
      list(res1, res2)
   }
   ```

   Note: No parens, no arguments
   This works with any R function. Type its name. Even `q`.

2. Ask R to “stop” whenever it tries to use your function with `debug()`.

   ```
   debug(getXYZ)
   ```

   After that, when you use that function, R will offer an interactive view of what that function does.

browser() and debug() ...

- result depends on which editor you are using, I'll demonstrate.

debug cheatsheet

keystroke	result
n	move into next sub-process or next line
Enter	run current line (similar to "n")
c	let the function run
Q	abort the function at its current position

3. Put the function call `browser()` in the middle of your function's code.

```
getXYZ <- function(m1, m2){
    res1 <- paste(m1, x, sep = "_")
    res2 <- paste(aa, m2, sep = "_what?_")
    browser()
    list(res1, res2)
 }
```

browser() and debug() ...

This is the same as `debug()`, except that the function runs up to the point at which you inserted `browser()`.

- especially handy when you have a long function and you don't want to run "n" over and over again.

Outline

1. Functions differentiate R from Others
2. Roxygen markup is increasingly popular
3. Example of a function
4. Problems/Opportunities to be Aware Of
5. Another example
6. Interacting with function objects
7. Conclusion

The Ease of Creating Functions

- The ease of creating (and packaging) new functions is, no doubt, an important part of the R success story
- We hope these slides give the user some confidence about writing functions, or reading more about writing functions.
- There is a chapter about writing functions in the *Introduction to R* that is provided with R itself.

Additional Readings

- Additional readings that I enjoy are
  - Matloff, Norman. S. (2011). *The Art of R Programming: a tour of statistical software design*. San Francisco: No Starch Press.
  - Chambers, J. M. (2008). *Software for Data Analysis: programming with R*. London: Springer.
  - Wickham, Hadley (2014). Advanced R. Boca Raton, FL: CRC.

vignettes in the rockchalk package

- Rstyle: Commentary about how your code ought to look.
- Rchaeology: more advanced function writing tips, especially concentrating on terminology about “calls”, “eval”, and R functions to interpret function arguments.

References

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Session

```
sessionInfo()
```

```
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 18.04 LTS

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
     LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=en_US.UTF-8     LC_MONETARY=en_US.UTF-8
     LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8
    LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  base

loaded via a namespace (and not attached):
[1] compiler_3.4.4 tools_3.4.4
```