Writing Functions In R

Necessity Really is a Mother

Paul E. Johnson[1][2]

[1]University of Kansas, Department of Political Science [2]Center for Research Methods and Data Analysis

October 5, 2016

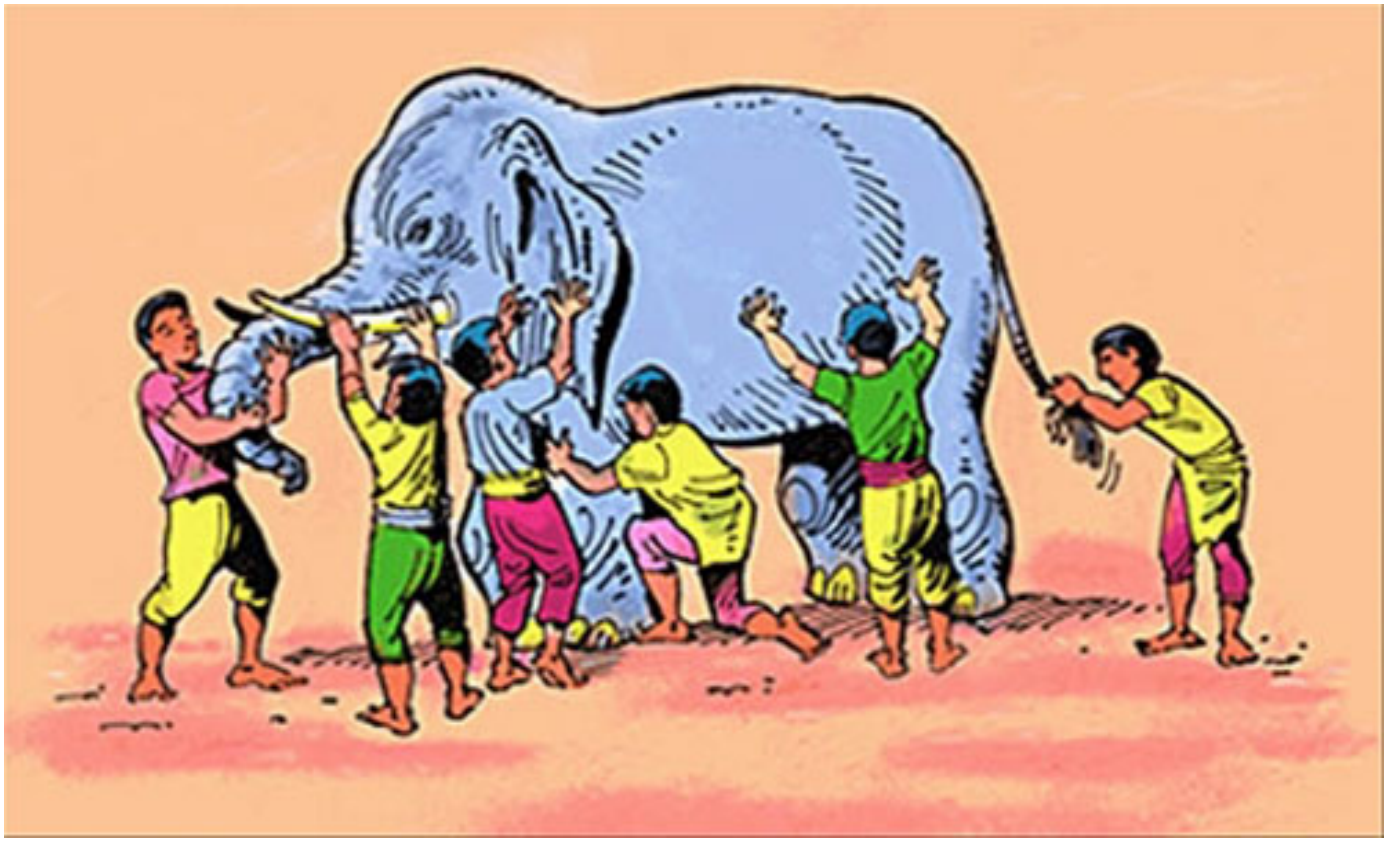
Outline

Outline

Did You Ever Write a Program?

- If Yes: R's different than that.
- If No: Its not like other things you've done.
- In either case, don't worry about it :)

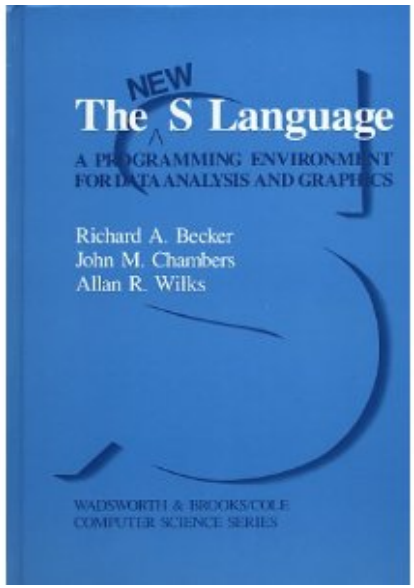
R is a little bit like an elephant



Its a tree trunk! Its a snake! Its a brush!

The R Language is like S, of course

- The S Language– John Chambers, et al. at Bell Labs, mid 1970s.
  See Richard Becker's "Brief History of S" about the AT&T years
- There have been 4 generations of the S language.
- Many packages now were written in S3, but S4 has been recommended for new packages for at least 5 years.
- A new framework known as "reference classes" is now being developed (was jokingly referred to as "R5" at one time)



S3: *The New S Language* 1988

Is R a Branch from S?

S pioneers now work to advance R.

- R can be seen as a competing implementation of 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.
- Open Source, Open Community, open repository CRAN

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

Outline

Overview: 3 reasons to write functions

- Preserve your sanity: isolate specific work.
  - Side benefit: preserve sanity of others who have to read your code
- Facilitate re-use of your ideas
- Co-operate with R functions like `lapply()` and `boot()` which REQUIRE functions from us.

Functions: Separate Calculations Meaningfully

- New programmers tempted to craft a giant sequence of 1000 commands
- Just Don't!

  Problem No other human can comprehend that mess

  Solution Write functions to calculate separate parts
- I don't feel comfortable with any function until I have a small "working example" to explore it (many available `http://pj.freefaculty.org/R`)

Re-use your work

- If you write a function, you can put it to use in many different contexts
- If you write a gigantic stream of 1000 commands, you can't.

Avoid for loops with lots of meat inside

Instead of this:

```
for(i in 1:1000}{
      1000s of lines here full of x[i], z[i], and so forth
      }
```

We want:

```
fn1 <- function( arguments ) { ... }
fn2 <- function( arguments ) { ... }
for(i in 1: 1000){
   y <- fn1(x, ... )
   z <- fn2(y, ...)
   }
```

This is easier to read, understand, and more re-usable!

Personal confession

- My first attack at any problem is often a long string of commands that are not separable, not readable
- The revision process usually causes me to segregate code into separate pieces
- One hint that you need a function: constant cutting and pasting of code scraps from one place to another

When finished, I Wish Your R program would look like this

```
myfn1 <- function (argument1, argument2, ...){
   ## lines here using argument1, argument2
 }
myfn2 <- function (argument3, argument4){
   ## lines here
 }
## Try to perfect the above. Then use them
##
 a <- 7
 b <- c(4, 4, 4, 4, 2)
 great1 <- myfn1(a, b, parm3 = TRUE)
 great2 <- myfn2(b, great1)
```

How to Create 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 one:

  The word `function` is a function that creates functions!

- A new function `somethingGood()` is defined by a stanza that begins like so:

```
somethingGood <- function(arguments){
```

- `somethingGood` is a name we choose

How to Create a Function ...

- The terms `arguments` and `parameters` are interchangeable. I often say `inputs`. In R, do not say "options", that confuses people because R has a function called `options()` that governs the working session.
- `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. We can even define functions inside the function!
- What happens in the function stays in the function. Things you create inside there do not escape the closure unless you really want them to.
- Return results: When when the function's work is finished, a single object's name is included on the last line.

How to Create a Function ...

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

- There are some little wrinkles about returns that will be discussed later. But, for now, I plant some seeds in your mind.
    - The return includes one object
    - The result will ordinarily print in the R terminal when the function runs, but we can prevent that by using the last line as

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

How to Create a Function ...

- It is possible to exit sooner, to short-circuit the calculations before they have all run their course. That is what the `return()` function is for.

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

R Functions pass information "by value"

- The basic premise is that users should organize their information "here", in the current environment, and it is important that the function should not accidentally damage it.
- Thus, we send info "over there" to a function
- We get back a new something.
- The function **DOES NOT** change variables we give to the function
- The super assignment $<<-$ allows an exception to this, but R Core recommends we avoid it. Only experts should use it.

A simple example of a new function: doubleMe

```
doubleMe <- function(input = 0){
  newval <- 2 * input
}
```

The function's name is "doubleMe"
I choose a name for the incoming variable "input".
Other names would do (must start with a letter, etc.):

```
doubleMe <- function(x){
    out <- 2 * x
}
```

The last named thing is the one that comes back from the function.
Note, explicit use of a return function is NOT REQUIRED. The last named thing comes back to us.

Key Elements of doubleMe

```
doubleMe <- function(input = 0){
  newval <- 2 * input
}
```

doubleMe a name with which to access this function. Because of my background in "Objective C", I like this style of name. Don't put periods in function names unless you know about "classes" and are using them.

input a name used INTERNALLY while making calculations

= 0 An *optional* default value.

newval Last calculation is returned.

How to Call doubleMe

- What is 2 * 7?

```
(doubleMe(7))
```

```
[1] 14
```

- The caller may name the arguments explicitly:

```
(doubleMe(input = 8))
```

```
[1] 16
```

- Wonder why I use parentheses around everything? Its just a presentational trick. The default action is "print", and that's what happens when you put something in parentheses without a function name.
- The alternatives are:

```
print(doubleMe(input = 3))
```

How to Call doubleMe ...

```
[1] 6
```

- or

```
x <- doubleMe(input = 2)
x
```

```
[1] 4
```

Generally, I Prefer Clarity in the Call

- The "call" is the code statement that puts a function to use. The call includes the function's name and all arguments.
- This works

```
doubleMe(10)
```

- But wouldn't you rather be clear?

```
doubleMe(input = 10)
```

- When there are many arguments, naming them often helps prevent accidental matching of input to arguments (R's positional matching can be fooled).

Function Calls

- But if you name the argument wrong, it breaks

```
> doubleMe(myInput = 7)
Error in doubleMe(myInput = 7) :
unused argument(s) (myInput = 7)
```

- What if you feed it something unsuitable?

```
> doubleMe(lm(rnorm(100) ~ rnorm(100)))
Error in 2 * input : non-numeric argument to binary
    operator
In addition: Warning messages:
1: In model.matrix.default(mt, mf, contrasts) :
  the response appeared on the right-hand side and was
      dropped
2: In model.matrix.default(mt, mf, contrasts) :
  problem with term 1 in model.matrix: no columns are
      assigned
```

Vectorization

This is not always true, but OFTEN:

- We get "free" "vectorization"

```
(doubleMe(c(1,2,3,4,5)))
```

```
[1]  2  4  6  8 10
```

- But it won't allow you to specify too many inputs:

```
> doubleMe(1,2,3,4,5)
  Error in doubleMe(1, 2, 3, 4, 5) :
        unused argument(s) (2, 3, 4, 5)
```

Vectorization: vector in $\Longrightarrow$ vector out

- Oops. I forgot the input

```
doubleMe()
```

Gives the default value.

print.function magic

- Oops. I forgot the parentheses

```
doubleMe
```

```
function(input = 0){
  newval <- 2 * input
}
```

- Similarly, type "lm" and hit return. Or "predict.glm". Don't add parentheses.
- When you type a function's name, R thinks you want to print that thing, and it invokes a "print method" for you, called `print.function()`. (That's `print` as applied to an object of class function.)
- Generally, `print.function()` will display the R internal functions in a "tidied up" format. Your functions–the ones you have created in your session–are generally not tidied up. That is discussed in the Rstyle vignette distributed with rockchalk.

predict.glm, for example

```
predict.glm
```

```
function (object, newdata = NULL, type = c("link", "response
    ",
    "terms"), se.fit = FALSE, dispersion = NULL, terms =
        NULL,
    na.action = na.pass, ...)
{
    type <- match.arg(type)
    na.act <- object$na.action
    object$na.action <- NULL
    if (!se.fit) {
        if (missing(newdata)) {
            pred <- switch(type, link = object$
                linear.predictors,
                response = object$fitted.values, terms =
                    predict.lm(object,
                  se.fit = se.fit, scale = 1, type = "terms"
                      ,
                  terms = terms))
            if (!is.null(na.act))
```

predict.glm, for example ...

```
                pred <- napredict(na.act, pred)
        }
        else {
            pred <- predict.lm(object, newdata, se.fit,
                scale = 1,
                type = ifelse(type == "link", "response",
                    type),
                terms = terms, na.action = na.action)
            switch(type, response = {
                pred <- family(object)$linkinv(pred)
            }, link = , terms = )
        }
    }
    else {
        if (inherits(object, "survreg"))
            dispersion <- 1
        if (is.null(dispersion) || dispersion == 0)
            dispersion <- summary(object, dispersion =
                dispersion)$dispersion
        residual.scale <- as.vector(sqrt(dispersion))
        pred <- predict.lm(object, newdata, se.fit, scale =
            residual.scale,
```

predict.glm, for example ...

```
            type = ifelse(type == "link", "response", type),
            terms = terms, na.action = na.action)
        fit <- pred$fit
        se.fit <- pred$se.fit
        switch(type, response = {
            se.fit <- se.fit * abs(family(object)$mu.eta(fit
                ))
            fit <- family(object)$linkinv(fit)
        }, link = , terms = )
        if (missing(newdata) && !is.null(na.act)) {
            fit <- napredict(na.act, fit)
            se.fit <- napredict(na.act, se.fit)
        }
        pred <- list(fit = fit, se.fit = se.fit,
            residual.scale = residual.scale)
    }
    pred
}
<bytecode: 0x2dbc848>
<environment: namespace:stats>
```

Function Calls: Local versus Global

- The variables we create in "our" session are generally in the Global Environment.
- Local variables in functions.
  - Function arguments are local variables
  - Variables created inside are local variables
- Local variables cease to exist once R returns from the function
- After playing with `doubleMe()`, we note that the variable `input` does not exist in the current environment.

```
> ls()
[1] "doubleMe"
> input
Error: object 'input' not found
```

Check Point: write your own function

- Write a function "myGreatFunction" that takes a vector and returns the arithmetic average.
- Generate your own input data, x1, like so

```
set.seed(234234)
x1 <- rnorm(10000, m = 7, sd = 19)
```

- In `myGreatFunction()`, you can use any R functions you want, it is not necessary to re-create the `mean()` function (unless you really want to :))
- After you've written `myGreatFunction()`, use it:

```
x1mean <- myGreatFunction(x1)
x1mean
```

- Now stress test your function by changing x1

```
x1[c(13, 44, 99, 343, 555)] <- NA
myGreatFunction(x1)
```