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 12, 2016

Outline

Outline

Function Anatomy

Arguments: The Arguments of a function

```
someWork <- function(what1, what2, what3)
```

what1, what2, and what3 become "local variables" inside the function.

- named arguments must begin with letters, may include numbers, or "_" or "-" or ".", but no "" or "," or "*"

Arguments: named and unnamed R variables

```
someWork <- function(what1, what2, what3, ...)
```

- In R, everything is an "object". what1, what2, what3 can be *ANYTHING!*
- That is a blessing and a curse
  - Blessing: Flexibility! Let an argument be a vector, matrix, list, whatever. The function declaration does not care.
  - Curse: Difficulty managing the infinite array of possible ways users might break your functions.
- It is your choice, whether your function should receive
  - 2 integers (x, y), or
  - one vector of 2 integers (x)
  - For examples of this, look at the arguments of plot.default, arrows, segments, lines, and matplot.

Peculiar spellings like . . .

Functions may have an argument "..."

- It seems funny, but "..." is actually a word made up of *three legal* characters
- If the caller includes any named arguments that are not `what1`, `what2`, or `what3`, then the R runtime will put them in a list and give them to "..."
- The "..." argument requires special effort.

Function Arguments: R is VERY Lenient

- R is lenient. Perhaps too lenient!
  - Arguments are not type-defined. In

```
myF <- function(x1, x2){
```

    x1 could be an integer vector, a character string, a regression model, or anything
  - Function writer may declare default values, but need not. These are acceptable definitions

```
someWork <- function(what1, what2, what3, what4,
    what5)

someWork <- function(what1 = 0, what2 = NULL, what3
    = c(1,2,3), what4 = 3 * what1, what5)

someWork <- function(what1 = 0, what2, what3, what4
    = 3 * what1, what5)
```

R is Lenient toward Users as Well

- R is lenient on format of function calls

```
someWork(1)
someWork(what=1, someObject)
someWork(what5 = fred, what4 = jim, what3 = joe)
```

  - Not all parameters must be provided
  - Partial argument matching

R is Lenient About Undefined Variables

This is a convenience and a curse

Variables inside functions might not be resolved the way you expect.

- If a variable is used, but not defined inside the function, R will "look outward" for it
- This is "lexical scope" in action. The runtime engine searches through a sequence of "frames", ending up at the user's workspace (which is the Global Environment).

Example of the Undefined Variable Problem

- Suppose "dat" exists in your workspace.
- Here is a function that will find "dat", even if that's not what you intend.

```
myFn <- function(x, y, z){
    dat1 <- 2 * x + 3 * y
    res <- sqrt(dat)
}
```

  Note my typographical error (dat where dat1 should be). The function should crash, but it will not.
- R will find the wrong "dat" and the result we get will be unhelpful
- In my experience, this is the single most important cause of hard-to-find flaws in user code.

Inside the function, Check Arguments

- Check and Re-format
  - "argument checking": diagnose what arguments the user provided
  - Figure out if they are "correct" for what the function requires.
- You choose how sympathetic you want to be toward users. Do you want to accept incomplete input and re-format it? (In rockchalk/R/genCorrelatedData.R, find my "re-formatter" functions like `makeVec()`, `vech2mat()`).

Arguments: When to Use Defaults?

- I don't know, but ...
- As an R beginner, I took a very conservative strategy of putting defaults on everything!

```
function(what1 = NULL, what2 = NULL, what3 = 3, what4 =
    ``a'')
```

- That way, if a user forgets to provide "what3", then the system will not go looking for it.
- If the defaults usually work, this is concise, easy to read.
- Most functions in R base code don't set defaults for all, or even most, variables.
- Instead, we manage arguments with more delicacy. Insisting on NULL defaults is "throwing the baby out with the bath water."

Functions that Help while Checking Arguments

missing()

- Inside a function, `missing()` is a way to ask if the user supplied a value for an argument.

```
doSomething <- function(what1, what2, what3, what4){
    if (missing(what1)) stop(``you forgot to specify
        what1'')
}
```

- I've found this conservative approach to be an error-preventer. If x is not provided, or if the user gave us a NULL, we better adapt!

```
if (missing(x) || is.null(x)) ## do something
```

Functions that Help while Checking Arguments ...

Type Checks

There are many "is." functions. Ask if x is a certain type of thing:

is.vector()	TRUE or FALSE: are you a vector?
is.list()	TRUE or FALSE: are you a list?
is.numeric()	TRUE or FALSE: are you numeric?
	You get the general idea, I hope

Functions that Help while Checking Arguments ...

Look at all of these is. functions!

is.array	is.loaded	is.object
is.atomic	is.logical	is.ordered
is.call	is.matrix	is.package_version
is.character	is.mts	is.pairlist
is.complex	is.na	is.primitive
is.data.frame	is.na<-	is.qr
is.double	is.na.data.frame	is.R
is.element	is.na<-.default	is.raster
is.empty.model	is.na<-.factor	is.raw
is.environment	is.name	is.recursive
is.expression	is.nan	is.relistable
is.factor	is.na.numeric_version	is.single
is.finite	is.na.POSIXlt	is.stepfun
is.function	is.null	is.symbol
is.infinite	is.numeric	is.table
is.integer	is.numeric.Date	is.ts
is.language	is.numeric.difftime	is.tskernel
is.leaf	is.numeric.POSIXt	is.unsorted
is.list	is.numeric_version	is.vector

Functions that Help while Checking Arguments ...

Size Checks

length()

nrow(), ncol() number of rows (columns) from a matrix

NROW(), NCOL() Works if input is matrix, but will treat a vector as a one column matrix.

dim() Returns 2 dimensional vector, works with matrices and arrays

Stop, Warn

stop(), stopifnot() Ways to abend the function

warning Will return, but throws a message onto the R warning log. User can run warnings() to see messages.

Outline

dot management strategies I use

- Get the dots. The R parser notices all named arguments that seem to be "unclaimed" by other arguments.
- If you have an argument named "...", those arguments go into a list named "...".
- So you should run:

```
dots <- list(...)
```

Inspect and filter the dots

- In some functions, I need to pass in "..." that are aimed at several different subroutines.
- examples
  1. rockchalk::plotSlopes
  2. kutils::peek
- Find out what arguments a function might want:
- So you should run:

```
formals(function_name_here)
## Possibly just
names(formals(function_name_here))
```

or read the documentation

Manually pick-and-choose items out of dots

- I will type in the names of the arguments I know I want to keep
- Put them (and their values) into a separate list
- then use "modifyList" to place the user-provided dots "on top" of my favorite settings.

```
drawHist <- function(x, normal = TRUE, pois = FALSE,
kde = TRUE, nbinom = FALSE, ...){
  dots <- list(...)
  pjfavoriteDefaults <- list(right = FALSE, prob = TRUE,
  border = gray(.80), include.lowest = TRUE,
  plot = FALSE)

histNames <- c("breaks", "freq", "probability",
               "include.lowest", "right", "density",
               "angle", "col", "border", "main",
               "xlim", "ylim", "xlab", "ylab", "axes",
               "plot", "labels", "nclass")
```

Manually pick-and-choose items out of dots ...

```
argsForHist <- dots[ names(dots) [names(dots) %in%
    histNames]]
argsForHist <- modifyList(pjfavoriteDefaults,
    argsForHist)
```

- Then we may want to purge those particular arguments from the dots list (not possibly needed for other functions). This prevents accidentally sending an argument for hist to an lm function, for example.

Trust the formals

- Typing in the names seems stupid.
- If the R functions change, then this code will be outdated.
- If I'm in a gambling mood, then I'll not type in all of the names and I'll use the formals function to get the names of arguments. Be careful.
- Example usage

```
histNames <- names(formals(function_name_here))
```

- After that, you may have to clean out ones you don't want to keep. Observe:

```
names(formals(hist.default))
```

Trust the formals ...

```
 [1] "x"                "breaks"           "freq"
                "probability"    "include.lowest" "right"
                "density"        "angle"          "col"
                 "border"         "main"
[12] "xlim"             "ylim"             "xlab"
               "ylab"           "axes"           "plot"
               "labels"         "nclass"         "
    warn.unused"     "..."
```

- Quick way to get the argument names into a vector:

```
dput(names(formals(hist.default)))
```

```
c("x", "breaks", "freq", "probability", "include.lowest
    ", "right",
"density", "angle", "col", "border", "main", "xlim", "
    ylim",
"xlab", "ylab", "axes", "plot", "labels", "nclass", "
    warn.unused",
"...")
```

Trust the formals ...

and then edit that for use in a function. Cut out the dots!

Pruning argument lists

- Usually, I want to check for some arguments and keep them for use, as described above.
- In order to explain this fully, I'd need you to understand the use of R's "do.call", which is a deep thought that we should probably investigate in a separate talk.
- I understood the beauty of that in this example code: `http://pj.freefaculty.org/R/WorkingExamples/efficiency-stackListItems-01.R`
- There is quite a bit of discussion of do.call in the essay "Rchaeology" that is included in the `rockchalk` package.

The kutils::peek example

- The peek function has 2 nested functions
  - one for creating histograms
  - one for creating barplots
- The dots argument may be users to pass in arguments intended for hist, barplot, or pdf.

```
peek <-
    function(dat, sort = TRUE, file = NULL, textout = FALSE, ask, ...,
             xlabstub = "kutils peek: ", freq = FALSE,
             histargs = list(probability = !freq),
             barargs = list(horiz = TRUE, las = 1))
{
    quickhist <- function(i){
        args <- list(xlab = paste0(xlabstub, i),
                     main = i)
        histargz <- modifyList(args, dots)
        histargz <- modifyList(histargz, histargs)
        qqq <- modifyList(list(x = dat[ , i]), histargz)
        h1 <- do.call("hist", qqq)
        if (textout){
            df1 <- data.frame("midpoints" = h1$mids, "density" = h1$density)
            cat(i, "\n")
            print(df1)
        }
```

The kutils::peek example ...

```
}

quickbar <- function(i){
    targs <- list(dat[ , i], exclude = NULL,
                  dnn =  if (!freq) "Proportion" else "Frequency")
    targz <- modifyList(targs, dotsForTable)
    ## don't allow hist or barplot arguments to to to table, silences
        warning
    ## targz[c(names.par, names.barplot.unique, names.hist.unique)] <- NULL
    t1 <- do.call("table", targz)
    if (!freq) t1 <- t1/margin.table(t1)
    names(t1) <- ifelse(is.na(names(t1)), "NA", names(t1))
    ## decision 1: brute force chop
    ## names(t1) <- shorten(names(t1), k = LIMIT, unique = TRUE)
    names(t1) <- sapply(names(t1), stringbreak, k = LIMIT)
    args <- list(t1, main = i,
                 xlab = paste(xlabstub,  i, if (freq)"Frequencies" else "(
                     Proportion)"))
    barargz <- modifyList(args, dots)
    barargz <- modifyList(barargz, barargs)
    ## Remove args that would have gone to hist
    barargz[names.hist.unique] <- NULL
    par.old <- par(no.readonly = TRUE)
    ## if longest name exceeds guess of space in margin,
    ## then pad the margin left side.
    marinch <- par("mai")
    marrequired <- max(strwidth(names(t1), units = "inches"))
    if (marrequired + 0.35 > marinch[2]) {
        marinch[2] <- marrequired + 0.5
        par(xpd = TRUE)
```

The kutils::peek example ...

```
            par("mai" = marinch)
        }
        do.call("barplot", barargz)

        par(par.old)
        if (textout) {
            cat(i, "\n")
            print(t1)
        }
    }

    ## limit on character strings for barplots
    LIMIT <- 30

    ## We get lots of warnings about inappropriate arguments to functions.
    ## Focus on most likely objections by getting names unique to
    ## hist and removing them from the bar portion, and
    ## names unique to bar and removing from the hist portion
    names.par <- names(par())
    names.hist <- removeMatches(names(formals(hist.default)), "...")
    names.barplot <- removeMatches(names(formals(barplot.default)), "...")
    names.barplot.unique <- names.barplot[!names.barplot %in% names.hist]
    names.hist.unique <- names.hist[!names.hist %in% names.barplot]

    varType <- function(x){
        ifelse(is.numeric(x), "numeric",
        ifelse(is.character(x) | is.factor(x), "factor", "noneoftheabove"))
    }

    if (is.atomic(dat)){
```

The kutils::peek example ...

```
        xclean <- xname <- deparse(substitute(dat))
        dat <- as.data.frame(dat)
        if (length(grep("\\$", xname) > 0)){
            xsplit <- unlist(strsplit(xname, "\\$"))
            xclean <- xsplit[length(xsplit)]
        }
        colnames(dat) <- xclean
    } else {
        if (!is.data.frame(dat)) dat <- as.data.frame(dat)
    }

    namez <- colnames(dat)
    namez <- if(sort) sort(namez) else namez

    colTypes <- sapply(dat, varType)
    colTypes <- colTypes[namez]

    if (length(colTypes[colTypes=="noneoftheabove"]) > 0) {
        cat(paste("These variables are being omitted because they are",
                  "neither numbers nor can they be interpreted as factors: \n"))
        cat(paste(names(colTypes[colTypes=="noneoftheabove"]), "\n"))
        ## remove unrecognized types
        colTypes <- colTypes[colTypes != "noneoftheabove"]
    }

    dots <- list(...)

    ## copy args for table, remove from dots
    tableFormals <- c("exclude", "dnn", "useNA", "deparse.level")
    dotsForTable <- dots[tableFormals[tableFormals %in% names(dots)]]
```

The kutils::peek example ...

```
dots[names(dotsForTable)] <- NULL

## copy args for pdf, remove from dots
pdfFormals <- c("width", "height", "onefile", "family", "title", "fonts",
                "version", "paper", "encoding", "bg", "fg", "pointsize",
                "pagecentre", "colormodel", "useDingbats", "useKerning",
                "fillOddEven", "compress")
dotsForPDF <- dots[pdfFormals[pdfFormals %in% names(dots)]]

deviceFormals <- c("width", "height")
dotsForDevice <- dots[deviceFormals[deviceFormals %in% names(dots)]]
dots[names(dotsForPDF)] <- NULL



## Unless user sets ask, we assume TRUE if there is no file argument
if (missing(ask)) {
    if (is.null(file)) ask <- TRUE else ask <- FALSE
}

if (!is.null(file)){
    if (!is.character(file)) stop("Sorry, file has to be a character string"
        )
    pdfargs <- list(file = file, onefile = TRUE)
    pdfargz <- modifyList(pdfargs, dotsForPDF)
    do.call("pdf", pdfargz)
} else {
    do.call("dev.new", dotsForDevice)
}
```

The kutils::peek example ...

```
    if (is.null(file) || isTRUE(ask)) devAskNewPage(TRUE)

    for (i in names(colTypes)){
        if ( colTypes[i]=="numeric" ){
            quickhist(i)
        } else {
            quickbar(i)
        }
    }
    if (!is.null(file)) dev.off()
    namez
}
```

Outline

The Return Has To Be Singular

- When you use a function, it is necessary to "catch" the output with a single object name, as in

```
newthing <- doubleMe(32)
newthing
```

```
[1] 64
```

```
is.numeric(newthing)
```

```
[1] TRUE
```

```
is.vector(newthing)
```

```
[1] TRUE
```

- We expect "doubleMe(32)" should return 64, and it does.

The Return Has To Be Singular ...

- The "vectorization for free" gives us a hint of what is to follow.

```
newthing <- doubleMe(c(1,2,3,4,5))
newthing
```

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

```
is.numeric(newthing)
```

```
[1] TRUE
```

```
is.vector(newthing)
```

```
[1] TRUE
```

- R allows us to return one "thing", but "thing" can be a rich, informative thing!

Generalization. Return a list

- A list may include numbers, characters, vectors ,etc
- or data frames or other lists
- Read code for function "lm"
- Almost ALL interesting R functions return a list of elements.

Hide details in attributes

- R objects can have attributes, which are, well, anything.
- Ever run `attributes(some_object_name)`?
- In S3, an object's class is simply determined by an attribute.

```
x <- c(1, 2, 3, 4, 6)
y <- c(44, 12, 44, 22, 4)
attributes(x)
```

```
NULL
```

```
attr(x, "hi") <- paste("Oh, my hearty friend, I greet you
    warmly", rnorm(6))
attr(x, "areg") <- lm(y ~ x)
attributes(x)
```

Hide details in attributes ...

```
$hi
[1] "Oh, my hearty friend, I greet  you warmly 0
    .58552881784385 6" "Oh, my hearty friend, I greet  you
    warmly 0.709466017509524" "Oh, my hearty friend, I
    greet  you warmly -0.109303314681054"
[4] "Oh, my hearty friend, I greet  you warmly
    -0.453497173462763" "Oh, my hearty friend, I greet  you
    warmly 0.605887455840394" "Oh, my hearty friend, I
    greet  you warmly -1.81795596770373"

$areg

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     44.919       -6.162
```

```
summary(attr(x, "areg"))
```

Hide details in attributes ...

```
Call:
lm(formula = y ~ x)

Residuals:
      1       2       3       4       5
  5.243 -20.595  17.568   1.730  -3.946

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   44.919     15.216   2.952   0.0599 .
x             -6.162      4.188  -1.471   0.2376
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
     1

Residual standard error: 16.11 on 3 degrees of freedom
Multiple R^2:  0.4191,  Adjusted R^2:  0.2255
F-statistic: 2.165 on 1 and 3 DF,  p-value: 0.2376
```

Check Point: Run this example function

Admittedly, this is a dumb example, but ...

- This function returns a regression object

```
aRegMod <- function(xin1, xin2, yout){ lm(yout ~ xin1
    + xin2) }
```

- Generate some data, run aRegMod()

```
dat <- rockchalk::genCorrelatedData(N = 100, rho = 0.3
    , beta = c(1, 1.0, -1.1, 0.0))
m1 <- with(dat, aRegMod(x1, x2, y))
```

- m1 is a "single object"
- Run "class(m1)", "attributes(m1)", "summary(m1)", 'str(m1)",