Rchaeology: Idioms of R Programming

Paul E. Johnson <pauljohn @ ku.edu>

March 22, 2013

This document was initiated on May 31, 2012. The newest copy will always be available at `http://pj.freefaculty.org/R` and as a vignette in the R package "rockchalk".

Rchaeology: The study of R programming by investigation of R source code. It is the effort to discern the programming strategies, idioms, and style of R programmers in order to better communicate with them.

Rchaeologist: One who practices Rchaeology.

These are Rcheological observations about the style and mannerisms of R programmers in their native habitats. Almost all of the insights here are gathered from the r-help and r-devel emails lists, the stackoverflow website pages for R, and the R source code itself. These are lessons from the "school of hard knocks."

How is this different from Rtips(`http://pj.freefaculty.org/R/Rtips.{pdf,html}`)?

1. This is oriented toward programming R, rather than using R.
2. It is more synthetic, aimed more at finding "what's right" rather than "what works."
3. It is written with Sweave (using Harrell's Sweavel style) so that code examples work.

Where did the "R Style" section go? It was removed to a separate vignette, RStyle, in the rockchalk package.

Contents

1 Introduction: R Idioms.

This vignette is about the R idioms I have learned while working on the rockchalk package. At the current time, I don't understand all of the R idioms that are common in the advanced R programmers' conversation, but I have a grasp on many of the staples that I have noticed before, and not understood.

There is a language gap between an R user and an R programmer. Users write "scripts" that use functions from R packages. Users don't write (many) functions. Users don't make packages. And many users are happy to keep it that way. For users who want to become programmers, there is usually a harsh awakening. R for development is a different language. Well, that's wrong. It is a different dialect.

A transitional R user will have to learn a lot of terminology and undergo a change of paradigm. There are resources available! Everybody should subscribe to the email lists for the R project (`http://www.r-project.org/mail.html`), especially r-help (for user questions) and r-devel. Rchaeologically speaking, that is the native habitat of the R programmers. There is also a burgeoning collection of blogs and Web forums, perhaps most notably the R section on StackExchange (`http://stackoverflow.com/questions/tagged/r`). Excellent books have been published. As an Rcheologist, I am drawn to the books that are written by the R insiders. *S Programming*, by William Venables and Brian Ripley (2000), is a classic. Books by pioneering developers Robert Gentleman, *R Programming for Bioinformatics* (2009), and John Chambers, *Software for Data Analysis* (2009), are, well, awesome. In 2012, I started teaching R programming using Norman Matloff's, *The Art of R Programming* (2012), which I think is great and recommend strongly (even though Professor Matloff is not one of the R Core Team members, so far as I know).

There are now three types of object-oriented programming in R (S3, S4, and reference classes) and the programmer is apparently free to select among them without prejudice. The best brief explanation of S3 that I've found is in Friedrich Leisch's brief note about R packaging, "Creating R Packages: A Tutorial" (2009). One should be mindful of the fact that R is provided with several manuals, one of which is the *R Language Definition*. I find that one difficult to understand, and I usually can't understand it until I search through the R source code and packages for usage examples.

2 do.call(), eval(), substitute(), formula().

If the transitional programmer understands these four functions, R programming will become much more understandable.

2.1 Rewriting Formulas. My Introductory Puzzle.

On May 29, 2012, I was working on a regression problem in the rockchalk package. I have a number of functions that receive elementary regressions and then change them.

The functions meanCenter() and residualCenter() receive a fitted regression model, transform some variables, and then fit a new regression. They have to change the data used by the regressions, and they have to revise the regression formulae. The non-centered variable "x1" is has to be centered to create "x1c". The original formula y ~ x1*x2 must be replaced with y ~ x1c*x2c. I found this to be a very complicated problem with a very satisfying answer.

My first effort used R's update function. That is the most obvious approach, it is the one that was recommended on r-help. I learned that update() is not sufficient. It is fairly easy to replace x1 with x1c in the formula, but not when x1 is logged or otherwise transformed. Here is the runable example code that demonstrates the problem.

```
> dat <- data.frame(x1 = rnorm(100, m=50), x2 = rnorm(100, m=50),
    x3 = rnorm(100, m = 50), x4 = rnorm(100, m=50), y = rnorm(100))
> m2 <- lm(y ~ log(x1) + x2*x3, data = dat)
> suffixX <- function(fmla, x, s){
      upform <- as.formula(paste(". ~ .", "-", x, "+", paste(x, s, sep=""), sep=""))
```

```
      update.formula(fmla, upform)
  }
> newFmla <- formula(m2)
> newFmla
> suffixX(newFmla, "x2", "c")
> suffixX(newFmla, "x1", "c")
```

Run that and check the last few lines of the output. See how the update misses x1 inside log(x1) or in the interaction?

```
> newFmla <- formula(m2)
> newFmla
y ~ log(x1) + x2 * x3
> suffixX(newFmla, "x2", "c")
y ~ log(x1) + x3 + x2c + x2:x3
> suffixX(newFmla, "x1", "c")
y ~ log(x1) + x2 + x3 + x1c + x2:x3
```

I asked the members of r-help.

Lately I've had very good luck with r-help. Gabor Grothendieck wrote an answer to r-help on May 29, 2012, "Try substitute:"

```
> do.call("substitute", list(newFmla, setNames(list(as.name("x1c")), "x1")))
y ~ log(x1c) + x2 * x3
```

Problem solved, in a single line.

That's quintessential R. It packs together a half-dozen very deep thoughts. It has most of the essential secrets of R's guts, laid out in a single line. It has do.call(), substitute(), it interprets a formula as a list, and it shows that every command in R is, when it comes down to brass tacks, a list.

I would like to take up these separate pieces in order.

2.2 A Formula Object is a List.

While struggling with this, I noticed this really interesting thing. The solution depends on it. The object "newFmla" is not just a text string. It prints out as if it were text, but it is actually an R list object. Its parts can be probed recursively, to eventually reveal all of the individual pieces:

```
> newFmla
```

```
y ~ log(x1) + x2 * x3
```

```
> newFmla[[1]]
```

```
`~`
```

```
> newFmla[[2]]
```

```
y
```

```
> newFmla[[3]]
```

```
log(x1) + x2 * x3
```

```
> newFmla[[3]][[2]]
```

```
log(x1)
```

```
> newFmla[[3]][[2]][[2]]
```

```
x1
```

How could I put that information to use? Read on.

2.3 do.call() and eval()

In my early work as an Rchaeologist, I had noticed eval and do.call, but did not understand their significance in the mind of the R programmers. Whenever difficult problems arose in r-help, the answer almost invariably involved do.call or eval.

2.3.1 do.call()

Let's concentrate on do.call first. The syntax is like this

```
do.call("someRFunction", aListOfArgumentsToGoInTheParentheses)
```

It is as if we were telling R to run this:

```
someRFunction(aListOfArgumentsToGoInTheParentheses)
```

Let's consider an example that runs a regression the ordinary way, and then with do.call. In this example, the role of "someRFunction" will be played by lm and the list of arguments will be the parameters of the regression. The regression m1 will be constructed the ordinary way, while m2 is constructed with do.call.

```
> m1 <- lm(y ~ x1*x2, data=dat)
> coef(m1)
```

```
(Intercept)          x1          x2       x1:x2
296.4693597  -5.7456882  -5.9574971   0.1154519
```

```
> regargs <- list(formula = y ~ x1*x2, data= quote(dat))
> m2 <- do.call("lm", regargs)
> coef(m2)
```

```
(Intercept)          x1          x2       x1:x2
296.4693597  -5.7456882  -5.9574971   0.1154519
```

```
> all.equal(m1, m2)
```

```
[1] TRUE
```

The object regargs is a list of arguments that R can understand when they are supplied to the lm function. do.call() is a powerful, mysterious symbol. It holds flexibility; we can calculate commands and then run them. I first needed it when we had a simulation project that ran very slowly when confronted with medium or large sized problems. There's a writeup in the working examples distributed with rockchalk called stackListItems-01.R. I was using rbind over and over to join the results of simulation runs. Basically, the code was like this

```
for (i in 1:10000){
   dat <- someHugeSimulation(i)
   result <- rbind(result, dat)
}
```

That will call rbind() 10000 times. I had not realized that rbind() is a comparatively time-consuming task because it accesses a new chunk of memory each time it is run. On the other hand, we could collect those results in a list, then we can call rbind one time to smash together all of the results.

```
for (i in 1:10000){
   mylist[[i]] <- someHugeSimulation(i)
}
result <- do.call("rbind", mylist)
```

It is much faster to run rbind only once. It would be OK if we typed it all out like this:

```
result <- rbind(mylist[[1]], mylist[[2]], mylist[[3]], mylist[[4]], ..., mylist[[10000]])
```

But who wants to do all of that typing? How tiresome! Thanks to Erik Iverson in r-help, I understand that

```
result <- do.call("rbind", mylist)
```

is doing the EXACT same thing. "mylist" is a list of arguments. do.call is *constructing* a function call from the list of arguments. It is *as if* I had actually typed rbind with 10000 arguments.

The beauty in this is that we could design a program that can assemble the list of arguments, and also choose the function to be run, on the fly. We are not required to literally write the function in quotes, as in "rbind". We could instead have a variable that is calculated to select one function among many, and then use do.call on that. In a very real sense, we could write a program that can write itself as it runs.

From all of this (and a peek at ?call), I arrive at an Rchaeological eureka! A call object is a quoted command plus a list of arguments for that command.

2.3.2 eval()

Where does eval fit into the picture? As far as I can tell, do.call("rbind", mylist) is basically the same as eval(call("rbind", mylist)). The call() function manufactures the call object, the eval() function tells it to do its work. I think of do.call() as a contraction of "eval" and "call". eval() evaluates any valid R expression, and a call is a valid expression. I'm leaving the question of "what is an expression" to a later time.

Here's a quick example that repeats the two regressions exercise that was completed with do.call. Now I'll create an expression regargs2. Note it is necessary for me to evaluate the expression before the lm function can understand it.

```
> m3 <- lm(y ~ x1*x2, data=dat)
> coef(m3)
```

```
(Intercept)          x1          x2       x1:x2
296.4693597  -5.7456882  -5.9574971   0.1154519
```

```
> regargs2 <- expression(y ~ x1*x2, data = dat)
> m4 <- lm(eval(regargs2))
> coef(m4)
```

```
 (Intercept)           x2           x3           x4            y
37.168385022  0.100178686  0.007236296  0.153423361  0.049381689
```

The main reason for using eval is that we can "piece together" commands and then run them after we have assembled all the pieces.

We can create a formula object implicitly (without explicitly asking for it) by using this code.

```
> f1 <- y ~ x1 + x2 + x3 + log(x4)
> class(f1)
```

```
[1] "formula"
```

```
> m5 <- lm(f1, data = dat)
```

The object f1 is a formula object because R has created it that way. Its not just a text string. R notices the ~ symbol and the whole line is interpreted as a formula. Observe it has separate pieces, just like newFmla in the example problem that started this section.

```
> f1[[1]]
```

```
`~`
```

```
> f1[[2]]
```

```
y
```

```
> f1[[3]]
```

```
x1 + x2 + x3 + log(x4)
```

```
> f1[[3]][[1]]
```

```
`+`
```

```
> f1[[3]][[2]]
```

```
x1 + x2 + x3
```

```
> f1[[3]][[3]]
```

```
log(x4)
```

Note that f1 created in this way must be a syntactically valid R formula; it cannot include any other regression options.

```
> f1 <- y ~ x1 + x2 + x3 + log(x4), data=dat
Error: unexpected ',' in "f1 <- y ~ x1 + x2 + x3 + log(x4),"
```

If I declare f1exp as an expression, then R does not re-interpret it as a formula (f1exp is an unevaluated expression, the R parser has not translated it yet). To use that as a formula in the regression, we have to evaluate it.

```
> f1exp <- expression(y ~ x1 + x2 + x3 + log(x4))
> class(f1exp)
```

```
[1] "expression"
```

```
> m6 <- lm(eval(f1exp), data=dat)
```

When f1exp is evaluated, what do we have? Here's the answer.

```
> f1expeval <- eval(f1exp)
> class(f1expeval)
```

```
[1] "formula"
```

```
> all.equal(f1expeval, f1)
```

```
[1] TRUE
```

```
> m7 <- lm(f1expeval, data=dat)
> all.equal(coef(m5), coef(m6), coef(m7))
```

```
[1] TRUE
```

The point here is that the pieces of an ordinary use command can be separated and put back together again before the work of doing calculations begins.

Now we turn back to the main theme. How is eval() used in functions? Some functions take a lot of arguments. They need to pick some arguments, and send those to some functions.

Let's consider the lm() code in some detail. Suppose a user submits a command like "lm(y ~ x, data = dat, x = TRUE, y = TRUE)." Inside lm(), it is necessary to pick through those arguments and then pass them off to other functions in order to build the data matrix and so forth. Here are the first lines of the lm() function

```
1  lm <- function (formula, data, subset, weights, na.action, method = "qr",
2      model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
3      contrasts = NULL, offset, ...)
4  {
5      ret.x <- x
6      ret.y <- y
7      cl <- match.call()
8      mf <- match.call(expand.dots = FALSE)
9      m <- match(c("formula", "data", "subset", "weights", "na.action",
10         "offset"), names(mf), 0L)
11     mf <- mf[c(1L, m)]
12     mf$drop.unused.levels <- TRUE
13     mf[[1L]] <- as.name("model.frame")
14     mf <- eval(mf, parent.frame())
```

Lets consider what those lines do with a command like this.

```
m1 <- lm(y ~ x1*x2, data=dat, x = TRUE, y = TRUE)
```

The lm function notices that I supply some arguments. In line 8, the match.call function is used to grab a copy of the command that I typed. If we use R's debugging facility to stop the program at that point, we would see that mf is exactly the same as my command, except R has named the arguments:

```
> mf
lm(formula = y ~ x1 * x2, data = dat, x = TRUE, y = TRUE)
```

That's not just a string of letters, however. It is a call object, a list with individual pieces that we can revise. Lines 10 and 11 check the names of mf for the presence of certain arguments, and throw away the rest. It only wants the arguments we would be needed to run the function model.frame. Line 12 adds an argument to the list, drop.unused.levels. Up to that point, then, we can look at the individual pieces of mf:

```
> names(mf) [1] ""                    "formula"           "data" [4] "drop.unused.levels"
> mf[[1]]
lm
> mf[[2]]
y ~ x1 * x2
> mf[[3]]
dat
> mf[[4]]
[1] TRUE
```

The object mf has separate pieces that can be revised and then evaluated. Line 13 replaces the element 1 in mf with the symbol "model.frame". That's the function that will be called. Line 14 is the coup de grâce, when the revised call "mf" is sent to eval. In the end, it is *as if* lm had directly submitted the command

```
mf <- model.frame(y ~ x1 * x2, data=dat, drop.unused.levels=TRUE)
```

It would not do to simply write that into the lm function, however, because some people use variables that have names different from y, x1, and x2, and their data objects may not be called dat. lm allows users to input whatever they want for a formula and data, and then lm takes what it needs to build a model frame.

2.4 substitute()

Most R users I know have not used substitute, except as it arises in the plotmath. In the context of plotmath, the problem is as follows. Plotmath causes the R plot functions to convert expressions into mathematical symbols in a way that is reminiscent of LaTeX. For example, a command like this:

```
text(4, 4, expression(gamma))
```

will draw the gamma symbol at the position (4,4). We can use paste to combine symbolic commands and text like so:

```
text(4, 4, expression(paste(gamma, " = 7")))
```

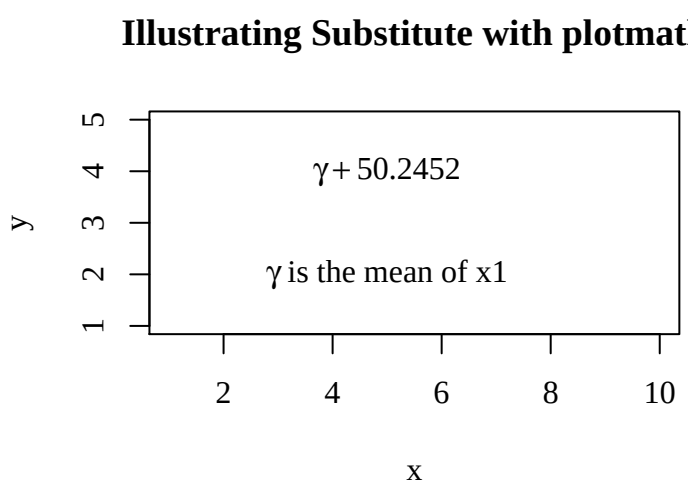
The number 7 is a nice number, but what if we want to calculate something and insert it into the expression? Your first guess might be to insert a function that makes a calculation, such as the mean, but this fails:

```
text(4, 4, expression(paste(gamma, mean(x))))
```

In order to smuggle the result of a calculation into an expression, some fancy footwork is required. In the help page for plotmath, examples using the functions bquote and substitute are offered.

For the particular purpose of blending expressions with calculation results, I find the bquote function to be more immediately understandable. In this section, I'm trying to understand the use of substitute, so let's stick with that. The plotmath help page points to syntax like this:

```
> plot(1:10, seq(1,5, length.out=10), type = "n", main="Illustrating Substitute with plotmath",
    xlab="x", ylab="y")
> text(5, 4, substitute(gamma + x1mean, list(x1mean = mean(dat$x1))))
> text(5, 2, expression(paste(gamma, " is the mean of x1")))
```



Run ?substitute and one is brought to a famous piece of Rchaeological pottery:

> 'substitute' returns the parse tree for the (unevaluated) expression 'expr', substituting any variables bound in 'env'.

Pardon me. parse tree? We've seen expressions already, that part is not so off putting. But "parse tree"? Really?

This is one of those points at which being an Rchaeologist has real benefits. The manual page gives us some insights into the R programmer, and it is his or her view of his or her own actions, but it doesn't necessarily speak to how we should understand substitute(). For me, the only workable approach is to build up a sequence of increasingly complicated examples.

I start by creating the list of replacements. This replacement list can have a format like this:

```
> sublist <- list(x1 = "alphabet", x2 = "zoology")
```

I want to replace x1 with alphabet and x2 with zoology. The quotes indicate that alphabet and zoology are strings, not other objects that already exist. Consider:

```
> substitute(expression(x1 + x2 + log(x1) + x3), sublist)
```

```
expression("alphabet" + "zoology" + log("alphabet") + x3)
```

Note that the substitution 1) leaves other variables alone (since they are not named in sublist) and 2) it finds all valid use of the symbols x1 and x2 and replaces them.

This isn't quite what I wanted, however, because the strings have been inserted into the middle of my expression. I don't want text. I just want symbols. It turns out that the functions as.name() and as.symbol() are exactly the same, and usually I use as.symbol() because that name has more intuition for me, but in Gabor's answer to my question, as.name() is used. Using as.name(), so I will illustrate that.

```
> sublist <- list(x1 = as.name("alphabet"), x2 = as.name("zoology"))
> substitute(expression(x1 + x2 + log(x1) + x3), sublist)
```

```
expression(alphabet + zoology + log(alphabet) + x3)
```

2.5 setNames and names

Almost every R user has noticed that the elements of R lists can have names. In a data frame, the names of the list elements are thought of as variable names, or column names. If dat is a data frame, the names and colnames functions return the same thing, but that's not true for other types of objects.

```
> dat <- data.frame(x1=1:10, x2=10:1, x3=rep(1:5,2), x4=gl(2,5))
> colnames(dat)
```

```
[1] "x1" "x2" "x3" "x4"
```

```
> names(dat)
```

```
[1] "x1" "x2" "x3" "x4"
```

After dat is created, we can change the names inside it with a very similar approach:

```
> newnames <- c("whatever","sounds","good","tome")
> colnames(dat) <- newnames
> colnames(dat)
```

```
[1] "whatever" "sounds"   "good"     "tome"
```

While used interactively, this is convenient, but it is a bit tedious because we have to create dat first, and then set the names. The setNames() function allows us to do this in one shot. I'll paste the data frame creating commands and the name vector in for a first try:

```
> dat2 <- setNames(data.frame(x1=rnorm(10), x2=rnorm(10), x3=rnorm(10), x4=gl(2,5)), c("good","
    names","tough","find"))
> head(dat2, 2)
```

```
       good      names     tough find
1 -1.420324 -1.998493 0.4130209    1
2 -2.466939 -1.234780 0.5642563    1
```

In order to make this more generally useful, the first step is to take the data-frame-creating code and set it into an expression that is not immediately evaluated (that's datcommand). When I want the data frame to be created, I use eval(), and then the newnames vector is put to use.

```
> newnames <- c("iVar", "uVar", "heVar", "sheVar")
> datcommand <- expression(data.frame(x1=1:10, x2=10:1, x3=rep(1:5,2), x4=gl(2,5)))
> eval(datcommand)
```

```
   x1 x2 x3 x4
1   1 10  1  1
2   2  9  2  1
3   3  8  3  1
4   4  7  4  1
5   5  6  5  1
6   6  5  1  2
7   7  4  2  2
8   8  3  3  2
9   9  2  4  2
10 10  1  5  2
```

```
> dat3 <- setNames(eval(datcommand), newnames)
```

The whole point of this exercise is that we can write code that creates the names, and creates the data frame, and then they all come together.

What if we have just one element in a list? In Gabor's answer to my question, there is this idiom

```
setNames(list(as.name("x1c")), "x1")))
```

Consider this from the inside out.

1. as.name("x1c") is an R symbol object,
2. list(as.name("x1c")) is a list with just one object, which is that symbol object.
3. Use setNames(). The object has no name! We would like to name it "x1".

It is as if we had run the command list(x1 = x1c). The big difference, of course, is that this way is much more flexible because we can calculate replacements.

2.6 The Big Finish

In the meanCenter() function in rockchalk, some predictors are mean-centered and their names are revised. A variable named "age" becomes "agec" or "x1" becomes "x1c". So the user's regression formula that uses variables agec or x1 must be revised. This is a function that takes a formula "fmla" and replaces a symbol xname with newname.

```
formulaReplace <- function(fmla, xname, newname){
    do.call("substitute", list(fmla, setNames(list(as.name(newname)), xname)))
}
```

This is put to use in meanCenter(). Suppose a vector of variable names called nc (stands for "needs centering") has already been calculated. The function std creates a centered variable.

```
newFmla <- mc$formula
for (i in seq_along(nc)){
    icenter <- std(stddat[, nc[i]])
    newname <- paste(as.character(nc[i]), "c", sep = "")
    newFmla <- formulaReplace(newFmla,  as.character(nc[i]), newname)
    nc[i] <- newname
}
```

If one has a copy of rockchalk 1.6 or newer, the evidence of the success of this approach should be evident in the output of the command example(meanCenter).

3 Function Arguments (Stub).

While surveying R packages, we see many different styles for handling arguments. This section will discuss some pros and cons of various styles.

How much argument checking is required?

Is it better to default arguments to NULL?

myfn <- function(x = NULL, y = NULL)

3.1 Make Re-Usable Tools (Rather than Cutting and Pasting)

We seek concise solutions that are generalizable. It is almost never right to cut and paste and then make minor edits in each copy to achieve particular purposes. This advice goes against the grain of the graduate students that I work with. They will almost always use cut and paste solutions.

Here is an example of the problem. A person needed "dummy variables". The code had several pages like this:

```
  if(setcorr == 1){
       corr.10 <-1
       corr.20 <-0
       corr.30 <-0
       corr.40 <-0
       corr.50 <-0
       corr.60 <-0
       corr.70 <-0
       corr.80 <-0
    }
    if(setcorr == 2){
       corr.10 <-1
       corr.20 <-1
       corr.30 <-0
       corr.40 <-0
       corr.50 <-0
       corr.60 <-0
       corr.70 <-0
       corr.80 <-0
    }
    if(setcorr == 3){
       corr.10 <-1
       corr.20 <-1
       corr.30 <-1
       corr.40 <-0
```

```
        corr.50 <-0
        corr.60 <-0
        corr.70 <-0
        corr.80 <-0
    }
```

Well, that's understandable, but a little bit embarrassing. I asked "why do you declare these separate variables, why not make a vector?" and "can't you see a more succinct way to declare those things?" The answer is "we do it that way in SAS" or "this runs".

If the mission is to create a vector with a certain number of 1's at the beginning, surely either of these functions would be better:

```
> biVec1 <- function(n = 3, n1s = 1) {
      c(rep(1, n1s), rep(0, n - n1s))
  }
> biVec2 <- function(n = 3, n1s = 1){
      x <- numeric(length = n)
      x[1:n1s] <- 1
      x
  }
> (corr <- biVec1(n = 8, n1s = 3))
```

```
[1] 1 1 1 0 0 0 0 0
```

```
> (corr <- biVec2(n = 8, n1s = 3))
```

```
[1] 1 1 1 0 0 0 0 0
```

The cut and paste way is not wrong, exactly. Its tedious.

I think any reasonable user should want a vector, rather than the separate variables. But suppose the user is determined, and really wants the individual variables named corr.10, corr.20, and so forth. We can re-design this so that those variables will be sitting out in the workspace after the function is run. Read help("assign") and try

```
> biVec3 <- function(n = 3, n1s = 1) {
      X <- c(rep(1, n1s), rep(0, n - n1s))
      xnam <- paste("corr.", 1:8, "0", sep = "")
      for(i in 1:n) assign(xnam[i], X[i], envir = .GlobalEnv)
  }
> ls()
```

```
 [1] "biVec1"      "biVec2"      "biVec3"      "corr"        "dat"         "dat2"        "dat3"
 [8] "datcommand"  "f1"          "f1exp"       "f1expeval"   "m1"          "m2"          "m3"
[15] "m4"          "m5"          "m6"          "m7"          "newFmla"     "newnames"    "regargs"
[22] "regargs2"    "sublist"     "suffixX"
```

```
> biVec3(n = 8, n1s = 3)
> ls()
```

```
 [1] "biVec1"      "biVec2"      "biVec3"      "corr"        "corr.10"     "corr.20"     "corr.30"
 [8] "corr.40"     "corr.50"     "corr.60"     "corr.70"     "corr.80"     "dat"         "dat2"
[15] "dat3"        "datcommand"  "f1"          "f1exp"       "f1expeval"   "m1"          "m2"
[22] "m3"          "m4"          "m5"          "m6"          "m7"          "newFmla"     "newnames"
[29] "regargs"     "regargs2"    "sublist"     "suffixX"
```

I don't think most people will actually want that kind of result, but if they do, there is a way to get it.

If we want to turn biVec1() or biVec2() into general purpose functions, we need to start thinking about the problem of unexpected user input. Notice what happens if we ask for more 1's than the result vector is supposed to contain:

```
> biVec1(n = 3, n1s = 7)
Error in rep(0, n - n1s) (from #2) : invalid 'times' argument
> biVec2(n = 3, n1s = 7)
[1] 1 1 1 1 1 1 1
```

Perhaps I prefer biVec1() because it throws an error when there is unreasonable input, while biVec2() returns a longer vector than expected with no error.

If the user mistakenly enters non-integers, neither function generates an error, and they give us unexpected output.

```
> biVec1(3.3, 1.8)
```

```
[1] 1 0
```

```
> biVec2(3.3, 1.8)
```

```
[1] 1 0 0
```

To my surprise, it is not an easy thing to ask a number if it is a whole number (see help("is.integer")). For that, we introduce a new function, is.wholenumber(), and put it to use in our new and improved function

```
> is.wholenumber <- function(x, tol = .Machine$double.eps^0.5){
   abs(x - round(x)) < tol
  }
> biVec1 <- function(n = 3, n1s = 1) {
      if(!(is.wholenumber(n) & is.wholenumber(n1s)))
          stop("Both n and n1s must be whole numbers (integers)")
      if(n1s > n)
          stop("n must be greater than or equal to n1s")
      c(rep(1, n1s), rep(0, n - n1s))
  }
```

In all of the test cases I've tried, that works, although the real numbers 7.0 and 4.0 are able to masquerade as integers.

```
> biVec1(3, 7)
Error in biVec1(3, 7) (from #4) : n must be greater than or equal to n1s
> biVec1(7, 3)
[1] 1 1 1 0 0 0 0
> biVec1(3.3, 4.4)
Error in biVec1(3.3, 4.4) (from #2) :
  Both n and n1s must be whole numbers (integers)
> biVec1(7.0, 4.0)
[1] 1 1 1 1 0 0 0
> biVec1(7.0, 4.0)
[1] 1 1 1 1 0 0 0
```

4 Do This, Not That (Stub)

R novices sometimes use Google to search for R advice and they find it, good or bad. They may find their way to the r-help email list, where advice is generally good, or to the StackOverflow pages for R, which may be better. A lot of advice is offered by people like me, who may have good intentions, but are simply not qualified to offer advice.

One of the few bits of advice that seems to grab widespread support is that "for loops are bad." One can write an lapply statement in one line, while a for loop can take 3 lines. The code is shorter, but it won't necessarily run more quickly. I recall being jarred by this revelation in John Chambers's book, *Software for Data Analysis*. The members of the apply family (apply, lapply, sapply, etc) can make for more readable code, but they aren't always faster. "However, none of the apply mechanisms changes the number of times the supplied function is called, so serious improvements will be limited to iterating simple calculations many times. Otherwise, the n evaluations of the function can be expected to be the dominant fraction of the computation"(Chambers, 2008, 213).

Todo: insert discussion of stackListItems-001.

Insert alternative methods of measuring executation time and measuring performance

Balance time spent optimizing code versus time spent running program.

5 Suggested Chores

I suggest the would be programmer should take on some basic challenges.

1. Try to create a generic function and several methods to handle classes of various types. If the term "generic function" and "method" cause disorientation, that means the reader is a user, not a programmer yet.

2. Consider developing a routine (or package) for statistical estimation or presentation. Find several R packages and compare the R code in them. Let me know if you agree with me about these points.

- Good code is compartmentalized. Separate pieces of work are handled by separated functions and results are returned in a well organized way.

- Functions should not be HUGE.

  The components of a project should be small enough so that we can comprehend them. Generally speaking, if we are reading code and we come to a line that uses a variable that we cannot find on the screen, that's a problem. I used to correspond with a programmer at the University of Michigan who said he began to feel uncomfortable when a function filled up the entire terminal screen.

- People who copy and paste sections over and over in order to handle slightly different cases are causing trouble for themselves and others. They should think harder on ways to separate that work into re-usable functions.

References

Chambers, J. M. (2008). *Software for data analysis: programming with R.* Statistics and computing. New York ; London: Springer. 4