Rchaeology: Idioms of R Programming

Paul E. Johnson <pauljohn @ ku.edu>

August 26, 2024

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 am getting some traction.

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* (2008), 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.

I usually approach R coding in three phases. I write code that "works", however tedious and slow it might be. There will be a lot of "copy and paste" constructions with tedious, manual editing of commands. Then I read through the code and look for repetitious elements, and I re-organize to make functions that abstract that work. If there are any stanzas that look mostly the same, except that "x1" is repalced by "x2", I know the work is not done. Finally, I look through the code to find "dumb" constructions that can be cleaned up and completed with fewer lines. A dumb construction is one that I would be ashamed to show to an expert R programmer.

Beginning R programmers, the ones who have never studied C or Java or Fortran, will often become "stuck" in stage one of that process. Experienced programmers with more formal training will usually have the "little voice" in the back of their minds saying "there's a better way," and, unfortunately, R novices don't hear that voice. Lacking some of the discipline imposed by

those other languages, the usual "rambling" R script will seem "good enough." The first step for the transitional R programmer, the one who does not want to be a novice any more, is to take a harsh look at the code that has already been written.

The rockchalk package has many functions that receive fitted regression models and re-arrange their components for re-fitting or plotting. A good deal of the advice I have to offer is about the process of managing R formula, but I have also accumulated some lessons in argument handling.

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

If the transitional programmer understands these four functions, she/he will be well on the way to escaping the category of novice R programmer. The other benefit is that the code written by the experts, including the R source code itself, 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. I need to receive a fitted model, extract the formula, change some variables, and then revise the formula to match the new variables. And then run the model over again.

The functions meanCenter() and residualCenter() receive a fitted regression model, transform some variables, and then fit a new regression. Suppose a regression has been fitted as "`y ~ x1*x2`". The R result will estimate a predictive formula such as $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x1_i + \hat{\beta}_2 x2_i + + \hat{\beta}_3 x1_i \cdot x2_i$. meanCenter() will replace the non-centered variables "x1" and "x2" with mean-centered variables "x1c" and "x2c". 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. I learned very quickly 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="",
          collapse=" "))
      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 for assistance. Lately I've had very good luck with r-help. Gabor Grothendieck wrote an answer to r-help on May 29, 2012. He said simply, "Try substitute," with this example.

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

Problem solved, in a single line.

That's very clever, I think. 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 pattern. 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. Coming to grips with these ideas is a critical step in the R programmer's growth, because they separate the "script writer" from the "language programmer." Whenever difficult problems arise in r-help, the answer almost invariably involves 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)
```

We use do.call() because it is much more flexible than calling someRFunction() directly.

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 "on the language" to create commands and then run them. I first needed do.call() when we had a simulation project that ran very slowly. 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 time-consuming. 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[[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? do.call() manufactures a call and executes it immediately. It is possible to arrest the process mid stream, to capture the call without evaluating it with eval(). The terminology in the R documentation on this is difficult, mainly because there are several different ways to create a piece of not-yet-evaluated code. By **not-yet-evaluated code**, I mean an "unevaluated expression" or a "call object", anything that can be handled by the eval() function. A not-yet-evaluated object holds some syntax that can we need later. I think of do.call() as a contraction of "eval" and "call". As far as I can tell, do.call("lm", list(arguments)) is the same as eval(call("lm", arguments)).

There are many ways to ask R to create the not-yet-evaluated object, that's one of the confusing things (in my opinion). Most obviously, call() function can be used for that. But functions like quote() and expression() can also create not-yet-evaluated code.

Everybody has experienced an error like this:

```
> x1 + x2
Error: object 'x1' not found
```

Whenever we use a variable, the R interpreter expects to find it. We know those variables are in the data frame dat, but the interpreter does not know that's what we mean. But we can ask R to trust us with the quote() function, which, basically, means "here's some code we will evaluate later":

```
> mycall <- quote(x1 + x2)
```

The object "mycall" is an R call object, something that might be evaluated in the future.

When we want to evaluate "x1 + x2", we ask R do to so:

```
eval(mycall, dat)
```

The second argument supplies the place where x1 and x2 are to be found. R users may never have noticed the eval() function, but most will have used the with() function, which is simply a clever re-phrasing of the eval function's interface. It does the same thing:

```
with(dat, x1 + x2)
```

and so does

```
with(dat, eval(mycall)).
```

Now, back to the regression example we were working on before. We can create the call object using the call() function, which requires the name of a function and a comma-separated list of quoted arguments. I quote each of the bits of syntax in the arguments because I don't want R to replace "dat" with the actual data frame, I only want it to remember the name.

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

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

```
> mycall <- call("lm", quote(y ~ x1*x2), data = quote(dat))
> m4 <- eval(mycall)
> coef(m4)
```

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

The main reason for using eval is that we can "piece together" commands and then run them after we have assembled all the pieces. The nickname for this seems to be "computing on the language." There is a discussion of it in the R Language Definition. It is also surveyed in Hadley Wickham's new book, *Advanced R* (2015).

Did you ever notice that you get an error if you run "x1 + x2", but if you run "y ~ x1 + x2", then there is no error? It is as if R is wrapping your code in protective tissue, something like quote(). However, the protective wrapper in this case is actually the function called formula(). The R interpreter notices the symbol ~ in the syntax, and so it assumes you meant to run

```
formula(y ~ x1 + x2)
```

A formula is another kind of not-yet-evaluated thing. It is something that we can fiddle with before evaluating.

Create a complicated formula. Note that the lm function receives that formula object without trouble.

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

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

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

```
(Intercept)          x1          x2          x3      log(x4)
26.39386649  0.03117335 -0.12593151 -0.01764095 -5.31745509
```

The object f1 is a formula object. Its not just a text string. 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(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. We can edit the formula. Suppose we replace a part:

```
> f1[[3]][[2]] <- quote(x1 + log(x2))
> m8 <- lm(f1, data = dat)
> coef(m8)
```

```
(Intercept)          x1      log(x2)      log(x4)
42.99241035  0.03109661 -6.30535922 -5.08943743
```

Now we turn back to the main theme. How is eval() used in functions? Some functions take a lot of arguments. We may want to do a lot of fine tuning on the arguments, before they are assembled and evaluated. Hence, it is very important that we can keep some bits of R that are symbolic, rather than calculated, until we need them.

If you are looking for examples of eval() in the R source code, there is a nice one in the beginning of the lm() function. Suppose a user submits a command like "lm(y ~ x, data = dat, x = TRUE, y = TRUE)." Inside lm(), the import is re-organized. 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())
```

Now suppose we turn on the debugger and run a regression in R with a command like this.

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

After that, we are "in" the function, stepping through line-by-line. In line 8, the match.call() function is used to grab a copy of the command that I typed. We can 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 can be revised. Recall that the first element in a call object is the name of a function, and the following elements are the arguments. 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 possibly if they try to use 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")))
```

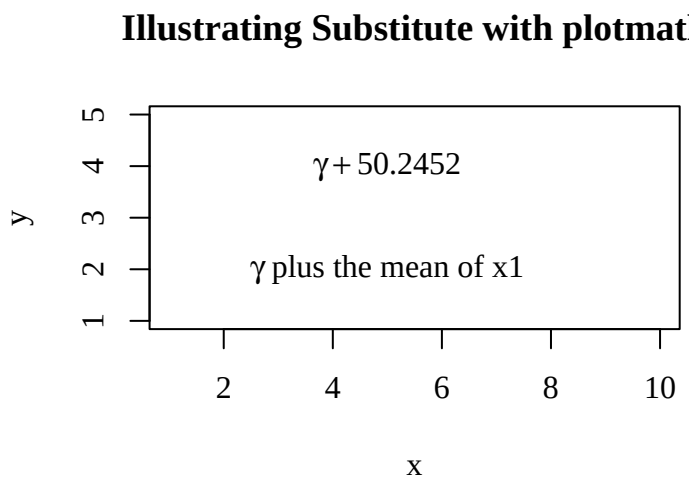
The number 7 is a nice number, but what if we want to calculate something and insert it into the expression? Your first guess might be to insert a function that makes a calculation, such as the mean, but this fails:

```
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. I have found this to be quite frustrating and difficult to do, and if one examines my collection of examples on `http://pj.freefaculty.org/R/WorkingExamples`, one will find quite a few plotmath exercises.

For the particular purpose of blending expressions with calculation results, I find the bquote function to be more easily understandable. However, when going beyond plotmath, I expect most R programmers will need to use substitute() instead, so I will discuss that in this section. 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, " plus 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. We have to dig deeper, to try to understand not only what the R programmer says, but what she is actually trying to do. 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")
```

Suppose I have some other object in which x1 and x2 need to be replaced. Specifically, when an expression has "x1", I want "alphabet", and "x2" should become "zoology." The quotes in the R code indicate that alphabet and zoology are character strings, not other objects that already exist. Consider replacing x1 and x2 in x1 + x2 + log(x1) + x3:

```
> 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 uses 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. I've seen the conversion from character to symbol done with as.symbol() and with as.name(), recently I realized that they are synonymous. I usually 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(), my example would become:

```
> 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 in what follows). 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 age 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 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". One might suspect that the author was eager to say, "my project is based on 10,000 lines of R code" in order to impress a lay audience.

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"          "m8"          "mycall"      "
    newFmla"
[22] "newnames"    "regargs"     "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"          "m8"          "
    mycall"
[29] "newFmla"     "newnames"    "regargs"     "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 Function Arguments.

While developing functions for the rockchalk package, one of the most important learning experiences I've had is in argument management.

One of the reasons that it is difficult to design functions is that there is no "mandatory manual" on this. The R framework allows programmer creativity, which is a good thing. But it is also a bad thing, since every programmer has his/her own opinion about what ought to be done.

To state the obvious, argument names should be clear, unambiguous, convenient, short, and minimally necessary.

Beyond that, what can we say? I urge new R programmers to study the R source code. This is part of the Rchaeological method. Study the most frequently used functions in the R distribution. I've learned the most from the way arguments are handled in the functions `hist()`, `lm()`, `plot()` and `termplot()`. These functions have had the most eyes on them and, therefore, they are most likely to use the best approach. I wrote several functions that eventually went into rockchalk before I realized that important fact and I regret it (on a weekly basis).

In this section, I offer this advice.

1. Protect your function's calculations from the user's workspace.
2. Check argument values.
3. Design the function so that it runs with a minimum number of arguments.

4.1 A few tidits about arguments

R includes several functions that are very helpful in dealing with arguments.

missing() Inside a function, such as `plotme()` we often need to check if the user supplied a value for an argument, such as `col`. If the user did not supply a value, we might construct one from the other input

```
if (missing(col)) col <- someTediousCalculation(y, z)
```

This is usually good enough, but there is danger that the user might have called the function like so:

```
plotme(x, y, z, col = NULL)
```

This outsmarts missing(col), which will return FALSE. If our goal is to make sure col is something, some non-NULL thing, a more careful argument check would be

```
if (missing(col) | is.null(col)) col <- someTediousCalculation(y, z)
```

stop() Halts a function and returns a message to the user. Usage example

```
if (isTRUE(x)) stop(paste("I'm sorry, you failed to supply the right information.")
    )
```

stopifnot() It is so frequent to run "`if(isTRUE(x)) stop()`" that they created a wrapper, `stopifnot(x)`.

warning() If you don't want the function to fail, but you need to warn the user about a condition that seems dubious, run

```
if (isTRUE(x)) warning(paste("Are you sure the value of x ought to be", x, "?"))
```

deparse(substitute()) Inside a function, it is often important to find out the symbolic value of the argument that the user supplied to the function. I don't mean the *value*, but rather the symbolic name. To see what I mean, run this

```
> dat <- data.frame(myx = c(1, 2, 3), myy = c(10, 5, 1))
> debug(plot.default)
> plot(dat$myx, dat$myy)
```

After you hit enter, the R debugger/browser will come open and the prompt will allow you to inspect the situation. First, type "n" and hit return. That takes the first step into the calculations. After that, run `match.call()`. The function is somewhat "self aware". It knows how it was called. You should see this

```
Browse[2]> n
debug: localAxiis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)
Browse[2]> match.call()
plot.default(x = dat$myx, y = dat$myy)
```

You asked for `plot()`, R sends the work to `plot.default()`. All is well.

Still inside the debugger now, type "x" and then "myx":

```
Browse[2]> x
[1] 1 2 3
Browse[2]> myx
Error: object 'myx' not found
```

Inside the function, the vector value that we passed in, which was referred to as dat$myx, is now called x. The thing x is a data holder, it is used in the following calculations. `x` does not remember that its passed-in value was named `dat$myx` in a previous life. It is as if `x` is the new identity of the same old data vector. However, we can ask `x` "what name were you known by in your previous life?" with this interesting idiom.

```
Browse[2]> deparse(substitute(x))
[1] "dat$myx"
```

The substitute() function, as we have already seen, is a way to replace a letter x with a value from somewhere. Maybe it would be more obvious to run this in 2 steps.

```
Browse[2]> xsubs <- substitute(x)
Browse[2]> xsubs
dat$myx
Browse[2]> deparse(xsubs)
[1] "dat$myx"
```

`substitute()` grabs the object name from whence `x` was calculated. We need to convert that back to a character string so that we can know the symbolic name of the thing that eventually became `x`. This is at the outer boundary of my R comprehension, so I better stop here.

Incidentally, you can go exploring in the debugger. Run some standards like `ls()` to see what's available. If you keep hitting return, the function will run line by line. If you get tired of hitting return, type "c" and the function will run to its conclusion. To let R know that you don't want to debug that function any more, run:

```
> undebug(plot.default)
```

If you don't do that, then the function browser will pop open every time you try to run plot.

list(...) The three periods, "...", are a valid word. (A period is a valid character). R scans user syntax for function arguments and it uses whatever it can understand (formal named arguments, positional matching), and then the other arguments are thrown into a thing called "...".

In a simple case, we could just pass those additional arguments to another function, as in

```
plotme <- function(x, y, ...){
    plot(x, y, ...)
}
```

Often, it is not sufficient to simply pass on the "..." variables, and we instead need to scan them and sort them for various purposes. Inside the function, we access those variables by turning them into a list:

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

We can still pass on the dots to other functions, but the usual problem is that some of the arguments in dots may be intended for several different functions. Suppose we are writing a plotter for categorical variables. We might need to receive some arguments, then run `table()`, then run `layout()`, then run `mosaicplot()`. The user may have put in arguments intended for those separate functions. There's a chance of an error, or at least a warning, if irrelevant arguments are passed to those arguments.

The rockchalk functions `plotSlopes()` has examples that do this work. They separate the dot arguments that are needed for the `predict()` function and the `plot()` function. Because I was learning about arguments when I wrote those functions, I did not take the cleanest or most direct route to do this, but they do stand as working examples that users can review.

A function may set a number of defaults that we want to use when the users do not supply a lot of details. In a case like that, it is useful to remember a very handy function in R called `modifyList()`. The basic idea is that we can build a list of default arguments for a function, and then we can scan the `dots` list and it will replace the defaults in the first with changes in the second. There's an example in rockchalk's `pctable()` function.

```
dots <- list(...)
dotnames <- names(dots)
tableargs <- list(rv, cv, dnn = c(rvlab, cvlab))
newargs <- modifyList(tableargs, dots, keep.null = TRUE)
t1 <- do.call("table", newargs)
```

The dots list, by definition, is named arguments, and it is often handy to have a copy of their names for checking whether an argument is included in the list. In the plotSlopes() function, here's what I ended up with:

```
validForPredict <- c("se.fit", "dispersion", "terms", "na.action",
                     "level", "pred.var", "weights")
dotsForPredict <- dotnames[dotnames %in% validForPredict]
if (length(dotsForPredict) > 0) {
    parms <- modifyList(parms, dotargs[dotsForPredict])
    dotargs[[dotsForPredict]] <- NULL
}
np <- do.call("predictCI", parms)
```

This scans dotnames for elements in the list of arguments that are valid for the predict method, and after they are put onto the parms list, then those arguments are set to NULL in the dotargs collection. Looking at that now, I see it would have been more clear to use `setdiff()` on dotargs to clean that up.

The central challenge in that work is to find out the names of the arguments that are intended for one function, then remove them from `dots` so that they are not passed to a subsequent function that needs them. How can one know what arguments are needed? This requires some experience, but the first step is to read the interface documentation of the function being considered. A shortcut is to run the function `formals()`, which prints out the names of the function being considered. This can sometimes be deceptive for implementations of generic methods. Observe that the return from the generic is not so informative as the return from the method:

```
> formals(plot)
```

```
$x


$y


$...
```

```
> formals(plot.default)
```

```
$x


$y
NULL

$type
[1] "p"
```

```
$xlim
NULL

$ylim
NULL

$log
[1] ""

$main
NULL

$sub
NULL

$xlab
NULL

$ylab
NULL

$ann
par("ann")

$axes
[1] TRUE

$frame.plot
axes

$panel.first
NULL

$panel.last
NULL

$asp
[1] NA

$xgap.axis
[1] NA

$ygap.axis
[1] NA

$...
```

If a method is not exported, if it does not show as "visible" in the output from `methods(plot)`, then a little more fancy footwork is required to see all of the legal arguments.

```
> formals(graphics:::plot.histogram)
```

```
$x


$freq
equidist

$density
NULL

$angle
[1] 45

$col
[1] "lightgray"

$border
NULL

$lty
NULL
```

```
$main
paste("Histogram of", paste(x$xname, collapse = "\n"))

$sub
NULL

$xlab
x$xname

$ylab


$xlim
range(x$breaks)

$ylim
NULL

$axes
[1] TRUE

$labels
[1] FALSE

$add
[1] FALSE

$ann
[1] TRUE

$...
```

4.2 Protect your function's calculations from the user's workspace.

In the early days of using R, around 2000, it was possible for a user to obliterate base functions. I recall working on a model about Democratic and Republican voters. I created a variable named `rep` and, after that, the `lm()` and `plot()` functions would not work. I wrote to r-help about this and learned that my `rep` had obliterated the built-in function `rep()`, which is vital in many calculations. Shortly after that, R was improved to protect functions from obliteration in that way. Today I tried to cause mischief by re-defining rep in a number of ways and it appears to me the problem is completely solved.

There has also been a problem that users sometime use the variable names `T` and `F`. Recall that R allows `T` as an abbreviation for `TRUE` and `F` for `FALSE`. R will allow the user to re-define T and F in ways that will break calculations. However, R will not allow the user to re-define TRUE or FALSE. Observe

```
> TRUE <- 2342
Error in TRUE <- 2342 : invalid (do_set) left-hand side to assignment
> FALSE <- function(x,y){x + y}
Error in FALSE <- function(x, y) { :
  invalid (do_set) left-hand side to assignment
```

In order to protect our function's calculations, we are urged to fully write out the symbols TRUE and FALSE inside functions. This way, users cannot cause damage.

Another danger we need to be concerned about is that our functions might have undefined variables in them. When R encounters these undefined variables, the lexical scoping process looks outward for values. It may find something we don't expect. Another danger is that a function seems to work right when we practice it in our session, but another user gets an error because something in our workspace is missing from theirs. The lexical scope can be convenient, but it is also dangerous.

Here the example function relies on variables `y` and `z` that are not passed in as arguments.

```
x <- c(1, 2, 3, 4)
y <- c(8, 9, 10, 11)
z <- c(1, 1, 2, 2)
plotme <- function(x){
    plot(x, y, col = z)
}
plotme(x)
```

That lets the function go and find `y` and `z`. This will run, but it is pretty risky. It depends on R's willingness to reach outside the boundaries of the function to find `y`, and `z`. If I have inadvertently altered `y` in the workspace, this will end badly.

If I'm writing a function for somebody else to use, as in a package, I believe the function should to NOT rely on variables from the environment. I want to prevent R from filling in the gaps for users. We'd rather give the user an error "variable not found" than an erroneous calculation that "seems" adequate. The best case scenario is that `y` and `z` do not exist in the workspace, so the user will receive an error message:

```
> plotme(x)
```

```
Error in xy.coords(x, y, xlabel, ylabel, log) : object 'y' not found
```

The worst case scenario is that the user has some other variables `y` and z sitting about in the workspace. Those "found" variables are put to use with a completely unintentional result.

As a first step toward fixing this, I propose:

> Every piece of input upon which we rely should be named in the list of arguments.

If we define the function as follows,

```
plotme <- function(x, y, z){
    plot(x, y, col = z)
}
```

the user's use of `plotme(x)` will result in the error

```
Error in xy.coords(x, y, xlabel, ylabel, log) :
    argument "y" is missing, with no default
```

Naming all of the arguments effectively protects the function against the accidental importation of data.

There's a function called `checkUsage()` in the `codetools` package that can help check functions for reliance on global variables. First, we'll make sure the workspace is clean by removing `x`, `y`, and `z`.

```
> rm(x, y, z)
> library(codetools)
> checkUsage(plotme)
<anonymous>: no visible binding for global variable 'y' (:2)
<anonymous>: no visible binding for global variable 'z' (:2)
```

Note that, if the variables `x`, `y`, an `z` exist in the user workspace, they are global and `checkUsage` will not offer a warning about them. Hence, it is very important to run `checkUsage()` after cleaning up the workspace.

checkUsage() will also offer warnings about unused variables. This can be helpful in cleaning up a function.

4.3 Check argument values.

We want users to provide input arguments for all information that the calculations will rely upon. The next step is to check their inputs to make sure they are reasonable.

In some cases this is very easy. If we want the user to supply a data.frame, we can stop when the user provides something else.

```
> plotme <- function(x, y, data, ...){
      if(missing(data) | !is.data.frame(data))
          stop(paste0("plotme: the object you supplied as data: '",
          deparse(substitute(data)) , "' is not a data.frame"))
      plot(as.formula(paste(y, "~", x, collapse=" ")), data = data, ...)
  }
> myx <- rnorm(10)
> myy <- rnorm(10)
> dat <- data.frame(myx, myy)
```

This works properly

```
> plotme("myx", "myy", data = dat)
```

However, if the user supplies something else where data ought to be, an error is returned.

```
plotme("myx", "myy", rnorm(10))
```

```
Error in plotme("myx", "myy", rnorm(10)) :
  plotme: the object you supplied as data: 'rnorm(10)' is not a data.frame
```

The main point here is that if we definitely know what we expect the user to supply, then we can add a filter that checks the input's type. That's what is.character, is.vector, is.integer, and so forth are intended for. (In an R session, type "is" and then hit the TAB key a couple of times. You'll see all the completions.

The problem is not so easy to manage when we can't be sure what the argument type must be. Sometimes, it appears the best we can do is pass through what the users give us and hope R can make sense out of it. I've wrestled with R's flexibility for the naming of colors. Suppose our plotme() function displays x and y, and we allow the user to specify colors for the points.

```
> plotme <- function(x, y, z){
      plot(x, y, col = z)
  }
> myx <- rnorm(10)
> myy <- rnorm(10)
```

R colors can be supplied in many formats, as integers, character strings, or more elaborate HTML-style color designators. Any of these will work.

```
> mycol <- 1:10
> plotme(myx, myy, z = mycol)
```

```
> mycol <- rainbow(10)
> plotme(myx, myy, z = mycol)
```

```
> mycol <- gray.colors(10)
> plotme(myx, myy, z = mycol)
```

R does not throw an error if the user runs this, but the result is not what we wanted, probably.

```
> mycol <- rnorm(10)
> plotme(myx, myy, z = mycol)
```

Suppose the user's code ends up supplying a NULL value for `z`, as in `plotme(x, y, z = NULL)`. They would not do that on purpose, but it might be an accident. If `z` is calculated in their code and something has gone wrong. To protect against that, we find this idiom is used in the R source code quite often:

```
> plotme <- function(x, y, z){
      if (missing(z) | is.null(z)) z <- rep(2, length(x))
      plot(x, y, col = z)
  }
```

If `z` is either not supplied by the user's command, or if the user's command explicitly sets it to NULL, then we'll fabricate `z`.

4.4 Design the function so that it runs with a minimum number of arguments.

Take a look at the most commonly used functions in R, such as lm() or plot.default(). These functions allow a great many arguments, but they will deliver what the users minimally need if the they run

```
> m1 <- lm(y ~ x, data = dat)
> plot(y ~ x, data = dat)
```

Put the arguments that users will use most often at the front of the argument list, and then design the function so that it will work even if all of the other arguments are not included. There are two elements in making this work.

4.4.1 Specify defaults

A graph can use any color scheme, but we might put in something colorful for users who like a little zest in life.

```
> plotme <- function(x, y, z = rainbow(length(x))){
      plot(x, y, col = z)
  }
> plotme(x = rnorm(100), y = rnorm(100))
```

We could as well have put the rainbow call inside the function if we want to keep a cleaner function declaration.

```
> plotme <- function(x, y, z){
    if(missing(z)) mycol <- rainbow(length(x))
    plot(x, y, col = mycol)
  }
> plotme(x = rnorm(100), y = rnorm(100))
```

It seems to me that the R source itself tends to have more elaborate specifications of defaults within the declaration, rather than inside the function.

In the R source code, I noticed a peculiar-looking example in which an argument's default depends on a function that is within the function itself. Observe that this runs fine:

```
> plotme <- function(x, y, z = getAColor(length(x)))
  {
      getAColor <- function(n){
              gray.colors(n)
      }
      plot(x, y, col = z, cex = 5)
  }
> plotme(rnorm(20), rnorm(20))
```

That was startling to me because the nested function, `getAColor`, does not exist outside the function, and yet it can be used in the definition of a default. I merely promise that `getAColor` will exist when the value of `z` is needed and all is well.

4.4.2 Extract or construct what else is needed from the user input

We can deduce information from the arguments that are passed in, we can eliminate the need for the user to pass in other arguments. We would never need to require both arguments x and xlab in the plot function, for example. xlab is optional because, if the user does not supply it, then deparse(substitute()) can find something to use as a label. The 2010 version of plotSlopes in rockchalk required the user to pass through many parameters that (I've since learned) can be deduced from the other arguments.

A key element in this is getting an understanding of the information that comes in with a standard R object. Quite a few–almost all–functions that generate interesting objects in R provide, as a part of their output, a copy of the call that ran the function. Inside our functions,

we can inspect that call argument to see what the users ran, and from that we can often deduce what we need without requiring users to supply a lot of information.

4.5 Return values versus attributes

If you review the R regression functions, like lm() or glm(), you notice that they build a list of things and pass it back to the user. That is the most obvious way to return information.

There is another way, however, which is a little interesting. One can mark an attribute on a return object. The R usage of attributes comes to mind at this point because, when we are scanning for inputs, we may need to inspect not only their values, but also their attributes. The most commonly inspected attribute, of course, is the class of the argument. So, if we are writing a package and we need to make sure that the objects being passed around among functions are correct for the purpose, then the object attributes are a very useful tool.

I don't want to write a lot about this so as to display my ignorance, but I would point user to `pctable()` as an example of a rockchalk function that uses both a return list and attributes to get the job done. Maybe I'll fill in this example at a later time, once I figure out the most direct way to explain it.

The only problem I find with attributes is that the printing functions for R objects generally "splat out" the attributes when we don't want them. If one puts attributes on an object, then the default printer will show them. I've not learned all of the ins-and-outs of preventing that, but in 2015 I became aware of it and learned that one way to deal with it can be found in the R source code for the method `print.table()`.

5 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.

6 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. 1, 5

Venables, W. N. and B. D. Ripley (2000). *S programming.* Statistics and computing. New York: Springer. 1

Wickham, H. (2015). *Advanced R.* CRC Press. 2.3.2