# Rchaeology

Paul E. Johnson <pauljohn @ ku.edu>

June 18, 2012

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.

## Contents

**1 Style Guides (or the Lack Thereof) 1**

**2 R Idioms. What's In R Guts? 9**

2.1 Rewriting Formulas. My Introductory Puzzle. . . . 9
2.2 do.call and eval . . . 10
2.2.1 do.call . . . 10
2.2.2 eval . . . 11
2.3 substitute . . . 14
2.4 setNames and names . . . 15
2.5 The Big Finish . . . 17

**3 Do This, Not That (Stub) 17**

# 1 Style Guides (or the Lack Thereof)

The R Core Team has not been eager to write out an exhaustive formal list of criteria that define "good R style." I believe there are many different opinions about how to name functions and variables. The *R Internals* section "R coding standards," is quite brief. Nature abhors a vacuum, as they say. Many others have seen fit to try to fill in the gaps (Google R style guide[1]; Hadley Wickham's Style Guide[2]). In my R

[1] `http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`
[2] `https://github.com/hadley/devtools/wiki/Style`

group at the University of Kansas, we have sought to do the same. Aaron Boulton surveyed these efforts and we developed some guidelines for our group.

It is important to remember the following. Any advice that does not come directly from the R Core Team has no "authority." I have advice, Hadley has advice, but none of it can be used as a trope with which to bludgeon someone who does things differently.

Another important point is that although the official requirements for R code are not stated in all-encompassing detail, there are generally accepted guidelines on what code ought to "look like." Code writers can make ugly code that "runs," but they can't compel anyone else to read it. With few exceptions, coding style is not about making things "work," it is about making them work in a way that is understood by the widest possible audience.

Generally speaking, according to the R team, what should your code look like? Here are the two best answers I have.

1. Open the source code for R itself, and navigate to the directory src/library/stats/R. Open the file "lm.R". There's your answer.

2. Open an R session, run the following commands

   ```
   > lm
   > summary.lm
   > stats:::print.summary.lm
   ```

   There's your answer again.

If I'm looking over your shoulder as you write code, I may say "ach, don't do that" or "nobody is going to want to look at that." You may say, as many students have, "but this works!" And all I can say in response is, "lucky for you, you don't need help fixing it. Because nobody will want to help that." Sometimes ugly code runs, but it is hard to understand, hard to debug.

From my expertise as an Rchaeologist, I have accumulated a list of bits of style advice. These proceed in order, from things that every knowledgeable expert will accept, to matters of personal taste that are more widely accepted, to things that I like, but nobody else does. In the remainder, I'm going to try to help the reader sort between my advice and the advice of "good" R programmers by assigning subjective probabilities of agreement for each of these points. If we could draw a random R programmer from the set of programmers that I admire, what is the probability that the programmer would agree with this advice? Let's call that the "Subjective and completely unscientific personal Estimate of Agreement," or SEA.

1. **(SEA 1.0) Indentation of sections is required.** This is one of the few guidelines that is explicitly spelled out in the R documentation from the core team. They discourage the use of the tab key for indentation, instead suggesting 4 blank spaces. Personally, I prefer 2 spaces, and until 2011 that is what I used. The documentation provides a brief example of startup code for Emacs so that indentions are set correctly at 4 spaces. Otherwise, Emacs will provide 2 spaces while editing R files.

2. **(SEA .98) Blank spaces around symbols are required.** Put blank spaces on both sides of assignment symbols, equal signs and mathematical symbols like "<","*","+", and so forth. Put one space after commas. This is purely a matter of convention and judgment, it does not affect the "rightness" of code. But every finished program by a well-qualified programmer will do this. While developing code, it sometimes helps me to leave spaces inside parentheses and squiggly braces. It helps me keep the logic straight. The experts never leave those spaces in their final version, however, and I try to remember to fix them.

   I believe 99.8% of the R programmers that I admire follow those first two standards. I believe that 95% of my favorites adhere to these principles for the placement of squiggly braces.

3. **(SEA .90) Place squiggly braces ("{" and "}" carefully.** This is partly a matter of appearance, but there are some conditions under which code will break if the braces are not placed properly.

   In R, the opening squiggly braces, "{" should be at the end of the line of code, rather than at the beginning of the next line. This is recommended with for and if statements. And it is vital in if/else statements.

In the C language, this is known as the "K&R style" (named after Kernighan and Ritchie, 1988). Here is an example of some code that follows the guideline, and it works.

```
> x <- 1
> if (x < 10) {
  print("hello")
  } else {
  print("goodbye")
  }
[1] "hello"
```

Programmers who don't follow this guideline should expect trouble. The trouble arrives in two flavors.

(a) **The unexpected else problem**. Sometimes the "else" part of an if/else statement will be "broken" if R does not realize that a command is continuing. Try this at the command line.

```
> if (x < 10) print("hello")
[1] "hello"
> else print("goodbye")
Error: unexpected 'else' in "else"
```

The else is not understood because it is not tied to the if statement. If we keep the braces on a line with the else, the danger is eliminated.

This is the problem that the help page ?if is referring to when it says, "In particular, you should not have a newline between '}' and 'else' to avoid a syntax error in entering a 'if ... else' construct at the keyboard or via 'source'. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for 'if' clauses."

This is a confusing area because sometimes code will work if the else is "out in the open" without any squiggly braces nearby. For example, when an if/else is wrapped inside a larger structure–say a function–then R will correctly interpret else on a line by itself. This works

```
> myfn <- function(x){
      if (x < 7)
      {
          print("x is less than 7")
      }
      else
      {
          print("x is excessive")
      }
  }
> myfn(3)
```

```
[1] "x is less than 7"
```

```
> myfn(88)
```

```
[1] "x is excessive"
```

The danger, however, is that code in that format cannot be run line-by-line, so developing it by running the individual lines will always result in failure. The else is disconnected from the if, when each individual line is executed in R.

As a result, I believe one is well advised to take the defensive approach that is mentioned in the help page and write like so:

```
if (x < 7){
    print("so far, so good")
} else {
    print("this is the "defensive" style mentioned in the R documents")
}
```

(b) **The accidental breakage of else statements**.

It is not necessary to use squiggly braces at all. We can write an if/else statement all on one line if we want to.

```
if (x < 7) print("so far, so good") else print("this is else")
```

This would be legal. In fact, it is the method in which if and else are documented in ?if.

We can also "dangle" the else at the end of the if statement, without any squiggly braces. R knows that there is more coming on the next line.

```
if (x < 7) print("so far, so good") else
print("this is else")
```

There is some danger in that way of writing. Suppose we want to add another command to be performed within the scope of the else command. Add something on (carelessly):

```
if (x < 7) print("so far, so good") else
print("this is else")
print("and this other thing if else is true")
```

That's broken. But every programmer I know has done it, at least once. The first print will run only when else is true, but the second one runs all the time. This will work properly:

```
if (x < 7) print("so far, so good") else{
    print("this is else")
    print("and this other thing if else is true")
}
```

If we forget the squiggly braces, the logic of the situation will not hold up. Thus, especially while developing and testing code, I insert squiggly braces even when they are not required. The squiggly braces reduce the chance that I will make a mistake while revising this code. Perhaps, when a program is done, I'll go back and "tighten it up" so that I use fewer lines (and won't look so much like a novice).

**Summary of points 1-3**

The advice so far mostly concerns "white space" in code. A programmer's text editor, such as Emacs, will generally have built-in functionality to correctly indent sections. It may automatically insert spaces. It generally will not re-position the squiggly braces for us, however.

There is a recently introduced R function that can do all three of these chores. This function, which is called "tidy.source", is available in the "formatR" package (Xie, 2012). The style that is followed in formatR is not "officially sanctioned," but in my experience, it does very well. I've put it to the test with some very ugly tangles of code that students have submitted and it works well. One of its very useful features is that it scans the input and refuses to re-format when there are coding errors. This helps in proof-reading student projects.

The tidy.source function can handle code files, but there is a quick "clipboard copy" feature. Below I've pasted in part of an Emacs session. I wrote a badly formatted function myfn, and copied it to the clipboard, and then tidy.source() reads the clipboard. It seems like magic!.

```
> myfn <- function(x){ if (x < 7) {i = 77; print(paste("x is less than 7 but i is", i))} else {
    print("x is excessive") }}
> library(formatR)
> tidy.source()
function(x) {
    if (x < 7) {
        i = 77
        print(paste("x is less than 7 but i is", i))
    } else {
        print("x is excessive")
    }
}
```

By adding the source parameter, a file name can be provided.

That output is not quite right, in my opinion, because it allows the equal sign for assignment of the variable i. However, tidy.source has an option to correct that.

```
> tidy.source(source = "clipboard", replace.assign = TRUE)
function(x) {
    if (x < 7) {
        i <- 77
        print(paste("x is less than 7 but i is", i))
    } else {
```

```
        print("x is excessive")
    }
}
```

I believe that the style advice to this point will be almost universally supported, or at least understood and accepted. There are some variations in the code from various projects, but the differences don't generally result from a philosophical disagreement with these white-space rules, but rather from differences in text editors.

Now we begin to consider some issues that are more subjective.

4. How to name functions. This is a difficult area because many styles are legal, but some are more easily understood. The programmer's experience may affect whether code looks "right" or not. It is also difficult because R syntax has changed over the years, and some things that were illegal are now allowed.

   (a) (.98 SEA) Avoid using names that are already in use, especially common ones. Don't write functions named "rep" "seq" "lm", and so forth. Don't do this, even though R (since 2.14) has a graceful mechanism to tolerate duplicated names. All functions exist within packages, so one could run base::rep to get the built-in version of rep, while using rep for a package-specific function. Doing that will likely to make code very difficult for the experts to read because (almost always) they assume "seq" is "seq" from base. Why confuse the R programmers by making a new function "seq" that does something different? Pick a new name.

   (b) (.65 SEA)Use periods to indicate classes, otherwise don't use periods in function names. Instead, use camel case to name functions. myFunction or getCalculatedValues are better function names than my.thing or get.calculated.values. A camel cased function may be ugly in the eyes of some, but it will never send the reader searching for a class called "calculated.values" or "thing".

   This is my opinion, but it is not unanimously shared. I bet one-half or two-thirds of the R programmers that I admire would agree. As a spot check, consider two of my favorite packages, MASS and car. There are not many camel case function names in the MASS package (Venables and Ripley, 2002), which is distributed with R. The preferred style in MASS is to give functions brief, all lower case letters, such as "boxcox." Contrast that with the car package (Fox and Weisberg, 2011), which is very widely used, which has a similar function called "boxCox". Some time ago, Professor Fox systematically revised car to change the period-style function names to camel case. If those two packages are counterbalancing each other in my mind (for and against camel case functions), the leading packages for mixed effects models, nlme (Pinheiro et al., 2012) and lme4 (Bates, Maechler and Bolker, N.d.), weigh in on "my side".

   Why would a period in a function name be distracting to some readers? Some readers are trained in Java or C++. In those languages, the period is a "connective" symbol that can join an object name with its variables or member functions. In Java, one would write myThing.x to extract a variable x from an object myThing. When I read R code, I am still (after 10 years) distracted by periods in function names for this reason.

   In R, the period is not used to extract variables, instead we write (in S3) myThing$x or (in S4) myThing@x. However, in R we do use the period as connective tissue between a generic function name and the type of object to which it must be applied. The "full name" of a function will often include a suffix, even though the R interface tries to conceal those suffixes from the readers. Observe the output from the methods function, which lists the class-specific methods, but indicates that they are not exported for easy use.

```
> methods("confint")
[1] confint.default confint.glm*    confint.lm*     confint.nls*

   Non-visible functions are asterisked
> methods("summary")
 [1] summary.aov             summary.aovlist         summary.aspell*
 [4] summary.connection      summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*           summary.factor
[10] summary.glm             summary.infl            summary.lm
[13] summary.loess*          summary.manova          summary.matrix
[16] summary.mlm             summary.nls*            summary.packageStatus*
```

```
[19] summary.PDF_Dictionary* summary.PDF_Stream*      summary.POSIXct
[22] summary.POSIXlt         summary.ppr*            summary.prcomp*
[25] summary.princomp*       summary.srcfile         summary.srcref
[28] summary.stepfun         summary.stl*            summary.table
[31] summary.tukeysmooth*
   Non-visible functions are asterisked
> methods("predict")
 [1] predict.ar*                predict.Arima*
 [3] predict.arima0*            predict.glm
 [5] predict.HoltWinters*       predict.lm
 [7] predict.loess*             predict.mlm
 [9] predict.nls*               predict.poly
[11] predict.ppr*               predict.prcomp*
[13] predict.princomp*          predict.smooth.spline*
[15] predict.smooth.spline.fit* predict.StructTS*
   Non-visible functions are asterisked
```

Observe that when the user runs "confint(myThing)" or "summary(myThing)", the R system has to select one particular method with which to get the work done. R checks for the class of myThing. If the class is "glm", and there is a function confint.glm, summary.glm, or so forth, then those specific methods are used to answer the user's request. If there is no class-specific method available, then the work is sent to confint.default or summary.default.

Many R commands create new objects of particular types that require specialized processing. The command

```
m1summ <- summary(m1)
```

creates a new object from the class "summary.lm". And in order to show that result to the user, the R system uses a function called "print.summary.lm," but the user would ordinarily not notice that a specialized print function exists. The same result is produced by these three commands:

```
m1summ
print(m1summ)
stats:::print.summary.lm(m1summ)
```

The first two are equivalent because typing an object's name is always understood as a request for a print function. The R design discourages us from using the last approach. We are supposed to let the R system select methods to match the classes of the objects being processed. That's why the print.summary.lm function is not exported from the stats package, and thus it is necessary to use the three colons when I want to access it directly.

The whole point is that, inside the R system, the periods are not just punctuation. They may indicate the class type of the object that is being received. Thus, I avoid gratuitous periods in function names.

Admittedly, many programmers, especially the ones who are trained in C++, find camel cased functions to be very ugly. On the other hand, Java and Objective-C programmers are used to them, and may even find them attractive. But the key point is that periods are distracting.

It seems to me that new functions introduced in R tend to have either camel case or underscores for punctuation. Run ?get_all_vars. That's an eye-opener. Some parts of R, especially the old parts, were developed before "object oriented" programming had come to the forefront and, as a result, they do not comply with this advice. However, newer functions generally do. Observe ?browseVignettes. If I knew how to use SVN very well, I'd download the R-devel code and then list all functions by the date on which they were introduced. I will bet a cup of coffee with anybody that the probability of camel cases is higher in the more recent commits. I also would bet that the chance of finding a period in a function name for purely punctuation reasons is almost zero in the most recent commits. Instead of periods, we find either short lower case names, camel case, or punctuation with underscores. I hasten to admit that the underscores look really strange to me!

5. **How to name variables** (and objects and other things you need to keep track of).

   (a) (1.0 SEA) Officially, R variable names must begin with an alphabetical character and must include only letters, numbers and the symbols "_", "-", and ".". They must not include "*","?","!","&" or other special symbols.

(b) (1.0 SEA) Never name a variable T or F.

This is one thing that almost everybody (99.9%) will agree with. NEVER name variables "T" or "F". These are too easily mistaken for TRUE and FALSE values. Since R uses TRUE and FALSE as vital elements of almost all commands and functions, and since users are allowed to abbreviate those as T or F, a horrible confusion can develop if variables are named T or F.

(c) (.75 SEA) Avoid declaring variables that have the same names as widely used functions. In 2001, I created a variable "rep" (for Republican party members) and nothing worked in my program. In exasperation, I wrote to the r-help list, and learned that I had obliterated R's own function rep with my variable. That kind of mistake was common. In 2002 or so, the R system was revised so that user-declared variables cannot "step on" R system functions. Nevertheless, it is disconcerting to me (probably others) when users create variables with names like "lm", "rep", "seq", and so forth. Its distracting; its confusing.

(d) (0.40) I avoid underscores in variable names. To understand why, please understand the history of S and R. At one time, the underscore "_" was used as the assignment symbol. That's right, instead of "<-", we used to write

```
y _ x + x^2
```

The underscore for assignment was allowed, but discouraged, when I started using R. In those days, R functions that imported data would translate underscores into other symbols. Underscore for assignment has since been forbidden altogether. A while after that, the underscore was allowed in variable and function names. Because of that history, R veterans may still consider it jarring if your variables include underscores.

(e) (0.40 SEA) Use long names for infrequently used variables. If a variable is going to be used twice, we might as well be verbose about it. "xlog" is better than "xl", if we are only writing it a few times. If we are going to use a name 50 times in a 5 line program, we should choose a short one. For abbreviations, include a comment to remind the reader what the thing stands for.

(f) (0.10 SEA) This is my personal naming scheme, nobody else knows about it, unless they have heard about it from me. But they might like it if they think it over. I suggest we use an alphabetical scheme for naming related things so that they always stay together in the workspace. As seen by ls(), the related bits should always be together. From now on, when I work with a variable named "x", then all transformations will begin with "x". I will use "xlog" rather than "logx" and so forth.

Example 1. Create a numeric variable, recode it as a factor, then create the "dummy" variables that correspond.

```
> x <- runif(1000, min = 0, max = 100)
> xf <- cut(x, breaks = c(-1, 20, 50, 80, 101), labels = c("cold", "luke", "warm", "hot"))
> xfdummies <- contrasts(xf, contrasts = FALSE )[xf,]
> colnames(xfdummies) <-  paste("xf", c("cold", "luke", "warm", "hot"), sep="")
> rownames(xfdummies) <- names(x)
> dat <- data.frame(x, xf, xfdummies)
> head(dat)
```

```
         x    xf xfcold xfluke xfwarm xfhot
1 72.09039 warm      0      0      1     0
2 87.57732  hot      0      0      0     1
3 76.09823 warm      0      0      1     0
4 88.61246  hot      0      0      0     1
5 45.64810 luke      0      1      0     0
6 16.63718 cold      1      0      0     0
```

Example 2. Estimate a regression, calculate the summary, extract summary statistics.

```
> set.seed(12345)
> x1 <- rnorm(200, m = 300, s = 140)
> x2 <- rnorm(200, m = 80, s = 30)
> y <- 3 + 0.2 * x1 + 0.4 * x2 + rnorm(200, s=400)
> dat <- data.frame(x1, x2, y); rm(x1,x2,y)
> m1 <- lm (y ~ x1 + x2, data = dat)
> m1summary <- summary(m1)
> (m1se <- m1summary$sigma)
```

```
[1] 397.3396
```

```
> (m1rsq <- m1summary$r.squared)
```

```
[1] 0.02527128
```

```
> (m1coef <- m1summary$coef)
```

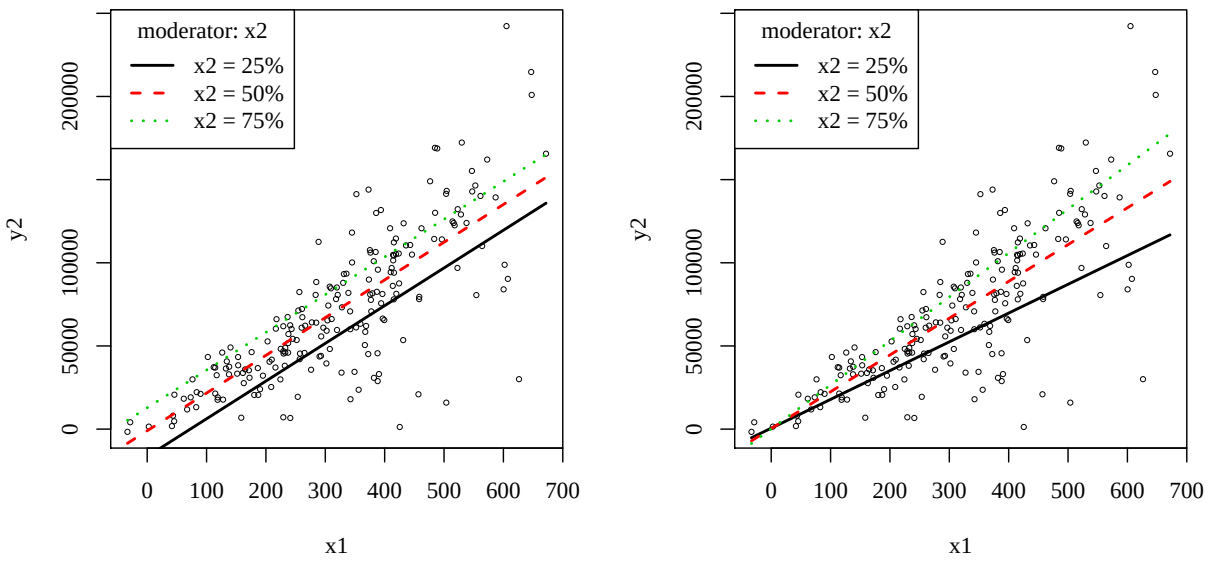
```
                Estimate  Std. Error    t value    Pr(>|t|)
(Intercept) -111.7117859 103.0981785 -1.083548 0.27988980
x1             0.3347818   0.1888153  1.773065 0.07776316
x2             1.3078582   0.9804299  1.333964 0.18375574
```

```
> (m1aic <- AIC(m1))
```

```
[1] 2966.469
```

Example 3. Run a regression, collect mean-centered and residual centered variants of it, summarize each, and compare them.

```
> library(rockchalk)
> dat$y2 = with(dat, 3 + 0.02 * x1 + 0.05 * x2 + 2.65 * x1 *x2 + rnorm(200, s=4000))
> par(mfcol=c(1,2))
> m1 <- lm(y2 ~ x1 + x2, data = dat)
> m1i <- lm(y2 ~ x1 * x2, data = dat)
> m1ps <- plotSlopes(m1, plotx = "x1", modx = "x2")
> m1ips <- plotSlopes(m1i, plotx = "x1", modx = "x2")
```




```
> m1imc <- meanCenter(m1i)
> m1irc <- residualCenter(m1i)
> outreg(list(m1, m1i, m1imc, m1irc), tight = TRUE, modelLabels = c("Linear", "Interaction", "Mean
    Centered", "Residual Centered"))
```

| | Linear Estimate (S.E.) | Interaction Estimate (S.E.) | Mean Centered Estimate (S.E.) | Residual Centered Estimate (S.E.) |
|---|---|---|---|---|
| (Intercept) | -72728.682* | 1369.158 | 70482.081* | -72728.682* |
| | ( 3373.402) | ( 2053.016) | ( 290.315) | ( 1064.652) |
| x1 | 226.746* | -2.383 | . | 226.746* |
| | ( 6.178) | ( 5.768) | | ( 1.95) |
| x2 | 860.306* | -12.511 | . | 860.306* |
| | ( 32.08) | ( 23.023) | | ( 10.124) |
| x1:x2 | . | 2.682* | . | . |
| | | ( 0.064) | | |
| x1c | . | . | 218.979* | . |
| | | | ( 1.958) | |
| x2c | . | . | 846.615* | . |
| | | | ( 10.13) | |
| x1c:x2c | . | . | 2.682* | . |
| | | | ( 0.064) | |
| x1.X.x2 | . | . | . | 2.682* |
| | | | | ( 0.064) |
| N | 200 | 200 | 200 | 200 |
| $RMSE$ | 13001.068 | 4103.162 | 4103.162 | 4103.162 |
| $R^2$ | 0.916 | 0.992 | 0.992 | 0.992 |
| adj $R^2$ | 0.915 | 0.992 | 0.992 | 0.992 |

* $p \leq 0.05$

# 2 R Idioms. What's In R Guts?

## 2.1 Rewriting Formulas. My Introductory Puzzle.

On May 29, 2012, I was working on a regression problem in the rockchalk package. The functions like meanCenter and residualCenter receive a fitted regression model and transform some variables. The non-centered variable "x1" is renamed to "x1c", and then the regression is executed with the new data. Thus it is necessary to take something that is fitted with a formula like y ~ x1*x2, and then re-fit with a formula like y ~ x1c*x2c. In the end, the answer is a single, if complicated line of code that speaks volumes about the way the advanced R user interacts with the system.

My first effort used R's update function. It is fairly easy to replace x1 with x1c in the formula, but not when x1 is logged or otherwise transformed. In exasperation, I wrote to r-help and described the problem with this working example.

```
> 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
```

It gets the target if the target is all by itself, but not otherwise.

While struggling with this, I noticed this really interesting thing. The object "newFmla" is not just a text string. It is actually a list. 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? 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
```

Bingo.

That's quintessential R. It packs together a half-dozen very deep thoughts that I will try to explain in the rest of this section. 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 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. Maybe both.

### 2.2.1 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
334.7175924  -6.5100168  -6.8113110   0.1325085
```

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

```
(Intercept)          x1          x2       x1:x2
334.7175924  -6.5100168  -6.8113110   0.1325085
```

```
> 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.

### 2.2.2 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 can handle 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
334.7175924  -6.5100168  -6.8113110   0.1325085
```

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

```
(Intercept)          x2          x3          x4           y
54.23244193  0.14541440 -0.13451021 -0.09570416  0.09045886
```

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.

## 2.3 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 this 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")))
```

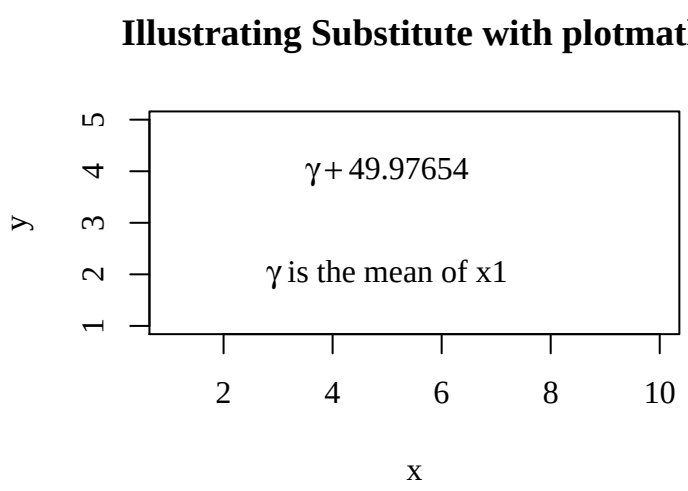
The number 7 is a nice number, but what if we want to calculate something and insert it into the expression? Your first guess might be to insert a function that makes a calculation, such as the mean, but this fails:

```
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)
```

The special things to note are 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 just want symbols. It turns out that the functions as.name and as.symbol are exactly the same, and usually I use as.symbol, but in Gabor's answer to my question, as.name is used, so I will illustrate that here.

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

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

## 2.4 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 -1.6598937  0.02030747 -0.63971861    1
2 -0.2763602 -2.65139775  0.08676547    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 an 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.

### 2.5 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 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.

## References


Bates, Douglas, Martin Maechler and Ben Bolker. N.d. "lme4: Linear mixed-effects models using Eigen and S4.". R package version 0.999902344-0/r1694.
**URL:** *http://R-Forge.R-project.org/projects/lme4/* 4b

Chambers, John M. 2008. *Software for data analysis: programming with R.* Statistics and computing New York ; London: Springer. 3

Fox, John and Sanford Weisberg. 2011. *An R Companion to Applied Regression.* Second ed. Thousand Oaks CA: Sage.
**URL:** *http://socserv.socsci.mcmaster.ca/jfox/Books/Companion* 4b

Kernighan, Brian W. and Dennis M. Ritchie. 1988. *C Programming Language.* 2nd ed. ed. Prentice Hall. 3

Pinheiro, Jose, Douglas Bates, Saikat DebRoy, Deepayan Sarkar and R. Core Team. 2012. "nlme: Linear and Nonlinear Mixed Effects Models.". R package version 3.1-104. 4b

Venables, W. N. and B. D. Ripley. 2002. *Modern Applied Statistics with S*. Fourth ed. New York: Springer. ISBN 0-387-95457-0.
**URL:** *http://www.stats.ox.ac.uk/pub/MASS4* 4b

Xie, Yihui. 2012. "formatR: Format R Code Automatically.". R package version 0.4.
**URL:** *http://CRAN.R-project.org/package=formatR* 1