R Style. An Rchaeological Commentary.

Paul E. Johnson <pauljohn @ ku.edu>

July 25, 2022

1 Introduction: Ugly Code that Runs

Because there is no comprehensive official R style manual, students and package writers seem to think that there is no style whatsoever to be followed. While it may be true that "ugly code runs," it is also 1) difficult to read and 2) frustrating to extend, and 3) tiring to debug. Code is a language, a medium of communication, and one should not feel free no ignore its customs.

After students have finished a semester of statistics with R, they may be ready to start preparing functions or packages. Those R users are the ones I'm trying to address with this note. It is important to realize that the readability of code makes a difference. It sometimes difficult to know that there is a "right way" and a "wrong way" because there are so many examples to study on CRAN.

This note describes R style from an Rchaeological[1] perspective. By examining the work of the R Core Development Team (R Core Team, 2018) and other notable package writers, we are able to discern an implicit style guide. However, this note is not "official" or endorsed from R Core.[2] With one exception at the end of this note, none of the advice here is "my" advice. Instead, it is my best description of the standards followed by the leading R programmers.

At one point, the only guide was the Google R style guide,[3] which was used as a policy for R-related "Google Summer of Code" projects. There are many excellent suggestions in Hadley Wickham's Style Guide.[4] In what follows, I'll try to explain why there are some variations among these projects and offer some advice about how we (the users) should sort through their advice.

2 Rchaeological Methodology

I am a student of R as a programming language. I am also student of the R community as an international success that created a working open source computer program. One of the most interesting differences between R and other open source projects I have observed is that R attracts non-programmers. There is an abundance of statistical novices and untrained computer programmers in the R user community. Many students begin with R as a way of learning about

[1] Definitions:

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.

[2] Yet :)

[3] https://google.github.io/styleguide/Rguide.xml

[4] http://adv-r.had.co.nz/Style.html

computer programming. In contrast, the developers of R are world-class software engineers. They have formal training in computer programming and years of experience in a variety of computer languages. The diversity creates a healthy tension that is easy to see in the r-help email list or on Web forums for R users.

2.1 "Use the Source, Luke," said Obi-Wan

What should R code look like? Stop guessing. The implicit style guide for R is the R source code itself. If users want to communicate with R Core developers, they ought to communicate using the style that developers use.

I'm often surprised to find that R users–even experienced ones–have never looked at the R source code. Before going any further,

> Open the source code for R. I mean, literally, download R-3.5.2.tar.gz (or whatever is current when you read this). Unpack that, navigate to the directory src/library/stats/R. Open the file "lm.R".

That's what R code should look like.

Browse other R files in the source code. Notice the files are suffixed by R, not r!

Then go read a lot of R packages. Begin with the recommended packages (in the R source code under src/library/Recommended). Then draw some samples from CRAN. Choose packages that are prepared by members of R Core, and then sample a few packages that are widely installed, such as John Fox's car package (Fox and Weisberg, 2011).

After that, pick a random sample of packages on CRAN. Don't be surprised by ugly code in a randomly chosen R package.

2.2 Notice How R Describes its Own Style

Type the name of a function at the R command prompt. That is the same as using the function called print.function() to review the contents of a function from an R package. For example, try "lm". The first few lines are

```
> lm
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok =
        TRUE,
    contrasts = NULL, offset, ...)
{
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset", "weights", "na.action"
        ,
        "offset"), names(mf), 0L)
    mf <- mf[c(1L, m)]
    mf$drop.unused.levels <- TRUE
    mf[[1L]] <- as.name("model.frame")
    mf <- eval(mf, parent.frame())
    if (method == "model.frame")
        return(mf)
    else if (method != "qr")
```

```
        warning(gettextf("method = '%s' is not supported. Using 'qr'
            ",
             method), domain = NA)
```

That's quite a bit like the code in the file lm.R, but it is not exactly the same. Even if the code in lm.R were an ugly, horrible mess, its output in the terminal would be indented and spaced just right. That is an important Rchaeological finding!

Why can there be a difference between the code for a function in a file (like "lm.R") and the output of the command (like "lm")? Admittedly, this is difficult to understand. The on-screen output is not (by default, anyway) the source that went into R, but rather it is R's rendition of the internal structure of the function. I recently had an epiphany while reading a section in the *Writing R Extensions* manual called "Tidying R code". That title is a bit misleading. It is not about tidying R source code; rather, it is about beautifying the rendition of internal structures for the terminal. "R treats function code loaded from packages and code entered by users differently. By default code entered by users has the source code stored internally, and when the function is listed, the original source is reproduced. Loading code from a package (by default) discards the source code, and the function listing is re-created from the parse tree of the function." That is to say, if ugly code is syntatically valid, R can parse it and structure it according to the internal dictates of the R runtime system, and when we ask to see the function, we get a nice looking result.

2.3 Formulate SEA estimates.

As already noted, there is no mandatory style for R code. The *R Internals* manual has a section "R coding standards," but it is quite brief. The main point that most readers take away concerns indentation: subsections in code should be preceded by 4 blank spaces, not a tab character.

But there is a larger point in *R Internals*, but novices don't recognize the importance of it. R is a GNU project, and there are GNU coding standards.[5] The R project's C code follows the standard closely. In the entire body of the R source code, we find the GNU thumb print. The importance of that fact is missed by untrained readers, who mistake the lack of a comprehensive discussion of style for an encouragement to "do anything you want."

In the following, I will try to point out the areas of greatest agreement by assigning an SEA score to each point. SEA stands for "Subjective and completely unscientific personal Estimate of Agreement." These are my Bayesian priors. If I could survey my favorite R programmers, I'd find some variety, and I am trying to make it clear where the disagreements might lie. But, then again, I may have been fooling myself. It has recently been suggested to me that these recommendations are not descriptions of the Rchaeological community I'm studying, they are rather my personal litmus test for admirable R programmers.

3 Nearly Universally accepted standards.

3.1 (SEA 1.0) Indentation of code sections is required.

This is explicitly spelled out in the R documentation. No tabs! Insert 4 blank spaces. Personally, I prefer 2 spaces, which has been the default in Emacs. But I'm changing my code to use 4 spaces. If you find my code with 2 spaces, please accept this apology and believe that it is an oversight.

[5] http://www.gnu.org/prep/standards/standards.html

3.2 (SEA .95). Use "<-", not "=", for assignments.

One cannot find the equal sign used for assignments in any file in the R source code. Nor can one find it in any of the Recommended packages (so far as I can tell).

Students who have learned R in introductory textbooks are sometimes shocked to learn that they were taught wrong. I'm sympathetic to their outrage. How can this be?

The equal sign was used by mistake so frequently that the R system was re-designed to tolerate that mistake. *Most* usages of the equal sign for assignments do not cause runtime errors. Not all possible problems were eliminated, however. Thus the equal sign is not recommended, it is tolerated. Nevertheless, A horrible profusion of textbooks and packages ensued using the equal sign for assignment.

3.3 (SEA .98) Blank spaces around symbols are required.

This is a general GNU coding standard.

1. Insert spaces before and after
   a) mathematical symbols like: "=", "<-", "<", "*", "+"
   b) R binary operations like: "%*%", "%o%", and "%in%".
2. Put one space after commas.
3. Insert one space before the opening squiggly braces "{".
4. Put one space after the closing parenthesis ")" and the closing squiggly brace "}".

This is purely a matter of convention and legibility, it does not affect the "rightness" of code.

Other observations about spaces,

1. Do not insert spaces between function names and their opening parentheses.
2. After reviewing the R source code, I was uncertain about whether one ought to insert one space after "if" and "for". From an Rchaeological perspective, this is a little bit perplexing. In the help page for those terms (see help("for")), there is no space after "if" or "for". In the R-3.0.0 source code folder src/library/base/R, I count 1741 instances of "if(" and 683 instances of "if (". The former style seemed right to me, at least at first, because people often say that R's "if" and "for" are functions. I asked for clarification in the R-devel email list, and Peter Dalgaard explained that the space should be used because those terms are

   > language constructs (and they *are* keywords, not names, that's why ?for won't work). The function calls are 'if'(fee, {foo}, {fie}) and something rebarbative for 'for'(....).
   >
   > Besides, both constructs are harder to read without the spaces. (r-devel, April 18, 2013)

   For me, that settles the question. For R code, as in C, "if" and "for" should be treated as keywords, and there would be a space after them, as in "if (x < 7)".
3. Do not insert "extra spaces" inside parentheses.

   Programmers who have written in the BASH scripting language may recall that a space inside brackets is required. That training causes me to think that R code is a little bit "jammed together." This is pleasant to my eye:

   ```
   if ( (x == 1) & (y == 2) ) {
   ```

   but, from an Rchaeological point of view, more the correct style is:

   ```
   if((x == 1) & (y == 2)) {
   ```

   The insertion of the interior parentheses for the smaller conditions inside the if statement is consistent with the GNU standard for C.

Is there an "argument exception" to the space rule for equal signs?

Package writers are not entirely consistent, and Rchaeologically speaking, we cannot be sure if these variations are accidental. We sometimes find no spaces, as in

```
plot(x, y, lwd=4, col=green, main="My Title")
```

It would surely be more correct like so:

```
plot(x, y, lwd = 4, col = green, main = "My Title")
```

Spaces may sometimes be omitted in an effort to keep code on one line. Especially where publishers are concerned about the use of scarce paper, the omission of spaces around equal signs is not uncommon. Please note, however, that it is NEVER acceptable to omit the spaces after commas!

What about indentation of long function declarations?

One of the interesting space related questions is the indentation of function declarations when there are many arguments. Consider the R source code for the function lm():

```
lm <- function (formula, data, subset, weights, na.action,
                method = "qr", model = TRUE, x = FALSE, y = FALSE,
                qr = TRUE, singular.ok = TRUE, contrasts = NULL,
                offset, ...)
```

Note that lines 2-4 are indented under the letter "f" in formula. If the function's name were longer, it would push all of that indented code to the right, probably causing line wraps. The solution is to put the function's name and the assignment symbol on separate line. This is the format of R's function plot.lm().

```
plot.lm <-
function (x, which = c(1L:3L,5L), ## was which = 1L:4L,
          caption = list("Residuals vs Fitted", "Normal Q–Q",
          "Scale–Location", "Cook's distance",
          "Residuals vs Leverage",
          expression("Cook's dist vs Leverage  " * h[ii] / (1 - h[ii
             ]))),
          panel = if(add.smooth) panel.smooth else points,
          sub.caption = NULL, main = "",
          ask = prod(par("mfcol")) < length(which) &&
             dev.interactive(), ...,
          id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75,
          qqline = TRUE, cook.levels = c(0.5, 1.0),
          add.smooth = getOption("add.smooth"),
          label.pos = c(4,2), cex.caption = 1)
{
```

The continuation is indented to be below the first argument. The benefit of this "declaration by itself" approach is that the additional lines are always re-formatted with consistent indentation and we are not creating a huge empty white space due to indentation.

Try formatR::tidy.source()

The advice so far mostly concerns "white space". We would like a programmer's text editor to handle automatically as much of that as possible.

The R package "formatR" (Xie, 2012) has a function called tidy.source() which can often (but not always) clean up code. 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 works 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")
    }
}
```

The tidy.source() function can get rid of equals sign assignments if we ask it to. (In my opinion, it should do that by default.)

```
> 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")
    }
}
```

The tidy.source() function can receive input as files or whole directories.

There are two reasons why tidy.source() is not a panacea. First, by design, tidy.source() will fail if there are programming errors in the original source code. That leads to a Catch-22. I want to clean up the code to find out why it does not run, but tidy.source() cannot clean it up because it does not run. Second, quite often it happens that tidy.source() chokes on unexpected user code. Especially problematic is code that has comments inserted in unexpected places. For example, I recently ran tidy.source() on the file emb.r in the package Amelia (Honaker et al., 2011).

```
> library(formatR)
> tidy.source("emb.r")
Error in base::parse(text = text, srcfile = NULL) :
  152:88: unexpected SPECIAL
151: }
152: if (ncol(as.matrix(startvals)) == AMp+1 && nrow(as.matrix(
   startvals)) == AMp+1)          %InLiNe_IdEnTiFiEr%
```

I would estimate that tidy.source() fails on about one-third of the R code I randomly select from CRAN.

3.4 (SEA .70) The "} else {" policy.

Did you notice "} else {" in the tidy.source() output for myfn()? That's the correct style. We should not have the left squiggly brace "}" on a separate line from the "else," and the right squiggly brace "{" should be on that same line. This is, well, obviously good (in my opinion).

Why? Try this at the command line.

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

R does not realize that it is not yet finished with the if keyword's work. The keyword else appears to begin a new thought, which is illegal. The if's help page (run help("if") or ?"if") is referring to this problem 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.

I agree with the somewhat extreme attitude, but will compromise: If one uses squiggly braces, always follow the "} else {" policy.

Some might follow a soft line on this, suggesting only that **users should not begin a line with the word else**. That does not go quite far enough for me. I'd add, **always use squiggles after else.** This is simply a way of avoiding a very common coding error. This code is OK:

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

But it invites a coding error like so:

```
if (x < 7) print("so far, so good") else
print("this is else")
print("and we want this also to be with else, but it is not")
```

To be perfectly clear, and to protect ourselves against editing errors in the future, we could follow the "somewhat extreme" advice and write this:

```
if (x < 7) {
  print("so far, so good")
} else {
    print("this is else")
    print("and we want this also to be with else")
}
```

Counter-argument based on the R source code

This would be a completely closed case if not for the fact that the "} else {" policy is ignored in vast expanses of the R source code. In the R source code, scan for the keyword else and in almost every file, one finds:

```
}
  else
```

A naked else! This is frustrating for writers of style guides. It ignores the advice in the "if" help page. We cannot run this code line-by-line.

On the other hand, the function that includes that apparently runs! Why doesn't that code crash? When an if/else statement is enclosed in a larger area that is demarcated by squiggly braces, then R will understand the naked else when it finds it. Observe the fix at the command line:

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

I don't think I'm going to have any luck persuading the R Core Development Team that their naked elses need to be fixed. The best I can do is to urge code writers to use "} else {" and make them responsible for errors that result from ignoring that rule.

One will note another interesting anomaly while reviewing R source code. Unlike programs written in C, where a consistent style for the placement of squiggly braces will be followed, in R we observe files that do not follow a particular rule. In src/library/src/logLik.R, we find functions in both the K&R (Kernighan and Ritchie, 1988) C style

```
nobs.logLik <- function(object, ...) {
    res <- attr(object, "nobs")
    if (is.null(res)) stop("no \"nobs\" attribute is available")
    res
}
```

and we also find the vertically aligned squiggly braces approach:

```
print.logLik <- function(x, digits = getOption("digits"), ...)
{
    cat("'log Lik.' ", paste(format(c(x), digits=digits), collapse="
        , "),
         " (df=",format(attr(x,"df")),")\n",sep="")
    invisible(x)
}
```

I am at a loss to explain these stylistic variations, so I conclude that R users can follow either style, while keeping in mind the "} else {" policy, which strongly pushes us toward the K&R style.

4 How to name functions.

Now we begin to consider some issues that are more subjective. Many styles are legal, but some are more easily understood. R syntax has changed over the years, and some things that were illegal are now allowed. And some styles that were standard might now be discouraged.

4.1 (.98 SEA) Avoid using names that are already in use by R, especially common ones.

Don't write functions named "rep()", "seq()", "c()", and so forth. Notice that my new function lm() does not obliterate the one from the stats package, but it sure does make it harder to use it.

```
> lm <- function(z) print("Hi, I'm z where lm was")
> x <- rnorm(100)
> y <- rnorm(100)
> lm (y ~ x)
[1] "Hi, I'm z where lm was"
> stats::lm(y ~ x)

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

Coefficients:
(Intercept)            x
    0.02688      0.01796
```

As long as we remember that lm() is in the namespace stats, we can find it.

Similarly, packages can declare namespaces of their own. (Since R version 2.14, all packages *must* do so.) We are allowed to place a new function like seq() or lm() into a package if we want to. Nevertheless, almost everybody will hate to read code like that.

The danger that user functions might interfere with core functionality was at one time very serious. Now it is, for the most part, a historical footnote. It is still possible to obliterate a function that is embedded within a namespace, but doing so requires a bit of effort and mischief.[6]

When we say that a namespace is imported, it means that all of the functions in that namespace can be accessed by the function's name, without the namespace name as a prefix. We might write base::seq(1, 10, length.out = 40) to be clear, but we need only write seq(1, 10, length.out = 40) because an R session imports the namespace base. I notice a trend in R to suggest that one should not import whole namespaces unless that is truly necessary, and even if a namespace is imported, we should strive for clarity by using syntax that includes the namespace name. In the source code for many R examples, one will find syntax like graphics::par() where, until recently, that would have simply been par().

4.2 (.65 SEA)Use periods to indicate classes, otherwise don't use periods in function names.

Instead, use camel case to name functions. This function name mySuperThing() is better than my.super.thing().

The period in a function name has a special meaning in the S3 object-oriented framework. A "generic function" (such as print() or summary()) is accompanied by methods that implement its work for particular kinds of objects, such as print.function() or print.lm(). Before the period, we have a function's name, and after the period, we have the class name of the object being managed. The function name my.super.thing() suggests the user might have an object of class "thing" and that my.super(x) would diagnose the class of x and send the work to my.super.thing(). A camel cased function name mySuperThing() will not convey the wrong meaning.

If we were starting with a clean slate, I believe many R functions would be re-named for the purposes of consistency. Since we do not have a clean slate, we live with an accumulation of function names from olde S and R. Changes in computer science–the growth of object-oriented programming–cause new naming conventions. Consider some of the traditional S function names

[6] In case you wonder, here's how to cause the worst case scenario.

```
nseq <- function(x) print("Hello, good to see you")
assignInNamespace("seq.default", nseq, "base")
```

that are still used in R, like read.table and read.csv. Those are not method implementations of a generic function read(). The period is simply part of a shorthand of the form "action.qualifier". Otherwise, if one had an object of type table, then read(x) would call read.table(x). But it does not:

```
> example(table)
> class(tab)
[1] "xtabs" "table"
> read(tab)
Error: could not find function "read"
```

I believe that, if these functions were being created today, they would be named readTable() and readCSV().

In the R source code, there are some very confusing function names and I have a hard time believing we would use them if we were re-designing everything today. The file src/library/base/readhttp.R has a function called url.show(), which follows none of the styles that I recognize. There's no class show and url() is not a generic function. In the "action.qualifier" tradition, it would be show.url(). And why not showURL()? I hasten to point out that the same file includes some camel cased functions like defaultUserAgent().

I like camel cased function names. They are common in Objective-C and Java. Some programmers vigorously disagree. Programmers trained in C++ seem to hate camel case names, almost at a visceral level. As a result, we find a division of opinion on function names. As a spot check, consider two of my favorite packages, MASS (Venables and Ripley, 2002) and car (Fox and Weisberg, 2011). There are not many camel case function names in the MASS, where we find brief names in lower case letters (such as boxcox()). In contrast, car calls that boxCox(). When I started using R, Professor Fox used function names with periods, but he has been systematically weeding them out and replacing them with camel case names. 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., 2013) and lme4 (Bates et al., 2012), weigh in on the camel case side of the ledger.

In conclusion, users should avoid gratuitous periods in function names because, after S3, the period has special meaning in R. When a function has been declared as a generic, then that function's name followed by a period has an object-oriented meaning. A period is not merely word separation. New functions introduced in R tend to use either camel case names (browseVignettes ()) or underscores (get_all_vars ()). Considering recent additions to R, I believe that the chance of finding a decorative period in a new function name is almost zero. But we are still living with an awful lot of older counter-examples.

5 How to name variables (and objects).

5.1 (1.0 SEA) Follow the "letters and numbers" rule.

R variable names must

1. begin with an alphabetical character
2. include only letters, numbers and the symbols "_" and ".".

They must not include "*","?","!","&" or other special symbols. They must not include spaces.

One peculiar side effect of this rule is that the ellipsis symbol, three periods, "...", is actually a legal object name. That's three periods, which is just as legal as aaa or bbb. Many R functions allow the argument "...", most users don't realize it literally is a word. When that is listed as a function argument, then any argument that the user includes is gobbled up by "...".

5.2 (1.0 SEA) Never name a variable T or F.

Almost everybody (99.999%) will agree with this. 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.

Here's some good news. R will not allow users to name variables TRUE or FALSE:

```
> TRUE <- 7
Error in TRUE <- 7 : invalid (do_set) left-hand side to assignment
```

But R will not prevent the usage of T and F for variable names.

5.3 (.75 SEA) Avoid declaring variables that have the same names as widely used functions.

This is just a handy rule of thumb now, but it used to be a "watch out for that tree!" warning. 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. rep() is used inside many R functions and thus obliterating it was a very serious mistake. 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.

5.4 (0.50 SEA) Use long names for infrequently used variables.

And use short names for variables that will be used very often.

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.

5.5 (0.10 SEA) Suggested naming scheme: keep related objects in an alphabetically sorted scheme.

This is my personal naming scheme. Nobody but me follows this policy now, but I like it so much I'm tacking it onto the end of this essay. I believe that R code is much more readable if objects that "go together" begin with the common series of letters. As seen by ls(), the related pieces 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. I include the output in order to emphasize the clarity due to the alphabetical emphasis:

```
> 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
```

I have not included the output of these code chunks, but the alphabetical emphasis is demonstrated in them.

Example 2. Estimate a regression, calculate summary information.

```
> 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$sigma
> m1rsq <- m1summary$r.squared
> m1coef <- m1summary$coef
> m1aic <- AIC(m1)
```

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

```
> 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)
```

6 Conclusion

R can be understood at several levels, varying in sophistication from an elementary statistics course or to an advanced platform for the development of computer programming concepts. In the future, I will be more cautious to teach new R users about coding style. I intend to prevent the accumulation of bad habits that result in code that is difficult to read and hard to debug.

Users who ask for help in the r-help email list [7] or on web forums [8] are well advised to remember the importance of style. Most newcomers believe that the experts will understand what they write, but that's not true. Experts will find it much easier to spot errors in code that has the correct indentation and uses a proper naming scheme for variables and functions. In my experience, the most likely source of trouble in R code is not actually the style, but rather poor

[7] http://www.r-project.org/mail.html
[8] e.g., http://stackoverflow.com/questions/tagged/r

compartmentalization of separate calculations. The potential to compartmentalize, however, is obscured by bad style.

When users throw together 2000 lines of spaghetti code with no indentation (I can point to examples on CRAN), there's almost no chance than anyone except the author will be able to understand and extend that kind of code. Ugly code writers will respond, "my ugly code runs!" That misses the point. 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. And where possible, the code should be re-usable and extended to other purposes.

References

Bates, D., M. Maechler, and B. Bolker (2012). *lme4: Linear mixed-effects models using S4 classes*. R package version 0.999999-0. 4.2

Fox, J. and S. Weisberg (2011). *An R Companion to Applied Regression* (Second ed.). Thousand Oaks CA: Sage. 2.1, 4.2

Honaker, J., G. King, and M. Blackwell (2011). Amelia II: A program for missing data. *Journal of Statistical Software 45*(7), 1–47. 3.3

Kernighan, B. W. and D. M. Ritchie (1988, April). *C Programming Language* (2nd ed. ed.). Prentice Hall. 3.4

Pinheiro, J., D. Bates, S. DebRoy, D. Sarkar, and R Core Team (2013). *nlme: Linear and Nonlinear Mixed Effects Models*. R package version 3.1-107. 4.2

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0. 1

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S* (Fourth ed.). New York: Springer. ISBN 0-387-95457-0. 4.2

Xie, Y. (2012). *formatR: Format R Code Automatically*. R package version 0.4. 3.3