Style guide

Style guides for R code are numerous, diverse, and contradictory. The following is borrows heavily from Google's R style guide (https://google.github.io/styleguide/Rguide.xml) and Hadley Wickham's guide (http://r-pkgs.had.co.nz/r.html) , which is largely similar.

The main aesthetic difference between Wickham and Google is that the former prefers underscore ( \_ ) to separate "words" while Google prefers dot ( . ). Both accept using a capital letter to begin a 2nd (or later word) as an alternative to their preferred scheme, which is the solution adopted in this guide.

In naming variables and functions, the general principles are:

(1) to create names that are concise yet readily interpretable;

(2) when possible, create names that reflect natural text representations of statistics writing;

(3) use lower case letters in names, with capital letters (not underscores) to distinguish words in a name;

(4) treat singular terms as single words when possible;

(4) use abbreviations when there is little chance of misinterpretation.

**Exceptions**

The coding conventions described below should be followed unless there is compelling reason to do otherwise. Some exceptions include legacy code and modifying third-party code.

**Arrays, etc.**

In most cases, use arrays, matrices, vectors, and lists rather than dataframes for data. Compared with dataframes, the alternatives are more efficient (in terms of storage space and calculation speed), more flexible (in terms of re-indexing for calculations or other tasks), and frequently less clumsy (in terms of using the data directly in calculations). Dataframes are preferred when data from the structure is to be extracted via formula, when storing spreadsheet-like data containing columns of different data types (text, numeric, dates), or when required for compatibility with legacy code. Also, note that when a dataframe is created with a text column, the default is to store the column as a factor rather than vector of character strings. Factors and character strings behave differently, and if you want strings in your dataframe, use stringsAsFactors = FALSE when creating your dataframe, e.g.:

data.frame(a = 1:2, b = c("a", "b"), stringsAsFactors = FALSE)

GOOD

xyArray <- cbind(x = 1:5, y = sample(5))

dbinom(xyArray, 10, .2)

xyArray %\*% 1:2

AVOID

xyDataframe <- data.frame(x = 1:5, y = sample(5))

dbinom(dfxyDataframe, 10, .2)

xyDataframe %\*% 1:2

GOOD

lm(y ~ x, xyDataframe)

AVOID

lm(y ~ x, xyArray)

Use dimension names rather than numbers to subset or extract data from matrices, arrays, vectors, dataframes, and lists, especially when data structure naturally lends itself to named dimensions. The possibility of creating errors when using “magic numbers” to index matrices, arrays, vectors, data frame, and lists are numerous. Using names rather than magic numbers makes it easier to edit data formats; reduces the chances for errors when data structures are altered; and makes your code more readable, both to others and to yourself when you return to your code after a hiatus.

GOOD

myData$SiteName # if myData is a dataframe

GOOD

myData[ , "SiteName"] # if myData is a dataframe, array, or matrix with named columns

AVOID

myData[ , 3]

**Identifiers**

Simple variables should usually be all lower case, avoiding upper case and underscores ( \_ ) except when necessary to avoid confusion. Dot (.) has a standard meaning in S3 classes, so avoid using it altogether in variable and function names unless you are using it for S3 generics. If you are tempted to use a . , see if you can get by with capital letter or \_ instead.

For variables that involve actual words (as opposed to, say, a parameter with a simple modifier), judicious use of capital letters to distinguish separate words should be used to improve readability.

GOOD

singleYearEstimate

AVOID

singleyearestimate

single\_year\_estimate

single.year.estimate

When possible, names should be chosen to reflect the way they appear and are conceptualized in statistical writing. For example, a single statistical parameter or quantity modified by text representation of another symbol or by a word, use a single lower-case word to name it. For example, betahat would be preferred over beta\_hat or betaHat or BetaHat. In the expression, , the hat is intimately associated with the beta and is not a separate "word." The simple, natural text rendering of would be a unified betahat rather than a separated beta\_hat. Also, varbeta would be preferred over beta\_var because varbeta is closer to the way statistics is written and read and it reduces ambiguity (e.g., "beta\_var ...Is that the beta variable? Or is it the variance of beta?")

GOOD

betahat

varbeta

AVOID

beta\_hat

beta\_var

Abbreviations:

Commonly used abbreviations are preferred over whole words unless there is danger of misinterpretation. Strive to avoid using abbreviations that readers of the code could not be expected to decipher with ease (e.g., bm21, xmdp).

Prefixes and suffixes:

n is a widely used and commonly understood symbol for "number of..." Use it freely as a prefix to what it is enumerating: ndays, nsearch, nyr, ncarc, nsim, nobs. If the enumerated quantity has a long name or if an n prefix could introduce ambiguity or confusion, capital letters may improve readability (e.g., nPrimaryFactors). Other widely used and commonly understood symbols can be used in a similar way as a prefix or suffix: min, max, var, hat, bar, p, etc.

GOOD

nsim

ncarc

NOTE: Abbreviations like carc may not be in general use. However, in mortality estimation software, carcasses are a central theme, so carc would be a useful abbreviation in such a project.

AVOID:

number\_of\_simulation\_reps

number\_of\_carcasses

n\_carcasses

For nested loops, names of loop counters should be suggestive of what is being looped over followed by an i suffix.

GOOD:

for (simi in 1:nsim){

for (xi in 1:nx){

for (si in 1:nsize){

for (gi in 1:length(g)){

for (modi in 1:nmods){

<lots of code>

}

}

}

}

}

AVOID:

for (i in 1:nsim){

for (j in 1:nx){

for (k in 1:nsize){

for (l in 1:length(g)){

for (m in 1:nmods){

<lots of code>

}

}

}

}

}

Underscores:

When a conceptually similar quantity is calculated for a small number of subunits and using capital letters to distinguish among them is awkward (e.g., because it would force capitalization of a statistical term that is normally lower-case), underscores ( \_ ) can be useful. When a conceptually similar quantity is calculated for a larger number of subunits, an array or list is usually preferable.

When combining a word with a term that cannot be capitalized without affecting its interpretation or when referencing specific instances of a broader statistical measure, an underscore is a good alternative to capitalization of things that should not be capitalized.

GOOD:

betahat\_p

betahat\_k

# Here, p and k are specific instantiations of estimator betahat, but in context they should appear as lower case.

AVOID:

betahatP

betaPhat

betahat.p

beta\_hat\_p

**Quotes**

When using quoted names, strings, and text, use double ( " ) quote symbols rather than single ( ' ). In R, the symbols have the same meaning, but consistency is helpful for readability and for searches (ctrl + f). The choice of one over the other is largely one of aesthetics, but singles have additional meanings in comments (e.g., apostrophes, a signal to devtools::document()that to parse the line for creating .Rd help files) and doubles are used in C and C++, so use doubles.

GOOD

saveRDS(myData, "myData.RDS")

AVOID

saveRDS(myData, 'myData.RDS')

**Spacing**

In most cases, place spaces around binary operators (=, +, -, <-, etc.). This is particularly important for <- and = to facilitate searches for function definitions and arg names. Always place a space after a comma. Place one before a comma only when no word or variable immediately precedes it. Place a space before left parentheses, except in a function call.

GOOD  
colnames(x) <- c("tubes", "lumps", "branches")

x[ , "branches"] <- 21

if (debug) plot(testdata)

AVOID   
x[,3] <- 21

if(debug)plot ( testdata )

# include spaces after if and before plot

# no spaces after plot or around testdata

Exceptions: Spaces may be profitably omitted to economically group terms in mathematical expressions by order of operations to facilitate reading.

GOOD a\*x + b

OK a \* x + b

AVOID a\*x+b

Do not put spaces around : or :: operators

GOOD

a:b

survival::survreg

AVOID

a : b

survival :: survreg

**Indentation**

Indent code two spaces for each unresolved opening brace above it. Never use tabs or mix tabs and spaces.

GOOD:

if (weight < 6){

plot(x, y)

for (j in 1:nrow(CI)){

lines(rep(j, 2), CI[j, ])

}

}

AVOID:

if (weight < 6){ # absense of indenting makes it difficult to determine code hierarchy

plot(x, y)

for (j in 1:nrow(CI)){

lines(rep(j, 2), CI[j, ])

}

}

AVOID:

if (w < 6){ # indenting top level code makes it harder to discern its position in code hierarchy

plot(x, y)

for (j in 1:nrow(CI)){

lines(rep(j, 2), CI[j, ])

}

}

**Line Length**

Strive to keep line length to ≤ 80 characters to facilitate printing code, side-by-side viewing of files, and code editors with narrow windows. Lines that occasionally spill longer than that are OK.

When long statements that exceed well beyond 80 characters are difficult to avoid (e.g., long arg lists for a function, complicated calculations), break lines in a way that preserves natural groupings and coherence. Use the "two spaces" convention for indenting.

GOOD

plot(x, y,

type = "n", axes = F,

xlim = c(1, 80), ylim = c(miny[ai], 1), # natural grouping of parameters

xlab = "method", ylab = "coverage",

main = paste0("1 - \u03b1 = ", 100\*(1-alpha[ai]), "%")

) # closing parenthesis matches the indentation of the function call

legend("topright",

legend = c(

"horoscopes",

"psychics",

"entrails",

"omens",

"coin tosses",

"expert opinions",

"Ouija board",

"dice rolls",

"gut feelings"

),

pch = 1:9, lty = rep(1:3, each = 3), col = rep(1:3, 3)

)

ALSO GOOD

longNamedPlottingFunction(x, y,

type = "n", # parameters on separate lines

axes = F,

xlim = c(1, 80),

ylim = c(miny[ai], 1),

xlab = "method",

ylab = "coverage",

main = paste0("1 - \u03b1 = ", 100 \* (1 - alpha[ai]), "%")

)

AVOID:

longNamedPlottingFunction(x, y,

type = "n",

axes = F,

xlim = c(1, 80),

ylim = c(miny[ai], 1),

xlab = "method",

ylab = "coverage",

main = paste0("1 - \u03b1 = ", # broken line

100 \* (1 - alpha[ai]), "%"))

Extreme indenting burns space on the left side of the page and often forces awkward line breaks which negatively affect readability.

**Braces**

An opening curly brace should never go on its own line; a closing curly brace should always go on its own line. Avoid empty lines immediately below an opening brace or immediately above a closing brace. Short control statements may be placed entirely on a single line (without curly braces) or split into separate lines (with curly braces).

GOOD

if (is.null(ylim)) ylim <- c(0, 0.06)

if (is.null(ylim)){

ylim <- c(0, 0.06)

}

AVOID

if (is.null(ylim))

ylim <- c(0, 0.06)

if (is.null(ylim)) {ylim <- c(0, 0.06)}

**Assignment**

Use <- for assignment outside parentheses, and = for assigning names in lists, arrays, vectors, dataframes, and R6 classes and in function calls. The assignment operator ( <- ) inside of parentheses reassigns the value to the variable outside the parentheses as well. An equal sign does not. If you need to reassign or define an outside-the-parentheses variable, do so outside the parentheses.

GOOD  
y <- c(1, 2, 4, 5)

x <- 5

x <- list(y1 = 3, y2 = c("b", "doofus"))

summary\_y <- c(avg = mean(y), sd = sqrt(var(y)))

AVOID  
x = 5

summary\_y = c(avg <- mean(y), sd <- sqrt(var(y)))

GOOD  
y <- c(1, 2, 4, 5)

y <- y + 2\*x

funFun(y = y)

AVOID  
y <- c(1, 2, 4, 5)

funFun(y <- y + 2\*x)

# This framework may kill two birds with one stone, but one of the birds might be the value of y that you were counting on. This is a recipe for error (and confusion for others reading your code).

**Semicolons**

Avoid terminating your lines with semicolons or using semicolons to put more than one command on the same line. One exception would be to assign and normalize a variable on the same line.

GOOD

pMgX <- dbinom(X, M, g) \* pM; pMgX <- pMgX/sum(pMgX)

LESS DESIRABLE

pMgX <- dbinom(X, M, g) \* pM / sum(dbinom(X, M, g) \* pM) # redundant calculation

pMgX <- dbinom(X, M, g) \* pM # dangling definition

pMgX <- pMgX/sum(pMgX)

tmp <- dbinom(X, M, g) \* pM # superfluous and potentially confusing variable

pMgX <- tmp/sum(tmp)

**Exceptions for Temporary Debugging Code**

Authors may insert temporary debugging code in non-standard formats.

OK

if (!debug){ # non-indented, temporary debugging code within braces...helps identify the code as top level

x <- 5

x <- list(y1 = 3, y2 = c("b", "doofus"))

summary\_y <- c(avg = mean(y), sd = sqrt(var(y)))

}

for (i in 1:1000){

result[i] <- doLengthyCalculations(x[i])

if (i %% 100 == 0) {print(i); flush.console()} # prints occasional progress report on loop

<much more code>

}

Non-indented if facilitates finding all instances of this type of line for easy removal for production code.

Putting the entire debug code on a single line efficiently highlights it a debug code.

**Commenting Guidelines**

Comment your code. In creating a package, much of the commenting for the function details will be outside the body of the function in a format conducible to creating .Rd help files (e.g., http://r-pkgs.had.co.nz/man.html). Within function bodies, a brief summary of what the function does should immediately follow the function definition. Very brief description of input args, output structure, and/or side effects are optional inside the function body if they are included outside.

Simple commented lines should begin with # and one space. Short comments can be placed after code. Extensive comments may be organized in outline form, with two or more #s at the top level and one fewer # with each successive sublevel.

GOOD

calcDBmeanDiff <- function(DB1, DB2, xname, yname){

# extract x and y values from two dataframes and

# return the mean difference

x <- NULL

y <- NULL

for (i in 1:nDB){

x <- c(x, get(paste0("DB", i))[ , xname])

y <- c(x, get(paste0("DB", i))[ , yname])

}

mean(x - y)

}

Avoid over-wordiness and superfluous explanations.

AVOID

calcDBmeanDiff <- function(DB1, DB2, xname, yname){

# In this function, xname and yname data are extracted from

# the main databases in order to calculate the mean of the

# difference of x and y.

# x-values from the databases will be appended to x in a loop

x <- NULL

# y-values from the databases will be appended to y in a loop

y <- NULL

# loop to extract the data from each database, respectively

for (i in 1:nDB){ # i is an index to the n databases

# append the x-values from DBi to x vector

x <- c(x, get(paste0("DB", i))$x)

# append the y-values from DBi to y vector

y <- c(x, get(paste0("DB", i))$y)

}

# calculate the mean of the difference of the x and y vectors

meanval <- mean(x - y)

# return the mean value

return(meanval)

}

**Function Definitions and Calls**

Function definitions should first list arguments without default values, followed by those with default values.

In both function definitions and function calls, multiple arguments per line are allowed; line breaks are only allowed between assignments.

**Package Develoment**

Avoid using library() and require() in package source code. Rather, in most cases refer to functions from other packages using the :: operator—e.g., survival::survreg(), eoa::postM(). However, if functions from another package are used extensively, load the package via the Depends line in the DESCRIPTION file to reference the functions directly (without the :: operator) rather attaching via library() or require().