R Style Guide

J. Kyle Wathen Fei Chen Add other Authors

December 02, 2020

Contents

Chapter 1

Introduction

This document provides a list of programming guidelines for use in R software development in the Biopharmaceutical industry with a main focus on biostatistics.

This style guide was based on input of many people, many years of software development experiences and utilizing the Google R Style guide and Tidyverse Style guide. Both guides provided useful information and are very similar in structure and nature. The Tidyverse guide was originally based on the Google style guide, but Google's current guide is derived from Tidyverse style guide.

All style guides are a matter of opinion, therefore, the rational and thinking behind each recommendation is provided with examples. While one may develop software without a style guide, utilizing a style guide can make reading the software source much easier for people other than the original developer. As a developer, it is recommended to make your software source read like a book rather than a file of "code" intended to be difficult to understand and interpret.

When developing software, the developer should always have a goal in mind. This style guide strives to help R developers in creating an easy to ready, predictable and testable software source utilizing the following:

1. Balance between readability and efficiency. When given the option between the two following development approaches: a) a really efficient, very difficult to follow and understand approach vs b) a less efficient but easily understood option, ALWAYS take the less efficient option that is easy to follow and less likely to have bugs. The efficient version can be provided as alternative and comparison of results for testing is suggested.
2. Use descriptive and meaningful names for files, functions and variables.
3. Extremely long files can be very difficult to follow and test and should be avoided.

4. Where there are style options, pick a option and be consistent within a set of code such as a project, package or shiny app.

5. Keep in mind that, programming, like writing, is a naturally individualistic endeavor. But unlike writing, software development is often a joint and collaborative effort. Therefore great attention must be paid on balancing of individual style against collective consistency.

1.1 References

Google's R Style Guide https://google.github.io/styleguide/Rguide.html

Tidyverse Style Guide Used within the Tidyverse https://style.tidyverse.org/

Tidyverse Style Guide Git Repository https://github.com/tidyverse/style Several files in this document are modified versions from the Tidyverse GitHub repository.

Chapter 2

Files

2.1 File Naming

File names should be meaningful and end in .R. This style agrees with Google's recommendation of using BigCamelCase for file names. Do not use special characters in the file names and use only letters or numbers. That is, no spaces, - or _ should be included in a file name. While - or _ do make the file names more readable than running words together like runningwordstogether, the use of - or _ requires extra key strokes when compared to BigCamelCase where the first letter of each word is capitalized.

File names should be descriptive and provide insight into what the file contains. Generic names like foo.r or stuff.r should not be used. Vague file names like calculate.r are discouraged and more descriptive like CalculatePosteriorProbs.R are preferred.

Examples:

```
# Good Examples
CalcuatePosteriorProbs.R
AnalyzeSurvivalData.R
InputFunctions.R

# Bad
calculate.r
foo.r
get file name.r
functions for stuff.r
```

If files should be viewed or executed in a particular order, prefix the file name with a number using two digit format starting at 00.

```
00_SimulateData.R
01_SimulateArrivalTimes.R
02_CreateDataSet.R
...
09_LogisticRegressionModel.R
10_BetaBinomialModel.R
```

Capitalization can be important when sharing files with collaborators on differing operating systems and source control systems. Do not name files where the only difference is the capitalization.

2.2 File Organization

2.3 Internal Structure

Files should be kept as short as possible and not include numerous functions or R source. If a file contains many functions and is very long, it should be broken into meaningful files to group common functionality. Break the file into meaningful code blocks, preferably as functions, and separate code blocks by comments using -, # or =. [QUESTION: Should we allow all 3 for comments, eg -, # and = or just a subset like the first two?]

```
# Create simulated data ------

# Plot results ====

# Save output ####
```

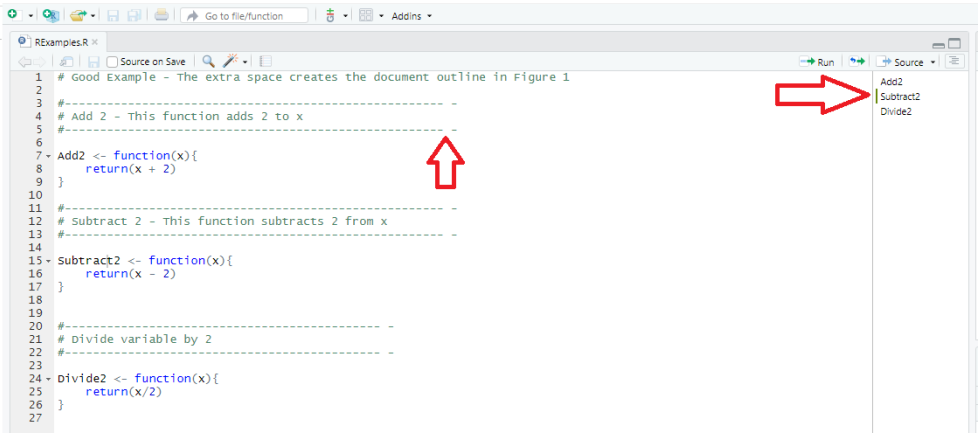
Each comment should end with 4 consecutive -, # or = as this helps with providing well organized outline in R studio. In addition, If you create a comment block, surrounded by —, ===== or #### then please make sure you put a space preceding the last charter. The space is to help with creating organized document outline in RStudio for R files.

```
# Good Example - The extra space creates the document outline in figure below
#------------------------------------------------------ -
# Add2 - This function adds 2 to x
#------------------------------------------------------ -
Add2 <- function(x){
    return(x + 2)
}
```

```
#---------------------------------------------------------- -
# Subtract2 - This function subtracts 2 from x
#---------------------------------------------------------- -
Subtract2 <- function(x){
    return(x - 2)
}

#---------------------------------------------- -
# Divide variable by 2
#---------------------------------------------- -
Divide2 <- function(x){
    return(x/2)
}
```

Good example with red arrows to indicate the extra space and resulting file outline to the right.



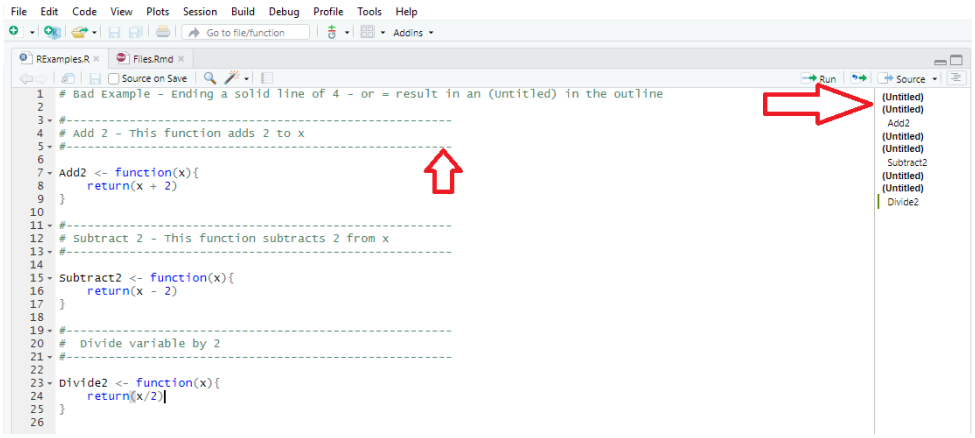

```
# Bad Example - Ending a solid line of 4 - or = result in an (Untitled) in the outline
#----------------------------------------------------------
# Add 2 - This function adds 2 to x
#----------------------------------------------------------
Add2 <- function(x){
    return(x + 2)
}

#----------------------------------------------------------
# Subtract 2 - This function subtracts 2 from x
#----------------------------------------------------------
Subtract2 <- function(x){
    return(x - 2)
```

```
}

#-------------------------------------------------------
#  Divide variable by 2
#-------------------------------------------------------
Divide2 <- function(x){
    return(x/2)
}
```

Bad example of comments, notice how the (Untitled) appears in the document outline



Chapter 3

Functions

3.1 Naming

Function names should be meaningful and descriptive and provide insight to what the function does. Typically, functions names are VERBS and describe the action the function performs. This style agrees with Google's recommendation of using BigCamelCase for function names. Do not use special characters in the function name, but using letters or numbers is acceptable. That is, -, . and _ should not be included in a function name. While - or _ do make the function names more readable than running words together like functionname, the use of - or _ requires extra key strokes when compared to BigCamelCase where the first letter of each word is capitalized. To avoid confusion with S3 methods a . should not be used in a function name unless it is to define an S3 method, see example below.

In addition, it is recommended that function names should contain verbs or a word that describes the action the function performs. Vague generic function names like PerformTasks() or Com() should not be used.

```
# Good Example
ComputePosteriorParameters()
ComputePosteriorParameters.BetaBinom()   # If ComputePosteriorParameters is a generic function th
SimulatePatientArrivalTimes()

# Bad Example
post.par()        # Incorrect name capitalization and including ., non-descriptive
arrival-time()    # Incorrect name capitalization and including -
addhr()           # Capitalizaiton and vague
ComputePosteriorParameters.BetaBinom()   # If ComputePosteriorParameters is NOT a generic functio
```

3.2 Structure

Functions should be self contained and only depend on the arguments. They should NOT use variables in a higher scope. A call to a function with the same arguments should always return the same value, unless the function is supposed to produce random variables ect. [QUESTION: What does everyone thing about there not being a preceding and trailing space before the first argument and after the last argument? I prefer function(x, y), but Tidyverse suggests not putting spaces inside or outside side of parentheses see https://style.tidyverse.org/syntax.html#spacing so it would allow function(x, y) but not function(x, y).

Also, I have always put the { } for a function in line with each other and the first letter of the function name like the examples below but both TV and Google recommend the opening { after function, for example Add2 <- function(x) { X <- x + 2 return(x) }

I am okay with either approach.]

```
# Example

#-------------------------------------------------------------------------------------------
# VERY BAD STYLE -  What this function returns depends on what y was before the call
#-------------------------------------------------------------------------------------------
MyFunction <- function(x)
{
    x <- x + y
    return(x)
}

y <- 5
MyFunction(4)   # return 9
y <- 10
MyFunction(4)    #returns 14


#-------------------------------------------------------------------------------------------
# Acceptable example - a call to the function only depends on arguments
#-------------------------------------------------------------------------------------------
MyFunction2 <- function( x, y )
{
    x <- x + y
    return( x )
}

y <- 5
```

```
MyFunction2( 4, 7 )   # return 11
y <- 10
MyFunction2( 4, 7 )    #returns 7
```

3.3 Explicit Returns

Do NOT rely on R's implicit return feature as an explicit return is much clearer for the reader. Tidyverse and Google do not agree on this recommendation and this guide follows Google as it is much more transparent to developers that may not be as familiar with R.

```
# Good Example - explicit return
MyFunction(x, y)
{
    return(x + y)
}

# Bad Example - rely on explicit return can be confusing and more error prone
MyFunction(x, y)
{
    x + y
}
```

3.4 Local function

A local function refers to defining a function within another function. Local functions are difficult to test, understand and prone to errors. Local functions should be avoided unless necessary.

```
## Good Example
RunAnalysis1 <- function( dfData )
{
    return( lm( dfData$y ~ dfData$x ) )
}
RunAnalysis2 <- function( dfData )
{
    return( lm( dfData$y ~ dfData$x + dfData$x*dfData$Trt ) )
}
RunAnalysis3 <- function( dfData )
{
    return( lm( dfData$y ~ dfData$x + dfData$x2 ) )
}
```

```
AnalyzeData <- function( strType, dfData )
{
    if( strType == "ONE" )   fit <- RunAnalysis1( dfData )
    if( strType == "TWO" )   fit <- RunAnalysis2( dfData )
    if( strType == "THREE" ) fit <- RunAnalysis3( dfData )

    return( fit )
}

## Bad Example
AnalyzeData <- function( strType, dfData ){
    if( strType == "ONE" )   ana <- function( dfData ){ lm( dfData$y ~ dfData$x ) }
    if( strType == "TWO" )   ana <- function( dfData ){ lm( dfData$y ~ dfData$x + dfDat
    if( strType == "THREE" ) ana <- function( dfData ){ lm( dfData$y ~ dfData$x + dfDat

    fit <- ana( dfData )
    return( fit )
}
```

The above "Good" example provides functions RunAnalysis1, RunAnalysis2, RunAnalysis3 that are much easier to test. However, AnalyzeData could be improved further in this example with the use of s3 classes where the class(dfData) determines which analysis to call thus eliminating the need for the if statements.

Chapter 4

Variables

4.1 Naming

Variable names should be meaningful and descriptive. Do not use special characters in the variable names, use only letters or numbers. That is, no spaces, - or _ should be included in a variable name. While - or _ do make the variables names more readable than running words together like runningwordstogether, the use of - (not allowed in R) or _ requires extra key strokes. Do not use a . in variable names as this can cause confusion for the S3 object system.

Variable names should be in camelCase with the first letter being lower case. For example, sampleMean and not SampleMean as the the first letter capital is reserved for function names.

Abbreviating long words with a common abbreviation is acceptable but be consistent within a code base or repository. For example, Qty or Quant for Quantity, Pat or Pats for Patients. Avoid using single letter abbreviations. Within a project it is often helpful to create a list of common abbreviations in the project so that all developers use consistent naming.

Single letter variable names should only be used for looping variables, but it is often easier to read and follow your source code if the looping variable is meaningful. For example, using iPat is better than i as an variable that loops over patients in a vector and is easier to follow.

Since R does not require types, for example integer, double, vector etc it can be VERY help and there for strongly encouraged to use common prefixes for variable names. Many users find this helpful, especially, for functions as the user of a function can easily understand what variable types are expected.

Use the following prefixes to help others understand what the intended type of a variable is.

1. Prefix **integer** variable with an n then camel case, eg nQtyOfReps would be an integer variable for the quantity of replications, nQtyOfPats = quantity of patients
2. Prefix **double** of **float** variables with d, eg dMean would be a double/float variable for mean.
3. Prefix **logical** value (TRUE or FALSE) with a b, eg bSingleArm, bAdjust
4. Prefix **vectors** with v, eg vMeans would be a vector of means
5. Prefix **matrix** with m, eg mVarCov would be a matrix for the variance-covariance.
6. Prefix a **dataframe** with df, eg dfPats would be a dataframe containing patient data.
7. Prefix **list** with a l, eg lData would be list of data
8. Prefix a **class** variable with a c, eg cAnalysis = structure(list(), class= "TTest"")

Examples

```
# Good Examples
qtyOfPatients
nQtyOfPatient   # Better name because it is clear it should be an integer

sampleMeans
vSampleMeans    # Better name because it is clear that a vecor of values is expected

dStdDev         # Std Dev is a common abbrivation for standard diviation

for(i in vPatients){
    vTreatment[i] <- vPatients[i]
}

# Better example because more complex loops the iPat is clearer than i
for(iPat in vPatients){
    vTreatment[ iPat ] <- vPatients[ iPat ]
}

# Bad Examples
xxx
sd
x.m.3
x.mad
```

Chapter 5

Syntax

5.1 Spacing

Spacing is important so others can browse your code with more ease. Put space around arithmetic and logical operators, and after ",". Whether space should be inserted immediately after (or [and before) or] is a matter of personal choice, but whatever style you prefer, adhere to it throughout your project.

```
# Bad Examples
  m=1
  s=2
  x=rnorm(1000,mean=m,sd=s)

# Good Examples
  dMean <- 1 # mean
  dSd <- 2 # standard deviation
  nSize <- 1000 # sample size
  vSamp <- rnorm(n = nSize, mean = dMean, sd = dSd)

# Last is preferred making longer code block less cramped, especially when there are multiple arg
 x[i,] and f(x) # Acceptable
 x[i, ] # Better
 x[ i, ] and f( x )# Best
```

5.2 Code Blocks

Code should be organized so that there is less repetition and more abstraction.

```
# Bad Example
  a <- sin(1.2) + exp(5.7)
  b <- sin(2.4) + exp(-2)
  c <- a + b

# Better Example
  f <- function(x, y){ return( sin(x) + exp(b) ) }
  a <- f(2.4, -2)
  b <- f(1.2, 5.7)
  c <- a + b
```

One liner function definitions are reasonable if the line is short, simple and follow the explicit return, however for readability and testing the following is preferred.

```
# Even Better Example
  f <- function(x, y)
  {
    return( sin(x) + exp(b) )
  }

  a <- f( 2.4, -2 )
  b <- f( 1.2, 5.7 )
  c <- a + b
```

5.3 Assignment

There are five ways to assign a variable values in R.

```
# Five ways to say thank you
thx <- 1
thx <<- 1
thx = 1
1 -> thx
1 ->> thx
```

Even though "=" and "<-" are interchangeable, typical R style guides urge the use of "<-", one for historical reasons, another for consistency with the double "«-" operator for which "=" has no counterpart. The rightwards form work naturally in conjunction with pipe operator "%>%" although the use of "%>%" lends to obfuscation and is discouraged.

For clarity and consistency, it is recommended to only use <- and «- and do not use =, -> and -».

5.4 Semicolons

Don't put ; at the end of a line and don't use ; to put multiple commands on one line.

```
# Good Examples
x <- 5
y <- 2
z <- 3

#Bad Examples
x <- 5; y <- 2; z <- 3;


x <- 5; y <- 2;
z <- 3;

x <- 5;
y <- 2;
z <- 3;
```

5.5 Comments

Comments should be meaningful and add understanding to the code. If your code requires a comment for most lines, consider rewritting be clearer. Utilizing naming conventions is helpful. In addition, using organized files and creating functions for complex code blocks can be helpful.

5.6 Furthermore

There are tools to help you better format your code, packages such as "formatR", "styler", that can manage spacing and indentation reasonably.

Chapter 6

GitHub

6.1 Commit messages

Commit messages can be very helpful to figure out what a commit does. Non-descriptive commit messages should be avoided. The following general guidelines should be considered when writing commit messages:

- The first line is the subject of the commit and should give a brief description
- If you need to include more detail skip a line then include more content
- If you are fixing issues you can add Fix # as this will close the issue on GitHub when the commit is merged into the main branch.

6.2 Pull Requests

The title of the pull request should provide a brief description of what the pull request does.

If the PR fixes an issue include the phrase Fixes # so the issue is closed.

6.3 Branch names

Having descriptive branch names can be very useful in team development. Consistently naming branches can help with transparency and ease of tracking what branches are used for. Typically, branches are created to address issues such as

new features, bug fixes or documentation for a particular branch like, Development, Dev-V0.1.2, master. Using the following labels Feature, Bugfix, Doc (for documentation) a branch name should be made like the following:

< label >-< Key word(s) >-< Developer Initials >

For example, if you are working a branch named Dev and you want to create a feature branch with a brief description of perform analysis and your initials are KW then a good branch name would be

Feature-Add-Analysis-KW

and if you wanted to reference an issue #14 then you could also include the issue number in the branch, for example

Feature14-Add-Analysis-KW.

A common practice is to create a Dev branch, set Dev as the default branch and require code reviews and Pull Requests for merges to the main or Dev branches. Typically feature branches are created off of the Dev branch then PRs will merge feature branches into Dev. After a release version is completed then Dev is merged into the main branch via a PR.