Google's R Style Guide

R is a high-level programming language used primarily for statistical computing and graphics. The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. The rules below were designed in collaboration with the entire R user community at Google.

- ## Summary: R Style Rules
    1. File Names: end in `.R`
    2. Identifiers: `variable.name`, `FunctionName`, `kConstantName`
    3. Line Length: maximum 80 characters
    4. Indentation: two spaces, no tabs
    5. Spacing
    6. Curly Braces: first on same line, last on own line
    7. Assignment: use `<-`, not `=`
    8. Semicolons: don't use them
    9. General Layout and Ordering
    10. Commenting Guidelines: all comments begin with # followed by a space; inline comments need two spaces before the #
    11. Function Definitions and Calls
    12. Function Documentation
    13. Example Function
    14. TODO Style: `TODO(username)`
- ## Summary: R Language Rules
    1. `attach`: avoid using it
    2. Functions: errors should be raised using `stop()`
    3. Objects and Methods: avoid S4 objects and methods when possible; never mix S3 and S4

1. Notation and Naming

- **File Names**

  File names should end in `.R` and, of course, be meaningful.
  GOOD: `predict_ad_revenue.R`
  BAD: `foo.R`

- **Identifiers**

  Don't use underscores (_) or hyphens (-) in identifiers. Identifiers should be named according to the following conventions. Variable names should have all lower case letters and words separated with dots (.); function names have initial

capital letters and no dots (CapWords); constants are named like functions but with an initial `k`.

- `variable.name`
  GOOD: `avg.clicks`
  BAD: `avg_Clicks` , `avgClicks`
- `FunctionName`
  GOOD: `CalculateAvgClicks`
  BAD: `calculate_avg_clicks` , `calculateAvgClicks`
  Make function names verbs.
  Exception: When creating a classed object, the function name (constructor) and class should match (e.g., lm).
- `kConstantName`

2. Syntax

Line Length

The maximum line length is 80 characters.

Indentation

When indenting your code, use two spaces. Never use tabs or mix tabs and spaces.
Exception: When a line break occurs inside parentheses, align the wrapped line with the first character inside the parenthesis.

Spacing

Place spaces around all binary operators (=, +, -, <-, etc.).
Exception: Spaces around ='s are optional when passing parameters in a function call.

Do not place a space before a comma, but always place one after a comma.

GOOD:

```
tabPrior <- table(df[df$daysFromOpt < 0, "campaignid"])
total <- sum(x[, 1])
total <- sum(x[1, ])
```

BAD:

```
tabPrior <- table(df[df$daysFromOpt<0, "campaignid"])  # Needs
spaces around '<'
tabPrior <- table(df[df$daysFromOpt < 0,"campaignid"])  # Needs a
space after the comma
tabPrior<- table(df[df$daysFromOpt < 0, "campaignid"])  # Needs a
space before <-
```

```
tabPrior<-table(df[df$daysFromOpt < 0, "campaignid"])  # Needs
spaces around <-
total <- sum(x[,1])  # Needs a space after the comma
total <- sum(x[ ,1])  # Needs a space after the comma, not before
```

Place a space before left parenthesis, except in a function call.

GOOD:
```
if (debug)
```

BAD:
```
if(debug)
```

Extra spacing (i.e., more than one space in a row) is okay if it improves alignment of equals signs or arrows (<-).

```
plot(x    = xCoord,
     y    = dataMat[, makeColName(metric, ptiles[1], "roiOpt")],
     ylim = ylim,
     xlab = "dates",
     ylab = metric,
     main = (paste(metric, " for 3 samples ", sep="")))
```

Do not place spaces around code in parentheses or square brackets.
Exception: Always place a space after a comma.

GOOD:

```
if (debug)
x[1, ]
```

BAD:

```
if ( debug )  # No spaces around debug
x[1,]  # Needs a space after the comma
```

- **Curly Braces**

An opening curly brace should never go on its own line; a closing curly brace should always go on its own line. You may omit curly braces when a block consists of a single statement; however, you must *consistently* either use or not use curly braces for single statement blocks.

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

xor (but not both)

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

Always begin the body of a block on a new line.

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

- **Assignment**

  Use `<-`, not `=`, for assignment.

  GOOD:
  ```
  x <- 5
  ```

  BAD:
  ```
  x = 5
  ```

- **Semicolons**

  Do not terminate your lines with semicolons or use semicolons to put more than one command on the same line. (Semicolons are not necessary, and are omitted for consistency with other Google style guides.)

3. Organization

- **General Layout and Ordering**

  If everyone uses the same general ordering, we'll be able to read and understand each other's scripts faster and more easily.

  1. Copyright statement comment
  2. Author comment
  3. File description comment, including purpose of program, inputs, and outputs
  4. `source()` and `library()` statements
  5. Function definitions
  6. Executed statements, if applicable (e.g., `print`, `plot`)

  Unit tests should go in a separate file named `originalfilename_unittest.R`.

- **Commenting Guidelines**

  Comment your code. Entire commented lines should begin with `#` and one space.

  Short comments can be placed after code preceded by two spaces, `#`, and then one space.

```
# Create histogram of frequency of campaigns by pct budget spent.
hist(df$pctSpent,
     breaks = "scott",  # method for choosing number of buckets
     main   = "Histogram: fraction budget spent by campaignid",
     xlab   = "Fraction of budget spent",
     ylab   = "Frequency (count of campaignids)")
```

- **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.
  GOOD:

```
PredictCTR <- function(query, property, numDays,
                       showPlot = TRUE)
```

  BAD:

```
PredictCTR <- function(query, property, numDays, showPlot =
                       TRUE)
```

  Ideally, unit tests should serve as sample function calls (for shared library routines).

- **Function Documentation**

  Functions should contain a comments section immediately below the function definition line. These comments should consist of a one-sentence description of the function; a list of the function's arguments, denoted by `Args:`, with a description of each (including the data type); and a description of the return value, denoted by `Returns:`. The comments should be descriptive enough that a caller can use the function without reading any of the function's code.

- **Example Function**

```
CalculateSampleCovariance <- function(x, y, verbose = TRUE) {
  # Computes the sample covariance between two vectors.
  #
  # Args:
  #   x: One of two vectors whose sample covariance is to be
calculated.
  #   y: The other vector. x and y must have the same length,
greater than one,
  #      with no missing values.
  #   verbose: If TRUE, prints sample covariance; if not, not.
Default is TRUE.
  #
  # Returns:
```

```
  #   The sample covariance between x and y.
  n <- length(x)
  # Error handling
  if (n <= 1 || n != length(y)) {
    stop("Arguments x and y have invalid lengths: ",
         length(x), " and ", length(y), ".")
  }
  if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
    stop(" Arguments x and y must not have missing values.")
  }
  covariance <- var(x, y)
  if (verbose)
    cat("Covariance = ", round(covariance, 4), ".\n", sep = "")
  return(covariance)
}
```

- **TODO Style**

  Use a consistent style for TODOs throughout your code.
  `TODO(username): Explicit description of action to be taken`

4. Language

- **Attach**

  The possibilities for creating errors when using `attach` are numerous. Avoid it.

- **Functions**

  Errors should be raised using `stop()`.

- **Objects and Methods**

  The S language has two object systems, S3 and S4, both of which are available in R. S3 methods are more interactive and flexible, whereas S4 methods are more formal and rigorous. (For an illustration of the two systems, see Thomas Lumley's "Programmer's Niche: A Simple Class, in S3 and S4" in R News 4/1, 2004, pgs. 33 - 36: http://cran.r-project.org/doc/Rnews/Rnews_2004-1.pdf.)

  Use S3 objects and methods unless there is a strong reason to use S4 objects or methods. A primary justification for an S4 object would be to use objects directly in C++ code. A primary justification for an S4 generic/method would be to dispatch on two arguments.

  Avoid mixing S3 and S4: S4 methods ignore S3 inheritance and vice-versa.

5. Exceptions

The coding conventions described above should be followed, unless there is good reason to do otherwise. Exceptions include legacy code and modifying third-party code.

6. Parting Words

Use common sense and BE CONSISTENT.

If you are editing code, take a few minutes to look at the code around you and determine its style. If others use spaces around their `if` clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them, too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on *what* you are saying, rather than on *how* you are saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity will throw readers out of their rhythm when they go to read it. Try to avoid this. OK, enough writing about writing code; the code itself is much more interesting. Have fun!

7. References

http://www.maths.lth.se/help/R/RCC/ - R Coding Conventions
http://ess.r-project.org/ - For emacs users. This runs R in your emacs and has an emacs mode.