Hands-On Data Science with R
Stylistic Preferences for R

Graham.Williams@togaware.com

26th December 2015

Visit http://HandsOnDataScience.com/ for more Chapters.

Data Scientists write programs to load, manage, transform, analyze and model data in a variety of ways. Our programs need to be read and understood by others. Here we present guidelines for programming in R that assists with the transparency of our programs and the enjoyment of others reading our programs.

A programming style aims to ensure consistency and ease of understanding. When we write programs we **write for others to easily read and to learn from and to build upon**. We are writing a story to keep the reader (including ourselves) informed and engaged. If another person (or even ourselves at a later time) can't follow the program then we have not succeeded. Keep it simple and explain what is happening as we proceed through the program.

The programs we write have to run on a computer but computers care little about programming style—the computer's compiler or interpreter will translate our program automatically into machine code that the computer can directly execute. Very ugly code (with correct syntax) will also be executable why not make it attractive and accessible to other humans.

This chapter introduces a style for interacting with R. Most interactions should be through writing commands into a file and having those commands submitted to R for execution. We are always writing R scripts and our R script files should be easily readable and understandable by ourselves and be enjoyable for others to read—they should tell a story.

As we work through this chapter, new R commands will be introduced. Be sure to review the command's documentation and understand what the command does. You can ask for help using the `?` command as in:

```
?read.csv
```

We can obtain documentation on a particular package using the *help=* option of `library()`:

```
library(help=rattle)
```

This chapter is intended to be hands on. To learn effectively, you are encouraged to have R running (e.g., RStudio) and to run all the commands as they appear here. Check that you get the same output, and you understand the output. Try some variations. Explore.

1 Naming Files

1. Files containing R code use the `R` extension. This aligns with the fact that the language is unambiguously called "R" and not "r." Such files might contain the script of some data analysis we did or it could contain a support function that we have written to help us repeat some tasks more easily. For the former case, we might have some analysis of power generator data performed on the 20th of July 2016, and we might choose to name the file:

   Preferred

   ```
   power_analysis_160720.R
   ```

   Discouraged

   ```
   power_analysis_160720.r
   ```

2. Name a file to match the name of the function defined within the file. For example, if the support function we've defined in the file is `myFancyPlot()` then name the file as preferred below. This clearly differentiates support function filenames from analysis scripts, and we have a ready record of the support functions we might have developed simply by listing the folder contents.

   Preferred

   ```
   myFancyPlot.R
   ```

   Discouraged

   ```
   MyFancyPlot.R
   my_fancy_plot.R
   my.fancy.plot.R
   my_fancy_plot.r
   ```

3. R binary data filenames end in ".RData". I have no strong motivation for this except that it conforms to the capitalised naming scheme combined with the language being called R (i.e., using an uppercase `R`).

   Preferred

   ```
   weather.RData
   ```

   Discouraged

   ```
   weather.rdata
   weather.Rdata
   weather.rData
   ```

4. Standard file names use lowercase where there is a choice.

   Preferred

   ```
   weather.csv
   ```

   Discouraged

   ```
   weather.CSV
   ```


2 Naming Objects

5. Function names begin lowercase and use capitalized verbs. A common alternative we see in R is to use underscore to separate words within a function name but our style uses this for variables within datasets.

   Preferred

   ```
   displayPlotAgain()
   ```

   Discouraged

   ```
   DisplayPlotAgain()
   displayplotagain()
   display.plot.again()
   display_plot_again()
   ```

6. Variable names and function argument names use dot separated words. This is marginally simpler to type than variable names using underscore (requiring the use of the shift key). The use of underscores in variable names is kept for the names of variables within a dataset.

   Preferred

   ```
   num.frames <- 10
   num.libs   <- 4
   ```

   Discouraged

   ```
   num_frames <- 10
   numframes <- 10
   numFrames <- 10
   ```

7. Constants are all capitals. This makes them stand out and makes it clear that they should not be changed.

   Preferred

   ```
   MAX.LINES
   ```

   Discouraged

   ```
   const.max.lines
   ```

8. Variables within a dataset (i.e., data frame columns) are lowercase and use underscore to separate the words. This has the advantage that underscore is acceptable in SQL databases for columns, whereas a period is often used to identify the server/database/table/schema names. We often load/save data in data frames from/to databases. We can use `rattle::normVarNames()` from rattle (Williams, 2015) to normalize variables names in this way.

   Preferred

   ```
   min_temp
   wind_gust_speed
   ```

   Discouraged

```
max.pressure
wind.dir
WindSpeed
```

3 Layout

9. Keep lines to less then 80 characters. Whilst this might be based on the printed page limitations it remains salient as generally we do not like to read very long lines.

10. Don't add spaces around the "=" for named arguments in parameter lists. I prefer this as visually it ties the named arguments strongly together. This is the only situation where I tightly couple a binary operator. In all other situations there should always be a space around the operator. Another motivation is that it avoids splitting the line between the argument name and the argument value.

    Preferred

```
read.csv(file="data.csv", sep=";", na.strings=".")
```

    Discouraged

```
read.csv(file = "data.csv", sep =
         ";", na.strings
         = ".")
```

11. Use a consistent indentation. I personally prefer 2 within both my Emacs ESS and RStudio environments with a good font (e.g., Courier font in RStudio works well but Courier 10picth is too compressed). Some argue that 2 spaces is not enough to show the structure when using smaller fonts. If it is an issue for you then 4 is okay or even choosing a different font will assist. The rationale for not having too large an indent is that we still often have limited lengths on lines on some forms of displays where we might want to share the code. This is usually about 80 characters and we do like to not extend too far to the right. Indenting 8 characters is probably too much because it makes it difficult to read through the code with such large leaps for our eyes to follow to the right. Nonetheless, there are plenty of tools to re-indent to a different level if we so choose.

```
window_delete <- function(action, window)
{
  if (action %in% c("quit", "ask"))
  {
    ans <- TRUE
    msg <-"Terminate?"
    if (!dialog(msg))
      ans <- TRUE
    else
      if (action == "quit")
        quit(save="no")
      else
        ans <- FALSE
  }
  return(ans)
}
```

```
window_delete <- function(action, window)
{
        if (action %in% c("quit", "ask"))
```

```
        {
                ans <- TRUE
                msg <- "Terminate?"
                if (!dialog(msg))
                        ans <- TRUE
                else
                        if (action == "quit")
                                quit(save="no")
                        else
                                ans <- FALSE
        }
        return(ans)
}
```

12. Have a single `base::return()` from a function. Understanding a function with multiple and nested returns can make it difficult to understand the function. However, for simple functions, like the one below, multiple returns are just fine.

```
window_delete <- function(action, window)
{
  if (action %in% c("quit", "ask"))
  {
    ans <- TRUE
    msg <-"Terminate?"
    if (!dialog(msg))
      ans <- TRUE
    else
      if (action == "quit")
        quit(save="no")
      else
        ans <- FALSE
  }
  return(ans)
}
```

```
window_delete <- function(action, window)
{
  if (action %in% c("quit", "ask"))
  {
    ans <- TRUE
    msg <-"Terminate?"
    if (!dialog(msg))
      return(TRUE)
    else
      if (action == "quit")
        quit(save="no")
      else
        return(FALSE)
  }
```

```
}
```

13. Align curly braces. Thus an opening curly brace is on a line by itself. This is a particular difference with some other programming styles. My motivation is that the open and close curly braces are then aligned visually and this provides an added visual check of syntax correctness and visually gives a very strong code block view. The placement of the open curly bracket at the end of the previous line is, however, endemic, hiding the opening of a block of code simply to save on having some additional lines (which was only important many years ago when we used punched cards or terminals limited to 24 lines). The preferred style also makes it easier to comment out, for example, just the line containing the "while" and still have valid syntax. Don't be afraid of the extra white space—for the human reader, white space is good, and the computer does not care.

    Preferred

    ```
    while (blueSky())
    {
      openTheWindows()
      doSomeResearch()
    }
    retireForTheDay()
    ```

    Discouraged

    ```
    while (blueSky()) {
      openTheWindows()
      doSomeResearch()
    }
    retireForTheDay()
    ```

    ```
    if (TRUE)
    {
      42
    }
    else
    {
      666
    }
    ```

    The problem is that after the first `}` the interactive interpreter looks ahead for an else on the same line, because that is all that it has to go on interactively, and does not find it and thus assumes this is the end of the if and performs the command. Then the else on the next line is an error

    ```
    Error: unexpected 'else' in "else"

    ALSO

    > source("tmp.R")
    Error in source("tmp.R") : tmp.R:5:1: unexpected 'else'
    4: }
    ```

```
5: else
   ^
```

Put it inside a function, where it is not interactively interpreted, and all is okay:

```
myFun <- function()
{
  if (TRUE)
  {
    42
  }
  else
  {
    666
  }
}
myFun()
```

No simple solution for the interpreter. So might need to do something less satisfactory for top level if statements in a script file or when writing interactively (though we would not often doe this interactively):

```
if (TRUE)
{
  42
} else
{
  666
}
```

14. Align the assignment operator for blocks of assignments. The rationale for this idiosyncratic style suggestion is that it is easier for us to read the assignments in a tabular form than it is when it is jagged. This is akin to reading data in tables—such data is much easier to read when it is aligned. Space is used to enhance readability.

    Preferred

    ```
    a       <- 42
    another <- 666
    b       <- mean(x)
    brother <- sum(x)/length(x)
    ```

    Default

    ```
    a <- 42
    another <- 666
    b <- mean(x)
    brother <- sum(x)/length(x)
    ```

15. Align the `magrittr::%>%` operator in pipelines and the `base::+` operator for ggplot2 (Wickham and Chang, 2015) layers. Arguably the operators otherwise risk being lost among the text and pushing them out simply adds some space and provides a visually pleasing symmetry, much like a table would present. Aligning though requires extra work

and is not often supported by our editors. We don't generally adhere to this style within this book as a disadvantage is that the operator is often too far to the right and risks being overlooked by the novice.

Contentious

```
library(rattle)
ds        <- weatherAUS
names(ds) <- normVarNames(names(ds))
ds                                          %>%
  group_by(location)                        %>%
  mutate(rainfall=cumsum(risk_mm))          %>%
  ggplot(aes(date, rainfall))                +
  geom_line()                                +
  facet_wrap(~location)                      +
  theme(axis.text.x=element_text(angle=90))
```

Default

```
ds <- weatherAUS
names(ds) <- normVarNames(names(ds))
ds %>%
  group_by(location) %>%
  mutate(rainfall=cumsum(risk_mm)) %>%
  ggplot(aes(date, rainfall)) +
  geom_line() +
  facet_wrap(~location) +
  theme(axis.text.x=element_text(angle=90))
```

4 Function Definition Layout

16. Function definitions should be no longer than a screen full. Long functions generally suggest the opportunity to consider more modular design.
17. Always include a space after a comma.
18. Align function arguments by comma. This is an idiosyncratic style, but it works to emphasize the arguments and indeed it makes it easier to comment out particular arguments with little fuss.

Contentious

```
dialPlot <- function(label="UseR!"
                   , value=78
                   , dial.radius=1
                   , value.cex=3
                   , value.color="black"
                   , label.cex=3
                   , label.color="black"
                    )
{
  ...
}
```

Preferred

```
dialPlot <- function(label="UseR!", value=78, dial.radius=1,
                     value.cex=3, value.color="black",
                     label.cex=3, label.color="black")
{
  ...
}

dialPlot <- function(label="UseR!",
                     value=78,
                     dial.radius=1,
                     value.cex=3,
                     value.color="black",
                     label.cex=3,
                     label.color="black")
{
  ...
}
```

5 Function Call Layout

19. Arguments to function calls aligned by comma. Once again, the prefix comma on the line is quite convenient in allowing us to quickly comment out the whole line and retain correct syntax, except for the first line.

Contentious

```
dialPlot(label="UseR!"
       , value=78
       , dial.radius=1
       , value.cex=3
       , value.color="black"
       , label.cex=3
       , label.color="black"
        )
```

Preferred

```
dialPlot(label="UseR!", value=78, dial.radius=1,
         value.cex=3, value.color="black", label.cex=3,
         label.color="black")

dialPlot(label="UseR!",
         value=78,
         dial.radius=1,
         value.cex=3,
         value.color="black",
         label.cex=3,
         label.color="black")
```

6 Functions from Packages

20. R has a mechanism (called name spaces) for identifying the names of functions and variables from specific packages. There is no rule that says a package provided by one author can not use a function name already used by another package or by R itself. Thus functions from one package might overwrite the definition of a function with the same name from another package or from base R itself. A mechanism to ensure we are using the correct function is to prefix the function call with the name of the package providing the function, just like `plyr::mutate()`.

    Generally in commentary we will use this notation to clearly identify the package which provides the function. In our interactive R usage and in scripts we tend not to use the namespace notation. It can clutter the code and arguably reduce its readability even though there is the benefit of clearly identifying where the function comes from. We do note though that package writers are required to use the namespace notation for all calls to functions defined in external packages.

    Preferred

```
library(dplyr)     # Data wranlging, mutate().
library(lubridate) # Dates and time, ymd_hm().
library(ggplot2)   # Visualize data.

ds <- get(dsname)                                          %>%
  mutate(timestamp=ymd_hm(paste(date, time)))              %>%
  ggplot(aes(timestamp, measure))                           +
  geom_line()                                               +
  geom_smooth()
```

    Interesting
    The use of the namespace prefix reduces the attractiveness or conciseness of the presentation and that has a negative impact on the readability of the code. However it makes it very clear where each function comes from.

```
ds <- get(dsname)                                          %>%
  dplyr::mutate(timestamp=
                  lubridate::ymd_hm(paste(date, time)))    %>%
  ggplot2::ggplot(ggplot2::aes(timestamp, measure))         +
  ggplot2::geom_line()                                      +
  ggplot2::geom_smooth()
```

7 Kuhn Checklist

21. Max Kuhn, author of caret (Kuhn *et al.*, 2015) developed a checklist and posted it to the R developers mailing list in January 2012. I have paraphrased some of the points here and embellished it a little with my views, but they are mostly in sync with Kuhn's views.

    (a) Extend the work of others and avoid redundancy. Reuse others functions, with due credit, to add any missing features.

    (b) For a categorical model builder ensure the target is a factor (like Yes/No) rather than integers (like 1/0). The factor levels should be identified in the resulting model object and the `stats::predict()` function should return predicted classes as factors with the same levels and ordering of levels. Support a `type=` to switch between predicted classes and class probabilities. Use `type="prob"` for probabilities.

    (c) Implement a separate `stats::predict()`, using `predict.class()` where *class* is the class of the object returned by the model builder. Do not use special functions like `modelPredict()`.

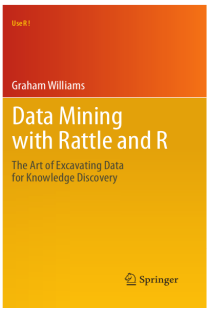
    (d) Provide both a formula interface as in `foo`(y~x, data=ds) and non-formula interface as in `foo`(x, y) to the function. "Formula methods are really inefficient at this time for large dimensional data but are fantastically convenient. There are some good reasons to not use formulas, such as functions that do not use a design matrix (e.g., `party::cforest()`) or need factors to be handled in a non-standard way (e.g., `Cubist::cubist()`)."

    (e) "Don't require a test set when model building."

    (f) If not all variables are used in the resulting model, allow the required subset of variables to be provided for `stats::predict()` and not all the original variables, and avoid referencing variables by position rather than name.

8 Further Reading

The Rattle Book, published by Springer, provides a comprehensive introduction to data mining and analytics using Rattle and R. It is available from Amazon. Other documentation on a broader selection of R topics of relevance to the data scientist is freely available from `http://datamining.togaware.com`, including the Datamining Desktop Survival Guide.



This chapter is one of many chapters available from `http://HandsOnDataScience.com`. In particular follow the links on the website with a * which indicates the generally more developed chapters.

I like the guidelines at Google but I have my own idiosyncrasies. The style decisions I have made I motivate above, based on over 30 years of programming in very many different languages. Also see Wikipedia for an excellent summary of many styles.

Rasmus Bååth, in The State of Naming Conventions in R, reviews naming conventions used in R, finding that the initial lower case capitalised word scheme for functions was the most popular, and dot separated names for arguments similarly. This is the style I prefer.

9 References

Kuhn M, Wing J, Weston S, Williams A, Keefer C, Engelhardt A, Cooper T, Mayer Z, Kenkel B, the R Core Team, Benesty M, Lescarbeau R, Ziem A, Scrucca L, Tang Y, Candan C (2015). *caret: Classification and Regression Training*. R package version 6.0-62, URL https://CRAN.R-project.org/package=caret.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Wickham H, Chang W (2015). *ggplot2: An Implementation of the Grammar of Graphics*. R package version 2.0.0, URL https://CRAN.R-project.org/package=ggplot2.

Williams GJ (2009). "Rattle: A Data Mining GUI for R." *The R Journal*, **1**(2), 45–55. URL http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Williams.pdf.

Williams GJ (2011). *Data Mining with Rattle and R: The art of excavating data for knowledge discovery*. Use R! Springer, New York.

Williams GJ (2015). *rattle: Graphical User Interface for Data Mining in R*. R package version 4.0.0, URL http://rattle.togaware.com/.

This document, sourced from StyleO.Rnw bitbucket revision 46, was processed by KnitR version 1.9 of 2015-01-20 and took 1.7 seconds to process. It was generated by gjw on theano running Ubuntu 14.04.3 LTS with Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz having 4 cores and 3.9GB of RAM. It completed the processing 2015-12-26 08:17:57.