Hands-On Data Science
Sharing R Code — With Style

Graham.Williams@togaware.com

24th March 2017

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

Data scientists write programs to ingest, manage, wrangle, visualise, analyse and model data in many ways. It is an art to be able to communicate our explorations and understandings through a language, albeit a programming language. Of course our programs must be executable by computers but computers care little about our programs except that they be syntactically correct. Our focus should be on engaging others to read and understand the narratives we present through our programs.

In this chapter we present simple stylistic guidelines for programming in R that support the transparency of our programs. We should aim to write programs that clearly and effectively communicate the story of our data to others. Our programming style aims to ensure consistency and ease our understanding whilst of course also encouraging correct programs for execution by computer.

A version of this is included in an upcoming book from CRC Press.

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 Why We Should Care

Programming is an art and a way to express ourselves. Often that expression is unique to us individually. Just as we can often tell the author of a play from their style or the artist from the painting, so we can often tell the programmer from the program coding structures and styles.

As we write programs we should keep in mind that something like 90% of a programmer's time (at least in business and government) is spent reading and modifying and extending other programmer's code. We need to facilitate the task—so that others can quickly come to a clear understanding of the narrative.

As data scientists we also practice this art of programming and indeed even more so to share the narrative of what we discover through our living and breathing of data. Writing our programs so that others understand why and how we analysed our data is crucial. Data science is so much more than simply building black box models—we should be seeking to expose and share the process and the knowledge that is discovered from the data.

Data scientists rarely begin a new project with an empty coding sheet. Regularly we take our own or other's code as a starting point and begin from that. We find code on Stack Overflow or elsewhere on the Internet and modify it to suit our needs. We collect templates from other data scientists and build from there, tuning the templates for our specific needs and datasets.

In being comfortable to share our code and narratives with others we often develop a style. Dictating a style to others is often a sensitive issue. We want freedom to innovate and to express ourselves but we also need consistency in how we do that. Often a style guide helps us as we journey through a new language and gives us a foundation for developing, over time, our own style.

A style guide is useful for sharing our tips and tricks for communicating clearly through our programs—our expression of how to solve a problem or actually how we model the world. We express this in the form of a language—a language that also happens to be executable by a computer. In this language we follow precisely specified syntax/grammar to develop sentences, paragraphs, and whole stories. Whilst there is infinite leeway in how we express ourselves and we each express ourselves differently we share a common set of principles as our style guide.

The style guide here has evolved from over 30 years of programming and data experience. Nonetheless we note that style changes over time. Change can be motivated by changes in the technology itself and we should allow variation as we mature and learn and change our views.

Irrespective of whether the specific style suggestions here suit you or not, do aim when coding to communicate to other human readers in the first instance. When we write programs we *write for others to easily read and to learn from and to build upon.*

2 Naming

1. Files containing R code use the uppercase `.R` extension. This aligns with the fact that the language is unambiguously called "R" and not "r." Ensure that files are under version control such as with github to allow recovery of old versions of the file and multiple people working on the same file.

   Preferred

   ```
   power_analysis.R
   ```

   Discouraged

   ```
   power_analysis.r
   ```

2. Some files may contain support functions that we have written to help us repeat tasks more easily. Name the 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 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

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

3. R binary data filenames end in ".RData". This is descriptive of the file containing data for R and conforms to a capitalised naming scheme.

   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
   ```

3 Multiple File Scripts

5. For multiple scripts associated with a project that have a processing order associated with them use a simple two digit number prefix scheme. Separating by 10's allows additional script files to be added into the sequence later.

   Suggested

```
00_setup.R
10_ingest.R
20_observe.R
30_process.R
40_meta.R
50_save.R
60_classification.R
62_rpart.R
64_randomForest.R
66_xgboost.R
68_h20.R
70_regression.R
72_lm.R
74_rpart.R
76_mxnet.R
80_evaluate.R
90_deploy.R
99_all.R
```

4 Naming Objects

6. **Function names** begin lowercase with capitalised *verbs*. A common alternative is to use underscore to separate words but we use this specifically for variables.

   Preferred

   ```
   displayPlotAgain()
   ```

   Discouraged

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

7. **Variable names** use underscore separated *nouns*. A very common alternative is to use a period in place of the underscore. However the period is often used to identify class hierarchies in R and the period has specific meanings in many database systems which presents an issue when importing from and exporting to databases.

   Preferred

   ```
   num_frames <- 10
   ```

   Discouraged

   ```
   num.frames <- 10
   numframes  <- 10
   numFrames  <- 10
   ```

8. **Function argument names** use period separated *nouns*. Function argument names do not risk being confused with class hierarchies and the style is useful in differentiating the argument name from the argument value. Within the body of the function it is also useful to be reminded of which variables are function arguments and which are local variables.

   Preferred

   ```
   buildCyc(num.frames=10)
   buildCyc(num.frames=num_frames)
   ```

   Discouraged

   ```
   buildCyc(num_frames=10)
   buildCyc(numframes=10)
   buildCyc(numFrames=10)
   ```

9. Keep variable and function names shorter but self explanatory. A long variable or function name is problematic with layout and similar names are hard to tell apart. Single letter names like x and y are often used within functions and facilitate understanding, particularly for mathematically oriented functions but should otherwise be avoided.

   Preferred

   ```
   # Perform addition.
   ```

```
addSquares <- function(x, y)
{
  return(x^2 + y^2)
}
```

Discouraged

```
# Perform addition.

addSquares <- function(first_argument, second_argument)
{
  return(first_argument^2 + second_argument^2)
}
```

5 Comments

10. Use a single # to introduce ordinary comments and separate comments from code with a single empty line before and after the comment. Comments should be full sentences beginning with a capital and ending with a full stop.

    Preferred

```
# How many locations are represented in the dataset.

ds$location %>%
  unique() %>%
  length()

# Identify variables that have a single value.

ds[vars] %>%
  sapply(function(x) all(x == x[1L])) %>%
  which() %>%
  names() %T>%
  print() ->
constants
```

11. Sections might begin with all uppercase titles and sub-sections with initial capital titles. The last 4 dashes at the end of the comment are a section marker supported by RStudio. Other conventions are available for structuring a document and different environments support different conventions.

    Preferred

```
# DATA WRANGLING ----

# Normalise Variable Names ----

# Review the names of the dataset columns.

names(ds)

# Normalise variable names and confirm they are as expected.

names(ds) %<>% rattle::normVarNames() %T>% print()

# Specifically Wrangle weatherAUS ----

# Convert the character variable 'date' to a Date data type.

class(ds$date)
ds$date %<>%
  lubridate::ymd() %>%
  as.Date() %T>%
  {class(.); print()}
```

6 Layout

12. Keep lines to less then 80 characters for easier human reading and fitting on a printed page.

13. Align curly braces so that an opening curly brace is on a line by itself. This is at odds with many style guides. My motivation is that the open and close curly braces belong to each other more so than the closing curly brace belonging to the keyword (`while` in the example). The extra white space helps to reduce code clutter. This style also makes it easier to comment out, for example, just the line containing the `while` and still have valid syntax. We tend not to need to foucs so much any more on reducing the number of lines in our code so we can now avoid Egyptian brackets.

    Preferred

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

    Alternative

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

14. If a code block contains a single statement then curly braces remain useful to emphasise the limit of the code block, however some prefer to drop them.

    Preferred

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

    Alternatives

    ```
    while (blueSky())
      doSomeResearch()
    retireForTheDay()
    ```

    ```
    while (blueSky()) doSomeResearch()
    retireForTheDay()
    ```

7 If-Else Issue

15. R is an interpretive language and encourages interactive development of code within the R console. Consider typing the following code into the R console.

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

After the first closing brace the interactive interpreter identifies a syntactically valid statement (an `if` with no `else`) and so executes it. The following `else` becomes a syntactic error. This will be true irrespective of whether we are interactively typing the commands directly into the R console or we are sending the commands from our editor in Emacs ESS or RStudio to the R console.

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

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

This is not an issue when the if statement is embedded inside a block of code as within curly braces as we might use within a function definition. Here the text we enter is not parsed until we hit the final closing brace.

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

There is no simple solution for the interpreter so we might need to do something less satisfactory for top level statements in a script file or when writing interactively:

```
if (TRUE)
{
  seed <- 42
} else
{
```

```
  seed <- 666
}
```

8 Indentation

16. Use a consistent indentation. I personally prefer 2 spaces within both Emacs ESS and RStudio 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 then try 4 or choose a different font. We still often have limited lengths on lines on some forms of displays where we might want to share our code and about 80 characters seems about 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 as we choose.

Preferred

```
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)
}
```

Not Ideal

```
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)
}
```

17. Always use spaces rather than the invisible tab character.

9 Alignment

18. 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)
```

19. In the same vein we might think to align the `stringr::%>%` operator in pipelines and the `base::+` operator for ggplot2 (Wickham and Chang, 2016) layers. This provides a visual symmetry and avoids the operators being lost amongst the text. Such alignment though requires extra work and is not supported by editors. Also there is a risk the operator too far to the right is overlooked on an inspection of the code.

Preferred

```
ds        <- weatherAUS
names(ds) <- rattle::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))
```

Alternative

```
ds        <- weatherAUS
names(ds) <- rattle::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))
```

10 Sub-Block Alignment

20. An interesting variation on the alignment of pipelines including graphics layering is to indent the graphics layering and include it within a code block (surrounded by curly braces). This highlights the graphics layering as a different type of concept to the data pipeline and ensures the graphics layering stands out as a separate stanza to the pipeline narrative. Note that a period is then required in the `ggplot2::ggplot()` call to access the pipelined dataset. The pipeline can of course continue on from this expression block. Here we show it being piped into a `base::print()` to have the plot displayed and then saved into a variable for later processing. This style was suggested by Michael Thompson.

Preferred

```
ds        <- weatherAUS
names(ds) <- rattle::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))
  } %T>%
  print() ->
plot_cum_rainfall_by_location
```

11 Functions

21. Functions should be no longer than a screen or a page. Long functions generally suggest the opportunity to consider more modular design. Take the opportunity to split the larger function into smaller functions.

22. When referring to a function in text include the empty round brackets to make it clear it is a function reference as in rpart().

23. Generally prefer a single `base::return()` from a function. Understanding a function with multiple and nested returns can be difficult. Sometimes though, particularly for simple functions as in the alternative below multiple returns work just fine.

    Preferred

```
factorial <- function(x)
{
  if (x==1)
  {
    result <- 1
  }
  else
  {
    result <- x * factorial(x-1)
  }

  return(result)
}
```

    Alternative

```
factorial <- function(x)
{
  if (x==1)
  {
    return(1)
  }
  else
  {
    return(x * factorial(x-1))
  }
}
```

12 Function Definition Layout

24. Align function arguments in a function definition one per line. Aligning the = is also recommended to make it easier to view what is going on by presenting the assignments as a table.

Preferred

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

Alternative

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

Discouraged

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

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

Alternative

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

13 Function Call Layout

25. Don't add spaces around the = for named arguments in parameter lists. Visually this ties the named arguments together and highlights this as a parameter list. This style is at odds with the default R printing style and is the only situation where I tightly couple a binary operator. In all other situations there should be a space around the operator.

    Preferred

    ```
    readr::read_csv(file     = "data.csv",
                    skip     = 1e5,
                    na       = ".",
                    progress = FALSE)
    ```

    Alternative

    ```
    readr::read_csv(file="data.csv",
                    skip=1e5,
                    na=".",
                    progress=FALSE)
    ```

    Discouraged

    ```
    read.csv(file = "data.csv", skip =
             1e5, na = ".", progress
             = FALSE)
    ```

26. Arguments to function calls can also be aligned similarly to the function definition. An advantage of doing this is that all but the last argument can easily be commented out during testing of different options in using the function. An idiosyncratic alternative illustrated below places the comma at the beginning of the line. This is actually particularly useful and works well to easily comment out specific arguments except for the first one. Often the first argument is the most important argument and is perhaps even a non-optional argument. Later arguments are often optional and we will explore different options and tune our code by commenting them in and out. This is quite a common style amongst SQL programmers and can be useful for R programming too.

    Preferred

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

    Alternative

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

Discouraged

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

14 Functions from Packages

27. R has a mechanism (called namespaces) 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 base R. 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.

    For common packages we tend not to use namespaces but for less well known packages a namespace at least on first usage provides valuable information. Also, when a package provides a function that has the same name as a function in another namespace it is useful to explicitly supply the namespace prefix.

    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()
```

    Alternative

    The use of the namespace prefix increases the verbosity 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()
```

15 Assignment

28. Avoid using `base::=` for assignment. It was introduced in S-Plus in the late 1990's as a convenience but is ambiguous (named arguments in functions, mathematical concept of equality). The traditional backward assignment operator `base::<-` implies a flow of data and for readability is explicit about the intention.

Preferred

```
a <- 42
b <- mean(x)
```

Discouraged

```
a = 42
b = mean(x)
```

29. The forward assignment `base::->` should generally be avoided. A single use case justifies it in pipelines where logically we do an assignment at the end of a long sequence of operations. As a side effect operator it is vitally important to highlight the assigned variable whenever possible and so out-denting the variable after the forward assignment to highlight it is recommended.

Preferred

```
ds[vars] %>%
  sapply(function(x) all(x == x[1L])) %>%
  which() %>%
  names() %T>%
  print() ->
constants
```

Traditional

```
constants <-
  ds[vars] %>%
  sapply(function(x) all(x == x[1L])) %>%
  which() %>%
  names() %T>%
  print()
```

Discouraged

```
ds[vars] %>%
  sapply(function(x) all(x == x[1L])) %>%
  which() %>%
  names() %T>%
  print() ->
  constants
```

16 Miscellaneous

30. Do not use the semicolon to terminate a statement unless it makes a lot of sense to have multiple statements on the one line. Line breaks in R make the semicolon optional.

    Preferred

    ```
    threshold <- 0.7
    maximum   <- 1.5
    minimum   <- 0.1
    ```

    Alternative

    ```
    threshold <- 0.7; maximum <- 1.5; minimum <- 0.1
    ```

    Discouraged

    ```
    threshold <- 0.7;
    maximum   <- 1.5;
    minimum   <- 0.1;
    ```

31. Do not abbreviate `TRUE` and `FALSE` to `T` and `F`.

    Preferred

    ```
    is_windows  <- FALSE
    open_source <- TRUE
    ```

    Dicouraged

    ```
    is_windows  <- F
    open_source <- T
    ```

32. Separate parameters in a function call with a comma followed by a space.

    Preferred

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

    Dicouraged

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

33. Ensure that files are under version control such as with github to allow recovery of old versions of the file and to support multiple people working on the same files.

17 Further Reading

There are many style guides available and the guidelines here are generally consistent and overlap considerably with many others. I try to capture he motivation for each choice. My style choices are based on my experience over 30 years of programming in very many different languages and it should be recognised that some elements of style are personal preference and others have very solid foundations. Unfortunately in reading some style guides the choices made are not always explained and without the motivation we do not really have a basis to choose or to debate.

The guidelines at Google and from Hadley Wickham and Colin Gillespie are similar but I have some of my own idiosyncrasies. 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. We are however seeing a migration away from the dot in variable names as it is also used as a class separator for object oriented coding. Using the underscore is now preferred.

I also thank the many colleagues over the years who have shaped or argued choices. Thanks also for recent comments from Andrie de Vries, Jacob Spoelstra, Angus Taylor, Michael Thompson, and Doug Dame.

18 References

R Core Team (2017). *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 (2016). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 2.2.1, 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 (2017). *rattle: Graphical User Interface for Data Mining in R*. R package version 5.0.8, URL http://rattle.togaware.com/.

This document, sourced from StyleO.Rnw bitbucket revision 168, was processed by KnitR version 1.15.1 of 2016-11-22 and took 2.7 seconds to process. It was generated by gjw on Ubuntu 16.10.

