Chapter 2

Coding with Style

20200105

Data scientists write programs to ingest, manage, wrangle, visualise, analyse, report on, data in many ways, followed by training, evaluating, deploying and maintaining models. It is an art to be able to communicate our explorations and understandings as data scientists, and of course we do so through language. Whether that is a programming language or a natural language, our focus needs to be on communication and an understanding of who we are communicating to. Indeed it is this last point that is often misunderstood. We regularly find that most programmers spend most of their time reading other peoples programs and increasingly, people not trained in software engineering are reading other people's programs. Programming languages become the means of communicating between people.

We can be as creative and differently expressive and personal in using programming languages as we are in using natural languages. Indeed, we can sometimes identify the authors of narratives written in a particular programming language, just as we might identify a Shakespearean play from a Noël Coward play.

Thus when we communicate with each other using programming languages, we must keep in mind that it truly is a mechanism for communicating to other people.

Of course our programs must be executable by computers but computers generally care little about our programs except that they be syntactically correct. So our focus should be on engaging others to read and understand the narratives we present through our programs. A badly written and presented program is not a pleasure to read. And to boot, it's not a pleasant experience for the computer either, in the sense that where care is not taken, our narratives will more like have bugs.

Please, aim to write programs that clearly and effectively communicate the story of our data to others, no matter how short or long that program is. Think of the other. Have empathy for those who need to read what you write.

In this chapter we present simple stylistic guidelines for programming in R that support and encourage the transparency of our programs. Our programming style aims to ensure consistency and ease our understanding whilst of course also encouraging correct programs for execution by computer. Over time develop your own stylistic nuances and idiosyncrasies.

2.1 Style Matters

20200105

Programming is an art and a way to express ourselves. Often that expression is unique to us individually. Just as we can often ascertain the author of a play or the artist of a painting from their style 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 programmers' time (at least in business and government) is spent reading and modifying and extending other programmers' 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. Our style is personal to us as we innovate and express ourselves and we 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 also 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, when coding do aim to communicate to other readers in the first instance. When we write programs we *write for others to easily read and to learn from and to build upon.*

2.2 Naming Files

20201102

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."

   Preferred

   ```
   power_analysis.R
   ```

   Discouraged

   ```
   power_analysis.r
   ```

2. Some files may contain a single support function. Name the file to match the name of the function defined within the file. If a file contains the support function `my_fancy_plot()` then name the file as below to differentiate it from analysis scripts.

   Preferred

   ```
   my_fancy_plot.R
   ```

   Discouraged

   ```
   utility_functions.R
   MyFancyPlot.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
   ```

2.3 Multiple File Scripts

20200105

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.

   Sometimes this can become a burden. Users find themselves reverting to a single script file for their code. It requires some judgement and discipline to modularise your code in this way, and maybe some assistance too from the integrated development environment being used.

   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
```

2.4 Naming Objects

20201102

6. **Function names** use underscores to separate *verbs*.

   Preferred

```
display_plot_again()
```

   Discouraged

```
DisplayPlotAgain()
displayplotagain()
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. See Section **??** to rename variables automatically. **Preferred**

```
num_frames <- 10
```

   Discouraged

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

2.5 Naming Functions

20201102

8. **Function argument names** use underscore separated *nouns*. Whilst function argument names do not risk being confused with class hierarchies the use of period to separate the nouns has become obsolete in the tidyverse world.

   Preferred

   ```
   build_cyc(num_frames=10)
   ```

   Discouraged

   ```
   build_cyc(num.frames=10)
   build_cyc(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. l
   Preferred

   ```
   # Perform addition.

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

   Discouraged

   ```
   # Perform addition.

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

2.6 Comments

20200105

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() ->
constants
```

11. Sections might begin with all uppercase titles and subsections with initial capital titles. The last four 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.

ds %<>% dplyr::rename_all(rattle::normVarNames)
names(ds)

# Wrangle weatherAUS ----

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

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

2.7 Layout

20200105

12. Keep lines to less then 80 characters for easier 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 (blue_sky())
    {
      open_the_windows()
      do_some_research()
    }
    retireForTheDay()
    ```

    Alternative

    ```
    while (blue_sky()) {
      open_the_windows()
      do_some_research()
    }
    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 (blue_sky())
    {
      do_some_research()
    }
    retire_for_the_day()
    ```

    Alternatives

    ```
    while (blue_sky())
      do_some_research()
    retire_for_the_day()
    ```

    ```
    while (blue_sky()) do_some_research()
    retire_for_the_day()
    ```

2.8 If-Else Issue

20200105

15. Try typing the following code into the R console.

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

After the first closing brace the interpreter identifies and executes a syntactically valid statement (`if` with no `else`). The following `else` is then a syntactic error.

```
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 embedding the if statement inside a block of code as within curly braces since the text we enter is not parsed until we hit the final closing brace.

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

Another solution is to move the `else` to the line with the closing braces to inform the interpreter that we have more to come:

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

2.9 Indentation

20200105

16. Use a consistent indentation. I 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). Try 4 if using a smaller font. Indenting 8 characters is probably too much, makes it difficult to read. There are plenty of tools to reindent to a different level as we choose. Keep lines to less than 80 characters even though displays can now support much longer lines. It seems easier to read.

    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.

2.10 Alignment

20200105

18. Align the assignment operator for blocks of assignments. 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. We might choose to align `tidyr::%>%` in pipelines and `base::+` for **ggplot2** (Wickham *et al.*, 2020) layers for a visual symmetry to avoid the operators being lost amongst the text. This requires extra work and is not supported by editors and 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))
    ```

2.11 Sub-Block Alignment

20200105

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
```

2.12 Function Guidelines

20200105

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))
      }
    }
    ```

2.13 Function Definition Layout

20200105

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

```
show_dial_plot <- function(label       = "UseR!",
                           value       = 78,
                           label_cex   = 3,
                           label_color = "black")
{
  ...
}
```

Alternatives

```
show_dial_plot <- function(label="UseR!",
                           value=78,
                           label_cex=3,
                           label_color="black")
{
  ...
}

show_dial_plot <- function(
                    label="UseR!",
                    value=78,
                    label_cex=3,
                    label_color="black"
                  )
```

Discouraged

```
show_dial_plot <- function(label="UseR!", value=78,
                           label_cex=3,
                           label_color="black")
{
  ...
}

show_dial_plot <- function(label="UseR!",
  value=78,
  label_cex=3,
  label_color="black")
```

2.14 Function Call Layout

20200105

25. Don't add spaces around = 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, progress=FALSE)
    ```

    Discouraged

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

26. For long parameter lists improve readability using a table format aligning on the =.

    Preferred

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

27. All but the final argument to a function call can be easily commented out. However, the latter arguments are often optional and whilst exploring them we will likely comment them out. An alternative puts the comma at the beginning of the line to easily comment out specific arguments except for the first one, which is usually more important and often non-optional. This is common amongst SQL programmers and can be useful for R.

    Usual

    ```
    dialPlot(value       = 78,
             label       = "UseR!",
             label_cex   = 3,
             label_color = "black")
    ```

    Alternative

    ```
    dialPlot(value       = 78
           , label       = "UseR!"
           , label_cex   = 3
           , label_color = "black"
            )
    ```

    Discouraged

    ```
    dialPlot( value=78, label="UseR!",
             label_cex=3, label_color="black")
    ```

2.15 Functions from Packages

20200105

28. 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 `dplyr::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()
```

2.16 Assignment

20200105

29. Avoid using `base::=` for assignment. It was introduced in S-Plus in the late 1990s 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)
    ```

30. 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
    ```

2.17 Miscellaneous

20200105

31. Do not use the semicolon to terminate a statement unless it makes a lot of sense to have multiple statements on 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;
    ```

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

    Preferred

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

    Discouraged

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

33. 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)
    ```

2.18 Good Practise

20200105

34. Ensure that files are under version control such as with gitlab, github or bitbucket. Version control systems support the recovery of previous versions of the file, to always be able to go back to a previous working version of the software, for example. Also these software development repositories support multiple people working on the same project. They also facilitate sharing of the software more broadly, contributing back to the community from which we are all benefiting. If the material is not public then the repositories can be marked as private, or as open source software, gitlab can be installed on your own internal server.

2.19 Style Resources

20200105

There are many style guides available and the guidelines here are generally consistent and overlap considerably with many others. In this chapter I aim to capture the motivation for each choice. My style choices are based on over 30 years of programming in very many different languages. 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.

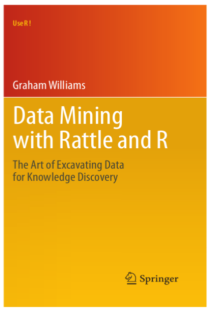
The canonical style guide for the tidyverse is Hadley Wickham's style guide at `https://style.tidyverse.org/`. Other guidelines can be found at Google and Hadley Wickham's older guide and Colin Gillespie. These are similar though with idiosyncratic differences. Also see Wikipedia for an excellent summary of many styles.

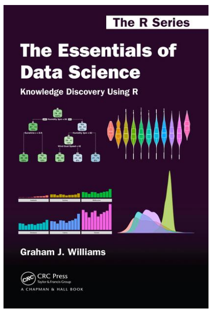
Rasmus Bååth, in The State of Naming Conventions in R, reviews naming conventions used in R, finding that the initial lower case capitalised word scheme for functions was the most popular, and dot separated names for arguments similarly. We are however seeing a migration away from the dot in variable names as it is also used as a class separator for object oriented coding. Using the underscore is now preferred.

Chapter 3

Resources

The Rattle book (Williams, 2011), published by Springer, provides a comprehensive introduction to data mining and analytics using Rattle and R. It is available from Amazon. Rattle provides a graphical user interface through which the user is able to load, explore, visualise, and transform data, and to build, evaluate, and export models. Through its Log tab it specifically aims to provide an R template which can be exported and serve as the starting point for further programming with data in R.


Use R!
Graham Williams
Data Mining with Rattle and R
The Art of Excavating Data for Knowledge Discovery
Springer



The Essentials of Data Science book (Williams, 2017), published by CRC Press, provides a comprehensive introduction to data science through programming with data using R. It is available from Amazon. The book provides a template based approach to doing data science and knowledge discovery. Templates are provided for data wrangling and model building. These serve as generic starting points for programming with data, and are designed to require minimal effort to get started. Visit https://essentials.togaware.com for further guides and templates.