Best Practices for Writing R Scripts

Jennifer Lin

2022-03-25

Overview

For chefs, the R script is equivalent to a recipe. For musicians, it is similar to pages of music. For drivers, it is a map. Regardless of the comparison, the R script is the guide for you to generate the same outcome each time. Statistical programming should be no different from making a hearty dinner or playing a majestic tune. Your outcome in R statistical computing should be like your dinner or tune – equally beautiful every time.

To do this, you will need to develop good habits for writing R scripts and I recommend starting early on. In this memo, I detail my philosophy when it comes to writing code in R, along with the testes I have for formatting my script. I include some examples and encourage you to begin thinking about this process as you embark in your R journey. I should note that I am by no means perfect and this guide should not be treated as so. I welcome feedback, comments and suggestions so that we can work towards writing more beautiful and more useful code.

Before diving into my opinions about good script writing, familiarize yourself with the R Studio window. You, by no means, are required to use R Studio. R, the default app, or other text editors that couple as code interpreters (such as VS Code or Sublime text) can be compatible with R. Proficient programmers might have tastes for these interfaces, but I do not recommend this for beginners.

Formatting the Script

When I started learning to program, I learned in Stata. Back then, my professor had one simple rule for assignments – *"I should be able to change the file path and get the same result"*. This is my general philosophy when it comes to programming itself. But this is hard to achieve unless you have a good commenting system yourself so that you know what you are doing and can direct your reader to the one component they should change and have the code run in the same way. In many ways, there is a different philosophy that code writing should lead to same results on different devices such that you are including code to download the data from the original source. This alleviates the need to change a file path. However, no matter which philosophy you choose to adopt in your own research, one thread holds true –

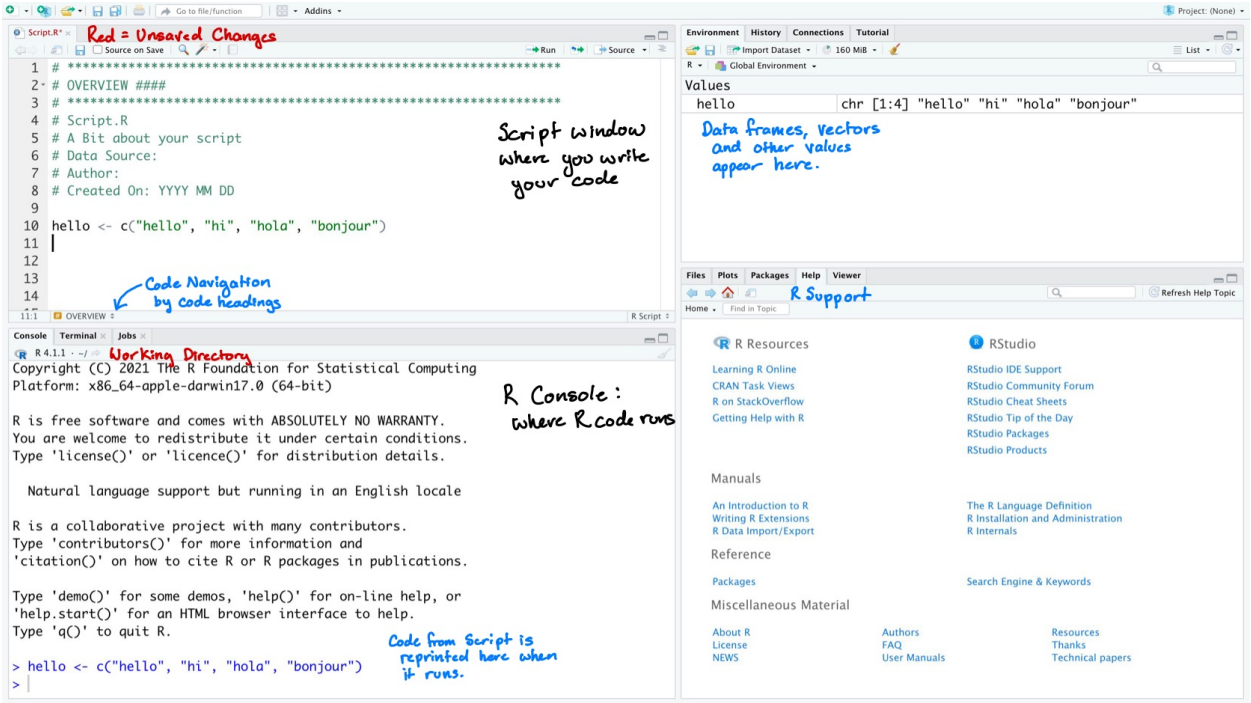

Figure 1: R Studio Interface Overview

you must have clearly commented code. This is why I begin my musings on code writing with comments and not executable code itself.

In my opinion, comments and a succinct way of organizing your code, is of the greatest essence in beautiful code. While you may know what you are currently trying to do, your future self in 1 day, 1 month, 6 months or 2 years might need to revisit the script that you write today and be able to understand the process. Not only that, if you publish a paper, your code will also be published and a user online will need to be able to understand your code in order to understand what you did in your research. Good code comes with clear heading and succinct comments, which I describe below.

Script Organization

In my scripts, I generally follow this outline

1. **Overview**: This section details the general information of your code. I like to include the filename, a short description of the purpose of the code, data information, external links to data (if applicable), my name and the date that this file was created. As you will notice, and I will discuss in more detail later, I like to encase my first level headings within two rows of asterisk chains. This is to distinguish these major sections from the rest of the headings and comments. They make the overall sections easier to spot if a reader is quickly scrolling through code and needs to find a section rather quickly.

```
# ****************************************************************
# OVERVIEW ####
```

```
# ****************************************************************
# Script.R
# A Bit about your script
# Data Source:
# Author:
# Created On: YYYY MM DD
```

2. **Packages and Functions**: I start each R script with all of the packages that I need in the document. This allows me to quickly load everything and allows a replicator to quickly see which packages they should download. Notice that I do not include `install.packages()` code. This is to avoid letting myself (and someone else) download packages that they do not intend to do. Since packages are rather easily searchable on Google, and newer R Studio interfaces allow you to automatically run installations, including installation code is redundant. Next to each package, I include a quick note about the functions that I intend to use from each package. This allows me to be conscientious about the packages that I am loading to avoid loading things I do not need. Down the road, loading too many packages, especially what you do not need can lead to function conflicts, which you will want to avoid. In this section, I also include the custom functions that I write for the code. This allows me to quickly see all the functions at once.

```
# ****************************************************************
# PACKAGES AND FUNCTIONS ####
# ****************************************************************

library(dplyr)        # For %>% , select(), filter()
library(ggplot2)      # For ggplot()
```

3. **Load Data**: In this section, I include code that loads the data. Here, I make it very clear what my working directory is. I include an object for my working directory and use the `here` package to load in the files or I invoke `setwd()`. The `here` package is useful because it assumes that if anyone reading your code has their files set up like your folder, then your code will run the same. Therefore, if you put your data and script in the same folder and include a comment about opening your script from that folder, `here` will know to pull the long front end of your working directory and know where to find the data file, which is in the same folder as your code. Therefore, external readers may not need to change the working directory for the code to run.

4. **Clean Data**: In this section, I typically include all of the code needed to generate a cleaned version of the dataset that I am using. Rarely, if ever, do I load in a pre-cleaned data set. This is for replication purposes as people do not typically post pre-cleaned data unless it is for a survey. Even then, for transparency, I like to include detailed code to show how I rename and recode each of the core variables of interest in my analyses.

5. **Descriptive Statistics**: Depending on the needs of the project that you are running, this section and the next can be rather interchangeable. For me, I like to include

everything that I create for descriptive statistics in one section, which includes the data tables, exports (if applicable) and graphs.

6. **Inferential Statistics**: Here, I include the code for regressions and other machine learning models, along with the tables and graphs that I need for the paper I am writing with this output.

Headings and Subheadings

This might be a feature that is only applicable to newer R Studio interfaces, but R scripts can be programmed to have headings and subheadings for easier code navigation. I like to implement this in the code regardless of whether R Studio can recognize it because it allows my sections to be easily identifiable when a reader is scrolling. R Studio has a navigation menu at the bottom of the R script window that includes all of your headings. Headings and subheadings will be added if a comment is trailed by at least four number signs.

Figure 2: Implementing Headings in R Scripts

Comments

Comments are perhaps the most important part of your code. Even if you do not develop tastes for what good or beautiful code should look like, I would strongly encourage you to develop an appreciation for what good commenting styles look like. For me, my commenting style may differ if I know who the audience is. For example, code that will be published online would get line-by-line commenting since I know there is a diverse audience who might

come across the code but have different levels of understanding in R. If the code is just for me or a co-author, I will still insert detailed commenting for each code chunk, but maybe not necessarily for each and every line.

Regardless of your commenting style, a good comment describes the purpose of the code line or the chunk. Even if the script is for your eyes only, you are likely not going to remember what each block of code does as you progress through a research project. Oftentimes, there is a long gap between when you start your project, to when you submit the manuscript for review, to when you get a revise and resubmit and then to publication. In each of these milestones and many more steps along the way, you will likely encounter many instances where you need to revisit your code and recall what you did. Developing good commenting habits early on allows you to have less stress remembering what you wrote last time. In short, your future self will thank you.

Formatting Code

When it comes to formatting code, the shorter and more succinct you can make your script, the better. It will be less of a hassle for you to follow and easier for any reader to understand what you did. Getting rid of noise in creative ways, such as streamlining code writing to condense repetitive lines, will help you do that, but also being mindful of how you insert code in a script can pay great dividends later on. In this section, I discuss some of my tastes for formatting code and will show some examples. Note that since I am not loading any data into this file, none of this code will actually run. It is simply designed to show you haw I write my R Scripts.

Code in Base R

When writing code in base R, there are a few things I like to keep in mind.

1. Always use the `<-` for object assignments. NEVER use `=`. Yeah, the majority of the time, they do the same thing, but it makes it very clear that you are creating a new object.

2. Always make sure that there is a space before and after the assignment arrow. For example:

```
# This is OK
short_state_abbv <- c("AL", "AK", "AZ", "AR", "CA")

# This is not OK
short_state_abbv<-c("AL", "AK", "AZ", "AR", "CA")
```

3. Do not let your code flow to the ends of the screen. Try to keep it to no more than 80 columns on the margin. This will help ensure that your code is more neatly organized and that your reader is not scrolling too far horizontally such that it make the next line hard to find. Therefore, if you know that you are incorporating code that might

run long, think about how you can break up the chunks. For example, I insert a longer vector below. Since this would be too long to fit in one line, I break it to rows with 5 units each. I incorporate an indent from where the vector starts and keep each subsequent row under the same indent. This way, I can easily count the number of items I inserted and the code is also easy to view.

```
short_state_abbv <- c(
  "AL", "AK", "AZ", "AR", "CA",
  "CO", "CT", "DE", "FL", "GA",
  "HI", "ID", "IL", "IN", "IA"
)
```

4. Make sure that your comments and the start of code chunks are left aligned with no indents. Some R versions will automatically insert tabs based on previous code, which can make the script hard to follow.
5. Write out the full logical outcomes for logical vectors. For example, instead of setting something to `T` for `TRUE`, write out `TRUE`. For example:

```
# Find the file path for a data set

# Do this
data_file <- dir(
  here::here("data/ANES2020"),
  pattern    = ".+.dta$",     # Finds stata Files
  full.names = TRUE,          # List Full File Path Names
  recursive  = TRUE)          # Repeat

# NOT this
data_file <- dir(
  here::here("data/ANES2020"),
  pattern    = ".+.dta$",     # Finds stata Files
  full.names = T,             # List Full File Path Names
  recursive  = T)             # Repeat
```

Code in Tidyverse

In my opinion, formatting nice code in the tidyverse universe is much easier, especially since R can help you detect the beginnings and endings of a chunk of code. Even with this support, I find that there are still a few things I like to do to make sure my code is as neat as possible.

1. At the start of the new object, make sure that it is left aligned. Insert tabs to signal the start of a new line with a new command. Insert two tabs for components within a subsequent function.
2. Make sure that the parentheses that close a function align, tab-wise, with the function that opened it.
3. Make sure to insert spaces before and after the `%>%`. Ideally, these are their own line

following a parentheses close.

I illustrate these points using the code below:

```
office_quotes_clean <- office_quotes %>%
  mutate(
    # Generate unique ID for scenes using episode numbers
    episodeID = paste0(
      "S", season, "E", episode, "Sc", scene
    )
  ) %>%
  rename(
    # Rename character variable to prevent R from confusing
    # this with the data type
    Team = character
  )
```

Notice how the object `office_quotes_clean` starts as the most left-aligned line. The `mutate()` and `rename()` each start one tab in and their contents are two tabs inwards. I close the `mutate()` and `rename()` parentheses on a new line, which is aligned to the original start.

Code in For loops, functions and if, else Statements

For me, writing for loops, functions and if-else statements follow the same structure as the tidyverse code chunks. The difference here is that the `for` or `if` starts the chunk and therefore is the most left aligned. Components that fall into this are tabbed inwards accordingly but align with neighboring code so that it is readable.

Here is an example of an if-else statement I wrote to loop through many, many JSON files to parse Congressional bill co-sponsors.

```
for (i in 1:length(Data_116)) {
  if (length(Data_116[[i]]$cosponsors) > 0){
    # Read file in as a data frame
    list_csp <- data.frame(
      # Extract co-sponsor names
      t(sapply(Data_116[[i]]$cosponsors,
               function(x) c(x, recursive = FALSE))),
      # Extract bill ID
      bill = Data_116[[i]]$bill_id)
    # Add it to the co-sponsorship list
    csp[[i]] <- list_csp
  }
}
```

Concluding Thoughts

The pointers that I outlined here are by no means a hard and fast set of rules that you ought to follow to have clean R script. However, I hope you can use these pointers to begin thinking about what clean code should look like. I encourage you to develop and refine your tastes for beautiful code and begin to develop a consistent way of keeping scripts organized and well commented. After all, as I have stated many times over in this document, your future self will thank you.