Programming Best Practices in R

Eugene Katsevich

2023-09-04

Contents

When programming, one often wishes to get to the answer as quickly as possible. However, programming quickly jeopardizes the correctness and reproducibility of the code that is written. This document lays out a number of programming best practices to promote the correctness and reproducibility of code, which may take additional time to master at first but which save time in the long run. This document is geared towards R programming, though many of the principles presented here are language-agnostic.

1 Code readability

Code readability refers to how easily a programmer can understand and interpret a piece of code. It is an essential aspect of software development because readable code is easier to maintain, faster to debug, more collaborative, and less prone to errors. Code readability can be broadly categorized into two main aspects: code style and code transparency. *Code style* pertains primarily to the format and presentation of the code. *Code transparency* delves deeper into the logic and structure of the code. It refers to how straightforwardly the functionality, logic, or operations are conveyed.

1.1 Example

Consider the following two examples:

Poor code readability:

```
a = sum(dat[dat[,1]>5 & dat[,2]<10,3])
b = mean(dat[dat[,1]>5 & dat[,2]<10,3])
print(paste('Sum =',a,', Mean =',b))
```

Good code readability:

```
# Library for data manipulation
library(dplyr)

# Analysis parameters
MIN_AGE <- 5
MAX_SCORE <- 10

# Extract summary performance metrics for subset of observations
summary_data <- dat |>
  filter(age >= MIN_AGE & score <= MAX_SCORE) |>
  summarize(
    sum_performance = sum(performance_metric),
    mean_performance = mean(performance_metric)
  )

# Print the computed results
cat(sprintf("Sum of Performance: %f | Mean of Performance: %f",
            summary_data$sum_performance,
            summary_data$mean_performance))
```

In this section, we will discuss how to write code like that in the second snippet above.

1.2 Code style

Code should adhere to the tidyverse style guide, which includes guidance on the following aspects of code:

- Naming conventions
- Spacing and indentation
- Commenting

You can use the `styler` package to automatically conform your spacing and indentation to the tidyverse style guide.

1.3 Code transparency

To write transparent code, follow these guidelines:

- Use tidyverse paradigms as much as possible (e.g. `dplyr` summaries instead of apply operations)
- Use names, rather than indices, for subsetting (e.g. `results["mse", "lasso"]` versus `results[2,4]`)
- Use named arguments in function calls, especially with more than one argument (e.g. `rbinom(3, 1, 0.5)` versus `rbinom(n = 3, size = 1, prob = 0.5)`)
- Put logically related chunks of code together into code blocks, with a comment describing the thrust of that code block.

2 Code safety

Code safety entails practices that proactively guard against, identify, and handle programming errors or unwanted outcomes.

2.1 Code portability

One of the challenges in R programming, especially when sharing your code with others or transferring it between different machines, is ensuring code portability. *Code portability* ensures that your R script or project can be executed seamlessly in different environments without any hitches. Here's how to enhance the portability of your R code:

Avoid hardcoded paths. Hardcoding paths, such as `C:/Users/JohnDoe/Documents/mydata.csv`, can break the code when run on a different machine or if files are moved. Instead, always aim to use relative paths or dynamic paths that adjust based on the current directory. This ensures that as long as the directory structure remains consistent, the paths will always be correct.

Use R Projects. R Projects, a feature of RStudio, allow you to maintain a consistent working directory regardless of where the project is located on the machine. When you open an R Project, RStudio sets the working directory to the project's root directory. This allows you to use relative paths effectively. To start an R Project, simply choose `File > New Project` in RStudio.

2.2 Defensive programming

When writing a function, the goal is for the function to either produce the intended output or an informative error message for any given input. Since a wide variety of inputs are possible, *defensive programming* is required to achieve this. Defensive programming in R entails anticipating potential issues, pitfalls, or mistakes in the code and implementing strategies to handle them gracefully.

Input validation: One of the most common tactics in defensive programming is input validation. Before processing, check if the provided inputs are valid or in the expected format. For instance, if a function expects a numeric vector, verify the input type before proceeding.

```
compute_mean <- function(data_vector) {
  if (!is.numeric(data_vector)) {
    stop("The input data_vector must be numeric.")
  }
  return(mean(data_vector))
}
```

Use informative error messages: When an error condition is detected, provide a clear and informative error message. This not only prevents silent failures but also helps users or developers identify and fix the problem.

```
safe_divide <- function(numerator, denominator) {
  if (denominator == 0) {
    stop("Error: Division by zero is not allowed.")
  }
  return(numerator / denominator)
}
```

3 Automation and reproducibility

To ensure the robustness and reliability of your analyses, strive for automation and reproducibility. This approach ensures your work remains consistent and easily repeatable.

3.1 Automate manual operations

Manual operations, including moving files, creating directories, and saving figures, are inefficient and error-prone. As many operations as possible should be automated instead.

```
# Instead of manually downloading a file, use R to do it programmatically:
download.file(url = "https://example.com/data.csv", destfile = "data/data.csv")
```

```
# Instead of manually saving figures, use ggsave():
ggsave("plots/my_plot.png", plot = p)
```

3.2 Embrace modularity

Avoid repetitive code. Repetition not only lengthens your script but also increases the chance for mistakes.

```
# Bad practice: Repetitive code
data$age[data$age < 0] <- NA
data$score[data$score < 0] <- NA

# Good practice: Create a function
replace_negatives_with_NA <- function(variable) {
  variable[variable < 0] <- NA
  return(variable)
}

data$age <- replace_negatives_with_NA(data$age)
data$score <- replace_negatives_with_NA(data$score)
```

3.3 Name constants in your scripts

"Magic numbers" are unexplained numbers in your scripts:

```
# Bad practice: Magic number
if (x > 30) ...

# Good practice: Using a named constant
MAX_AGE <- 30
if (x > MAX_AGE) ...
```

Descriptive constant names provide clarity. Furthermore, especially if these constants are used in multiple places throughout your script, updating them becomes as simple as changing one line of code. It is also advisable to put all such constants together, near the top of the script.

3.4 Produce results by running entire scripts

All results should be produced by writing scripts and then executing those scripts in their entirety. While pieces of the script can be run manually during development, run the entire script on the data when the time comes to actually produce a result.

4 Managing package dependencies

R's rich ecosystem of packages is one of its strengths. However, with the increasing number of packages, managing dependencies becomes essential to ensure the reproducibility and reliability of your code. This section provides some best practices and recommended tools for handling package dependencies.

4.1 Namespace conflicts

When you load multiple packages, there is potential for functions with the same name to clash, leading to ambiguity in your code. This is because different packages can have functions that share the same name.

4.1.1 Using `::` for explicit namespaces

One way to deal with such conflicts is by explicitly specifying the package's namespace when calling the function using the `::` syntax. This ensures that the correct function from the desired package is called. For example, if both packages `A` and `B` have a function named `fun()`, and you want to use the function from package `A`, you can do:

```
A::fun()
```

4.1.2 `conflicted` package

Another solution is to use the `conflicted` package. When loaded, `conflicted` will prevent you from using any function that has a namespace conflict, prompting you to specify which one you want:

```
library(conflicted)
conflict_prefer("fun", "A") # Always use fun from package A
```

4.2 Managing package versions with `renv`

For the sake of reproducibility and easy collaboration, it is essential to track which versions of R packages were used for a given analysis. The `renv` package can help by creating isolated, reproducible R environments. Using `renv`, you can snapshot your current package versions, ensuring that others (or yourself in the future) can replicate your environment:

```
# initializing renv
renv::init()
```

By doing this, `renv` will create a snapshot of your current package versions and save it in a `renv.lock` file. This file can then be shared, ensuring that the same package versions are used when your code is run elsewhere.

4.3 Best practices

Lastly, always load required libraries at the top of your R scripts. This not only makes it clear which packages are necessary but also ensures that potential namespace conflicts are identified early on, leading to more predictable and stable code.

5 Code speed

When optimizing the execution speed of your code, it's essential to strike a balance between code readability and efficiency. However, as Donald Knuth famously stated, 'premature optimization is the root of all evil.' Focus on writing clean and functional code first. Once your code works correctly, you can then consider optimizing the most computationally intensive parts if necessary.

5.1 Vectorization

R is a vectorized language, which means that operations can be performed on entire vectors rather than looping over individual elements. For example, instead of using a loop to square each element of a vector, you can simply square the vector directly:

```r
numbers <- c(1, 2, 3, 4, 5)

# Non-vectorized operation using a loop
squared_numbers <- vector("numeric", length(numbers))
for (i in seq_along(numbers)) {
  squared_numbers[i] <- numbers[i]^2
}

# Example of vectorized operation
squared_numbers <- numbers^2
```

5.2 Factoring code out of loops

Often, parts of the code inside a loop don't depend on the loop variable and can be taken outside the loop, leading to efficiency gains. For example, if you're repeatedly computing something within a loop that doesn't change, compute it once outside the loop.

```r
# Inefficient loop: Compute mean of the entire dataset in each iteration
for (i in 1:n_bootstrap) {
  sample <- sample(data_points, size = length(data_points), replace = TRUE)
  mu_data <- mean(data_points) # This is unnecessary in the loop
  bootstrap_means[i] <- mean(sample) - mu_data
}

# Optimized loop: Factor out the mean computation of the dataset
bootstrap_means_optimized <- numeric(n_bootstrap)
mu_data <- mean(data_points)
for (i in 1:n_bootstrap) {
  sample <- sample(data_points, size = length(data_points), replace = TRUE)
  bootstrap_means[i] <- mean(sample) - mu_data
}
```