Keeping Your Code Readable

When writing R code (or any other programming language), it is important to use a clear and consistent style that is free from errors. This helps make your code easier to read and understand. In this reading, you will learn some best practices to follow when writing R code. You will also go through some tips for identifying and fixing errors in R code, also known as debugging.

Style

Using a clear and consistent coding style generally makes your code easier for others to read. There's no official coding style guide that is mandatory for all R users. But over the years, the wider community of R users has developed a coding style based on shared conventions and preferences. You can think of these conventions as the unwritten rules of R style.

There are two main reasons for using a consistent coding style:

- If you are working with collaborators or teammates, using a consistent style is important so that everyone can easily read, share, edit, and work on each other's code.
- If you are working alone, using a consistent style is important because it makes it much easier and faster to review your code later on and fix errors or make revisions.

Let's go over a few of the most widely accepted stylistic conventions for writing R code.

Naming

	Guidance	**Examples of best practice**	**Examples to avoid**

Files	File names should be meaningful and end in `.R`. Avoid using special characters in file	`# Good`
		`explore_penguins.R`
		`annual_sales.R`
			`# Bad`
			`Untitled.r`
			`stuff.r`

	names—stick with numbers, letters, dashes, and underscores.		
Object names	Variable and function names should be lowercase. Use an underscore `_` to separate words within a name. Try to create names that are clear, concise, and meaningful. Generally, variable names should be nouns.	`# Good` `day_one`	`# Bad` `DayOne`
	Function names should be verbs.	`# Good` `add ()`	`# Bad` `addition ()`

Syntax

	Guidance	**Examples of best practice**	**Examples to avoid**

Spacing	Most operators (`==`, `+`, `-`, `<-`, etc.) should be surrounded by spaces.	`# Good`
		`x == y`
		`a <- 3 * 2`
			`# Bad`
			`x==y`
			`a<-3*2`

	Always put a space *after* a comma (never before).	`# Good`
		`y[, 2]`
			`# Bad`
			`y[,2]`
			`y[ ,2]`

	Do not place spaces around code in parentheses or square brackets (unless there's a comma, in which case see above).	`# Good`
		`if (debug) do(x)`
		`species["dolphin", ]`
			`# Bad`
			`if ( debug ) do(x)`
			`species[ "dolphin" ,]`

	Place a space before left parentheses, except in a function call.	`# Good`
		`sum(1:5)`
		`plot(x, y)`
			`# Bad`
			`sum (1:5)`
			`plot (x, y)`

Curly braces	An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line (unless it's followed by an `else` statement). Always indent the code inside curly braces.	`# Good`
		`x <- 7`
		`if (x > 0) {`
		`  print("x is a positive number")`
		`} else {`
		`  print ("x is either a negative number or zero")`
		`}`
			`# Bad`
			`x <- 7`
			`if (x > 0)`
			`{`
			`  print("x is a positive number")`
			`}`
			`else {`
			`  print ("x is either a negative number or zero")`
			`}`

Indentation	When indenting your code, use two spaces. Do not use tabs or mix tabs and spaces.	-	-
Line length	Try to limit your code to 80 characters per line. This fits nicely on a printed page with a reasonably sized font.	-	-

	Note that many style guides mention to never let a line go past 80 (or 120) characters. If you're using RStudio, there's a helpful setting for this. Go to Tools -> Global Options -> Code -> Display, and select the option Show margin, and set margin column to 80 (or 120).		
Assignment	Use `<-` , not `=` , for assignment.	`# Good` `z <- 4`	`# Bad` `Z = 4`

Organization

	Guidance	**Examples of best practice**	**Examples to avoid**
Commenting	Entire commented lines should begin with the comment symbol and a single space: `#`.	`# Good` `# Load data`	`# Bad` `Loaddata`

Resources

- Check out this tidyverse style guide to get a more comprehensive breakdown of the most important stylistic conventions for writing R code (and working with the tidyverse).

- The styler package is an automatic styling tool that follows the tidyverse formatting rules. Check out the styler webpage to learn more about the basic features of this tool.

Debugging

Successfully debugging any R code begins with correctly diagnosing the problem. The first step in diagnosing the problem in your code is to understand what you expected to occur. Then, you can identify what actually occurred, and how it differed from your expectations.

For example, imagine you want to run the **glimpse()** function to get a summary view of the *penguins* dataset. You write the following code:

```
Glimpse(penguins)
```

When you run the function, you get the following result:

```
Error in Glimpse(penguins) : could not find function "Glimpse"
```

You were expecting a display of the dataset. Instead, you got an error message. What went wrong? In this case, the problem can be diagnosed as a stylistic error: you wrote `Glimpse` with a capital "G," but the code is case sensitive and requires a lowercase "g." If you run the code `glimpse(penguins)` you'll get the result you expected.

When diagnosing the problem, it is more likely that you–and anyone else who might help debug your code–will understand the problem if you ask the following questions:

- What was your input?
- What were you expecting?
- What did you get?
- How does the result differ from your original expectations?
- Were your expectations correct in the first place?

Some bugs are difficult to discover, and finding the cause of the problem can be challenging. If you come across error messages or you need help with a bug, start by searching online for information about it. You might discover that it's actually a common error with a quick solution.

Resources

- For more information on the technical aspects of debugging R code, check out Debugging with RStudio on the RStudio Support website. RStudio Support is a great place to find answers to your questions about RStudio. This article will take you through the R debugging tools built into RStudio, and show you how to use them to help debug R code.

- To learn more about problem-solving strategies for debugging R code, check out the chapter on Debugging in Advanced R. Advanced R is a great resource if you want to explore the finer details of an R topic and take your knowledge to the next level.