1d. A Minimal Set of Good Practices

Martin Alfaro
PhD in Economics

REMARKS

We conclude this chapter by reviewing various principles to write code. They represent a minimum set of good practices that apply regardless of the programming language used. By adhering to these guidelines, you'll be able to write clear and maintainable code. I suggest incorporating these suggestions into your workflow from the very beginning, as it'll render the learning process smoother.

Several of the suggestions we present might seem inconsequential to you at this point, or give the impression that their importance is exaggerated. For small projects, there's some truth to this—they won't have a substantial impact. However, as projects grow in size and complexity, following these principles becomes crucial. [1] It's not uncommon to revisit your own code after a few months (or even days!) and struggle to understand it. When this occurs, extending the code becomes a daunting task, often resulting in non-reusable code.

As usual, the devil is in the details. Thus, the challenge lies in interpreting and implementing these suggestions effectively. Many of them rely on the reader's judgment, as they require a subjective assessment of when and how to apply them. For example, one suggestion we'll present is to use clear and descriptive names. However, determining what constitutes "clear" or "unclear" is ultimately a matter of personal interpretation. Hopefully, the implementation of the suggestions will become apparent as we move forward and apply these concepts.

WRITE EASY-TO-READ CODE

Code is read more often than it's written. I can't stress enough the importance of this statement. It has a stark implication: write code that is easy to read, even if this requires additional effort or some extra verbosity.

If you end up coding extensively in your future career, you'll likely learn this lesson the hard way. I certainly did. One of the first times I had to reuse an old script, I was completely clueless about my own code. As a consequence, I had to rewrite the entire script from scratch, as making sense of the old code would've taken longer.

If you end up coding extensively in your future career, you'll likely learn this lesson the hard way, just as I did. One of my earliest experiences with reusing old code was a humbling one - I was completely baffled by my own script, and rewriting it from scratch proved to be faster than trying to decipher the original code.

> **Remark**

> If you're concerned that more readable code requires excessive typing, remember that you can use Tab Completion to autocomplete names. Additionally, AI tools like GitHub Copilot will suggest code while you type, thereby also mitigating the inconvenience.

To illustrate this point, suppose you're reading a script that cleans some data. Imagine in particular that you come across a line that has two possible expressions: `na.rm=TRUE` and `dropmissing=true`. Even if you're unfamiliar with the language's syntax or the concept of missing data, you could likely infer the meaning of `dropmissing=true`: discard entries with no values provided. On the contrary, `na.rm=TRUE` offers no clue. Although this example may appear somewhat abstract, it actually highlights how to discard missing observations in R and Julia: `na.rm=TRUE` corresponds to R and `dropmissing=true` to Julia. [2]

The example also reveals why typing `na.rm=TRUE` might be tempting: it's short and requires less typing. However, it's essential to weigh the long-term benefits of readable code. Although typing more might seem inconvenient in the short term, it represents a minimal effort compared to the future costs of ambiguous code. Moreover, you may feel confident that you'll remember what you intended to write, but it's common to be puzzled by code you wrote just days before.

The benefits of clear code become apparent when you read a script written in an unfamiliar programming language: if the code is well-written and clearly structured, you might grasp the logic and tasks being performed. [3]

Several tips arise as a consequence of this. We list them below.

USE NAMES WITH A CLEAR MEANING

Clear names don't only refer to variables and functions, but files as well. In particular, you should avoid abbreviating. Code editors can be very helpful in this regard, by offering word auto-completion. This feature requires typing the first letters of each word and then pressing Tab. [4]

Avoiding abbreviations has the additional benefit of making it easier to substitute expressions. For instance, suppose you name a variable `re`, and later decide to replace it with a different name. Then, the substitution process becomes more challenging, as the search will also capture functions like `replace` and `repeat`.

Finally, using descriptive names reduces the need for comments. If the code is self-explanatory, comments become only necessary for exceptionally complex code or clarifications that go beyond what's written.

INDENT AND ALIGN YOUR CODE

The implementation details of this suggestion have already been covered in the previous section. For further details, please refer to that section.

When writing code sequentially, VS Code automatically provides indentation. You can also format a selected portion of code by pressing Ctrl + k, followed by f. Alternatively, to format the entire script, use the shortcut Alt + Shift + L. [5]

To illustrate how this feature improves readability, consider the following (somewhat exaggerated) example.

```
if x>0 display("x is a positive number") else display("x is a non-positive number") end

function example(a,b)
x=a/10#rescaling x
output=2*b+x
return output
end
```

```
if x > 0
    display("x is a positive number")
else
    display("x is a non-positive number")
end

function example(a, b)
    x      = a / 10               # rescaling x
    output = 2 * b +  x

    return output
end
```

To further improve readability, I suggest also aligning code blocks. Several plugins in VS Code can assist with this task, such as "Better Align" and "Cursor Align". Their use is demonstrated below.

```
this_is_a_variable = 1
x = 3
another_var = 2

computations_here = x + another_var
more_calcs = this_is_a_variable * another_var
```

```
this_is_a_variable = 1
x                  = 3
another_var        = 2

computations_here  = x + another_var
more_calcs         = this_is_a_variable * another_var
```

FOOTNOTES

[1.] For real-world examples, read "Brief Story" on this link or the perspective of a former worker from Oracle.

[2.] Python also tends to employ abbreviations that can hinder readability. For instance, to count the number of characters of a variable `x`, Python calls `len(str(x))` while Julia calls `length(string(x))`.

[3.] One way to learn how to write clear code is through AI chatbots, which are pretty good at providing highly readable examples.

[4.] You could eventually use the option "find and replace", whereby you substitute abbreviations for their full name. However, this is error-prone, and you may end up replacing unrelated expressions by substituting all words at once.

[5.] Unlike Python, Julia only uses indentation for readability purposes. It doesn't affect how code is executed.