

Editing Code for Rigor and Reproducibility

Brooke Anderson

Separating writing and editing



Write your code fast...



... but edit it slowly.



Editing code

The aims of editing code are to make it **easier to understand**, which will help make it **easier to maintain**.



Editing code

IT TOOK SOME EXTRA WORK TO BUILD, BUT NOW WE'LL BE ABLE TO USE IT FOR ALL OUR FUTURE PROJECTS.



HOW TO ENSURE YOUR CODE IS NEVER REUSED

LET'S NOT OVERTHINK IT; IF THIS CODE IS STILL IN USE THAT FAR IN THE FUTURE, WE'LL HAVE BIGGER PROBLEMS.



HOW TO ENSURE YOUR CODE LIVES FOREVER

Editing code

In this lecture, we'll cover several steps you can take to edit your code. These include:

- ▶ Change names to be good names
- ▶ Break up monolithic code
- ▶ Remove dead-end code
- ▶ Add useful comments

After the lecture, we'll have a lab, where you'll try applying some of these ideas in an example code script.

Use good names

LYMPHOCYTES AND THEIR FUNCTIONS

PLASMA B CELLS



CHURN OUT
ANTIBODIES

NAÏVE B CELLS



TRY TO STOP
PATHOGENS BY
ASKING NICELY

MEMORY B CELLS



VERY QUIETLY SING
"MEMORY" FROM
CATS AT ALL TIMES

REGULATORY
B CELLS



REQUIRED BY
LOCAL ORDINANCE

CD8+ T CELLS



MELEE COMBAT

CD4+ T CELLS



SCREAM AT
OTHER CELLS

GAMMA-DELTA
T CELLS



UNKNOWN/
CLASSIFIED

CDRw+ T CELLS



REWRITABLE,
700MB

DVD+R T CELLS



DIFFERENT FROM
DVD-R, THOUGH NO
ONE IS SURE HOW

NATURAL
KILLER CELLS



NAMED BY THE
WORLD'S COOLEST
IMMUNOLOGIST

ILC1, ILC2, AND
ILC3 CELLS



NAMED BY A
SIGNIFICANTLY LESS
COOL IMMUNOLOGIST

D CELLS



LARGER THAN C AND
AA CELLS, USED IN
OLD FLASHLIGHTS

Use good names

NATURAL
KILLER CELLS



NAMED BY THE
WORLD'S COOLEST
IMMUNOLOGIST

ILC1, ILC2, AND
ILC3 CELLS



NAMED BY A
SIGNIFICANTLY LESS
COOL IMMUNOLOGIST

Use good names

If you use good names, it makes your code self-documenting—in other words, the name by itself gives the reader information about what the code is doing.

Edit your code to have good names for, among others:

- ▶ objects (e.g., dataframes, vectors)
- ▶ columns in dataframes
- ▶ functions
- ▶ files

The tidyverse style guide includes advice for naming things in R.

Good names for objects

One common edit will be changing the names of objects from **metasyntactic variables** to better variable names.

Metasyntactic variables are a fancy name for placeholder variable names. Common examples include:

- ▶ foo, bar, baz (C, C#)
- ▶ spam, ham, eggs (Python)
- ▶ toto, tata, titi, tutu (French)
- ▶ pippo, pluto, paperino (Italian)

They're often used by coders when they're editing fast. That's fine, but be sure to edit them out in favor of better names.

Good names for objects

Here are some suggestions drawn from the tidyverse style guide for syntax. Consider selecting object names that:

- ▶ Convey what's in the object (e.g., `animal_calls` versus `foo`)
- ▶ Use only lowercase (e.g., `animal_calls` versus `Animal_calls`)
- ▶ Use underscores to make words in object name easier to read (e.g., `animal_calls` rather than `animalcalls`)
- ▶ not too short, not too long (although tab completion can help)
- ▶ Use nouns for objects that are data containers

Good names for columns

- ▶ Consider all the guidelines from naming objects
- ▶ There is one exception—it's fine to have the same column name in columns of different dataframes (whereas you can only one object with a given name per R environment)
- ▶ Avoid spaces in column names. You can force them to work, but it's a pain.

```
my_data %>%
  mutate(`A column with a long, Tricky name` =
         factor(`A column with a long, Tricky name`),
         better_name = as.numeric(better_name))
```

Fixing column names

Often, you'll have problems with column names when you read data in from an Excel file or other spreadsheet-based file.

There's an R package that can automate some basic improvements to the column names. The `janitor` package has a function called `clean_names` that:

- ▶ Converts to lowercase
- ▶ Replaces spaces with underscores
- ▶ Removes special characters (anything expect letters, numbers, and underscores)

Fixing column names

Here's an example of a simple dataframw with bad column names:

```
## # A tibble: 3 x 2
##   `Column with Long name` `Shorter Name!` 
##   <int> <chr>
## 1 1     a
## 2 2     b
## 3 3     c
```

Fixing column names

You can make a lot of improvements to the column names using `clean_names` from the `janitor` package:

```
## # A tibble: 3 x 2
##   column_with_long_name shorter_name
##           <int>     <chr>
## 1                   1 a
## 2                   2 b
## 3                   3 c
```

Fixing column names

You can finish with customized changes with `rename` from `dplyr`:

```
## # A tibble: 3 x 2
##   longer_name shorter_name
##       <int>     <chr>
## 1          1      a
## 2          2      b
## 3          3      c
```

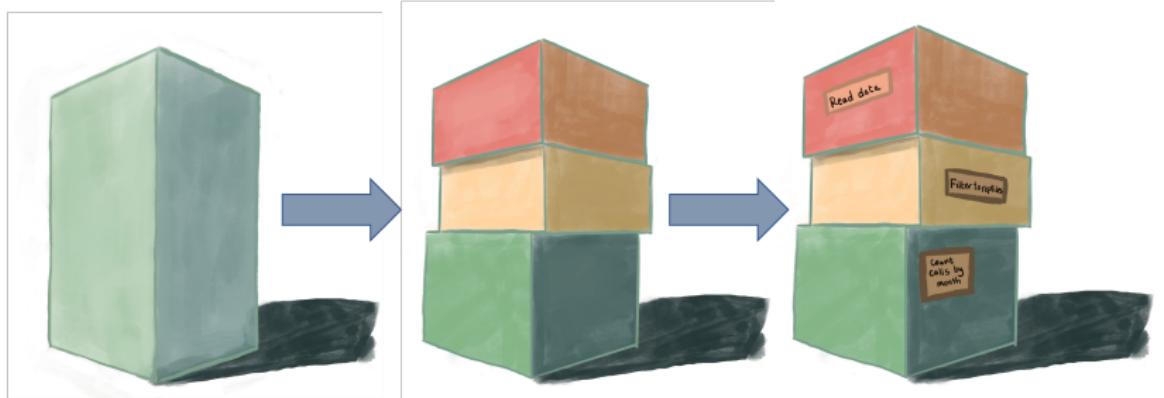
Break up monolithic code

One big aim of editing is to break up monolithic code to divide it into separate blocks and clarify what each block does.

Code that is easier to maintain and understand will often have more white space and clearer structure.

[Cartoon view—code with structure versus code all in one piece]

Break up monolithic code



Code scripts are often “monolithic” when you write them quickly

Edit the script to break it up into related subtasks. This provides some structure and order in the script.

Then use code comments to document each section so you can quickly navigate to a certain subtask.

Break up monolithic code

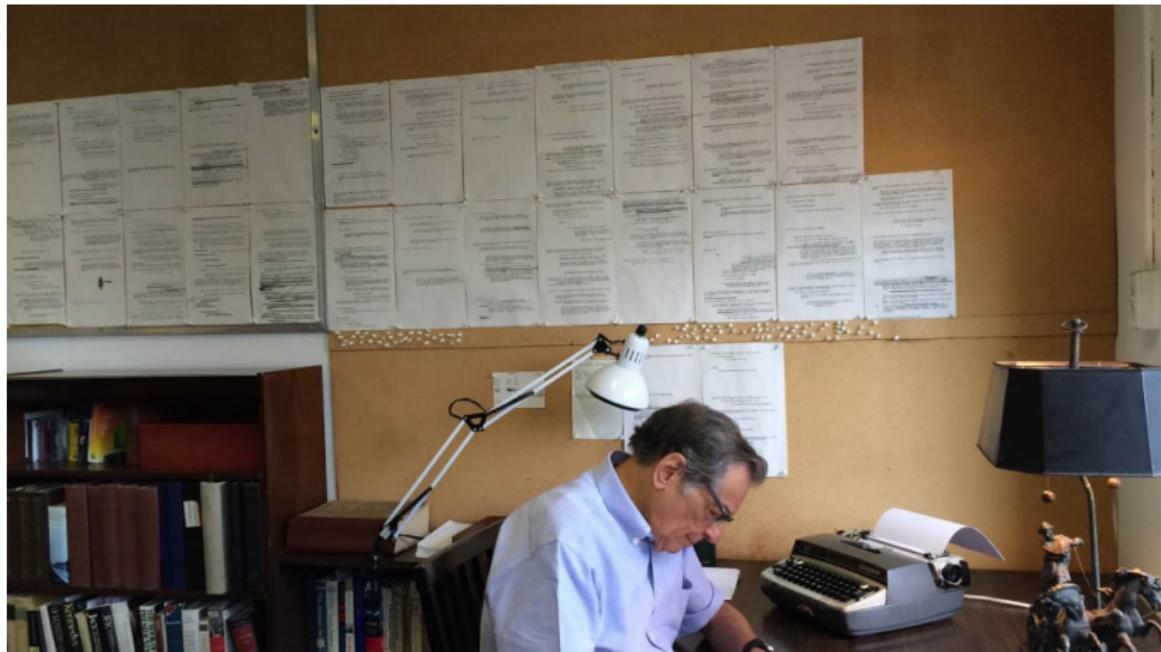


Break up monolithic code

- ▶ Identify sections of code that go together to solve a discrete part of your overall task
- ▶ Move calls to be near others doing the same subtask. For example, move all library calls (and function definitions) to the top of the code script
- ▶ Separate these sections using white space between sections (“Returns are free” in R!)
- ▶ Use code comments to document each section

Break up monolithic code

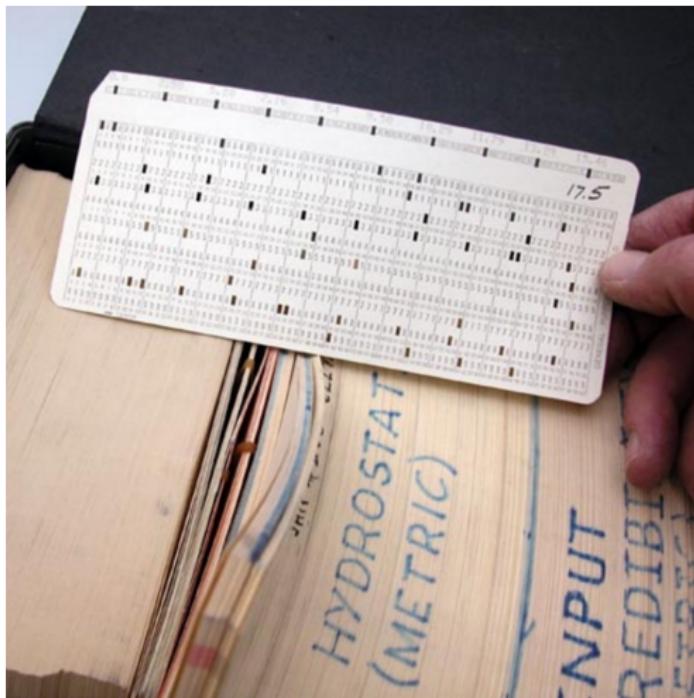
"I can't start writing until I've thought it through and can see it whole in my mind." —Robert Caro



Break up monolithic code

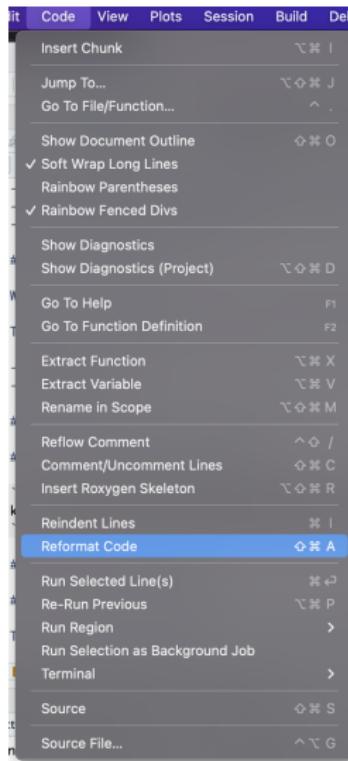
Don't be afraid to split calls across lines.

In general, you should try to keep each line of your code to 80 characters or fewer.



Break up monolithic code

RStudio has a tool that can help with character lines and indenting:



Break up monolithic code

There are also some packages that help to identify and fix some of these formatting issues, as well as complying with a style guide.

One example is the `lintr` package:

<https://lintr.r-lib.org/>

Breaking up monolithic code

In RMarkdown files, separate code into more code chunks.

You may find it helpful to name these chunks (so you can find and fix bugs).

```
```{r clean_column_names}
library(janitor)
df <- clean_names(df)
df
```
```

Add useful comments

```
2230  /*
2231   * If the new process paused because it was
2232   * swapped out, set the stack level to the last call
2233   * to savu(u_ssav).  This means that the return
2234   * which is executed immediately after the call to aretu
2235   * actually returns from the last routine which did
2236   * the savu.
2237   *
2238   * You are not expected to understand this.
2239 */
```

Add useful comments

Don't go overboard with your comments. If the code is self-documenting (clear on its own by using good function names and object names), don't repeat with a comment.

For example, you don't need the comment here:

```
# Rename column `old_name` to `new_name`
df <- df %>%
  rename(new_name = old_name)
```

Edit out dead-ends

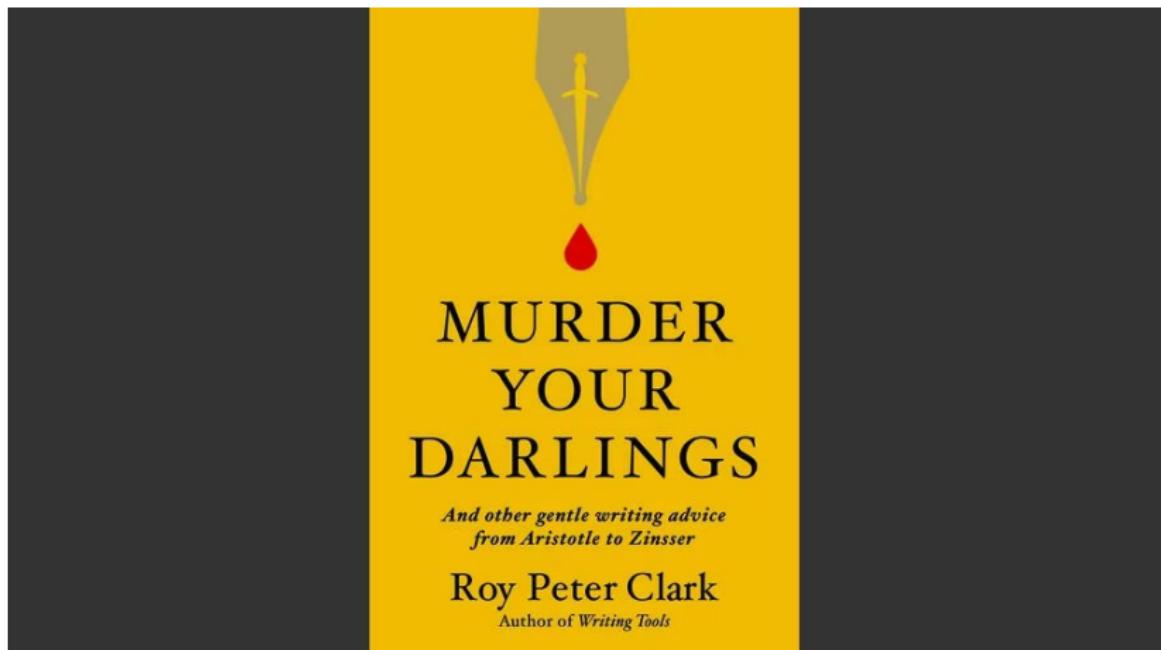
Writing code fast results in lots of dead ends.

There are a few types of “dead-ends” in code. They include:

- ▶ “Interactive” code
- ▶ “Darlings”

Types of dead-ends: Interactive code

Types of dead-ends: Darlings



Introduction to the labs

Introduction to the labs

The rest of the workshop will alternate between lectures and labs.