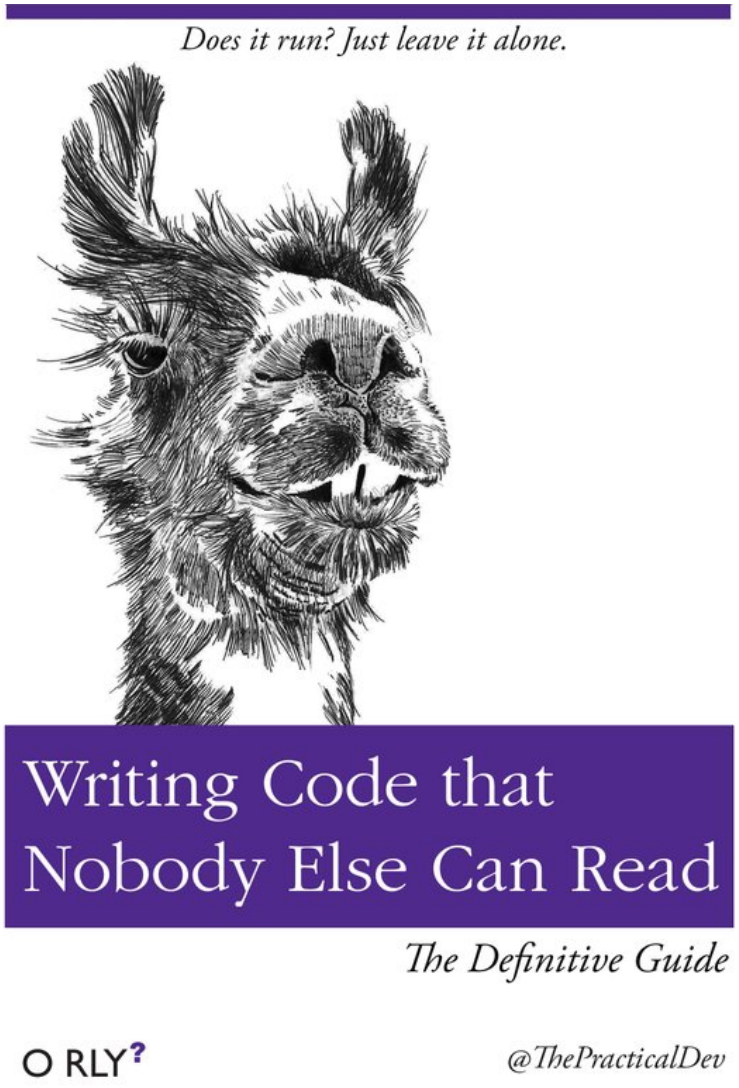
Editing Code for Rigor and Reproducibility

Brooke Anderson

Reproducibility challenge

One key challenge to reproducibility is when the code is hard to figure out.



Separating writing and editing

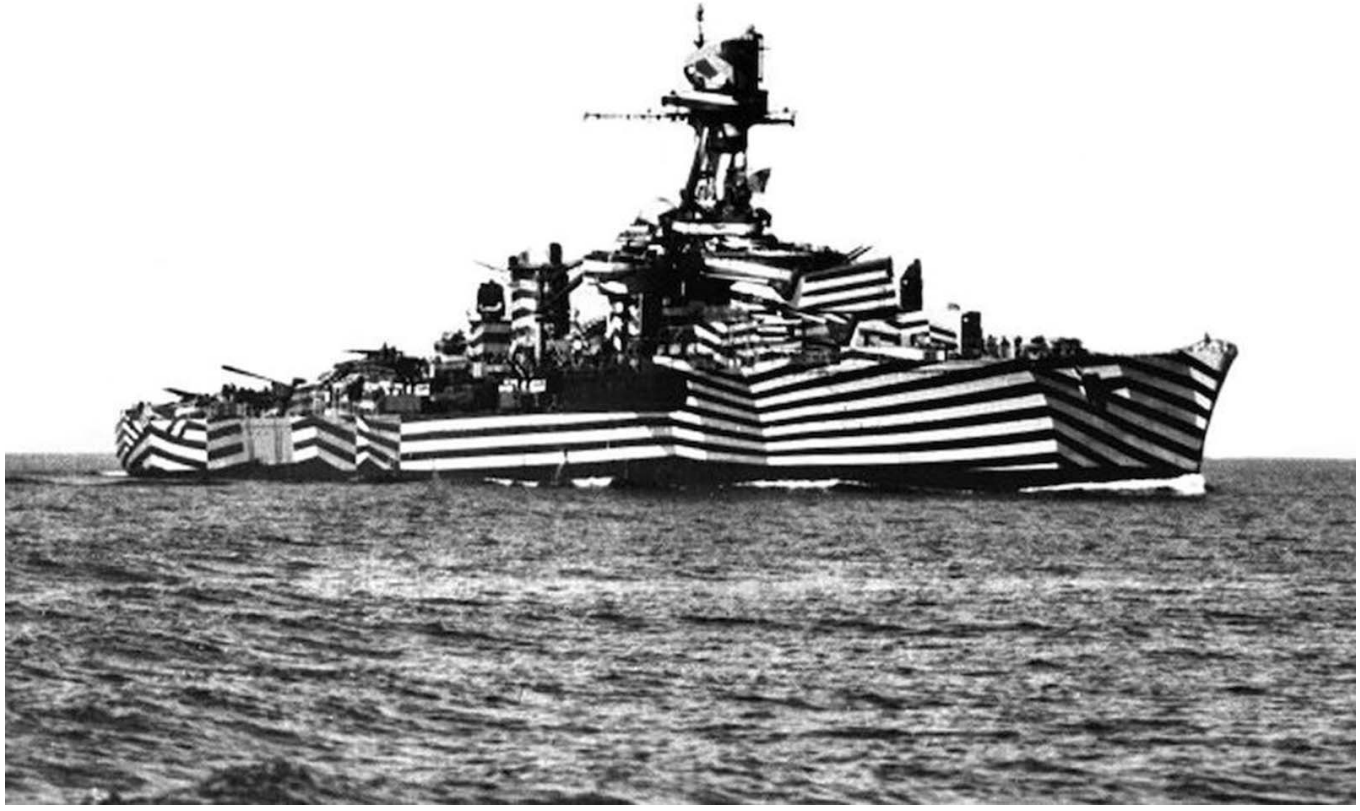
To address this issue, it's critical that you remember that you can edit code in a separate step than when you initially write the code.

Write your code fast. . .

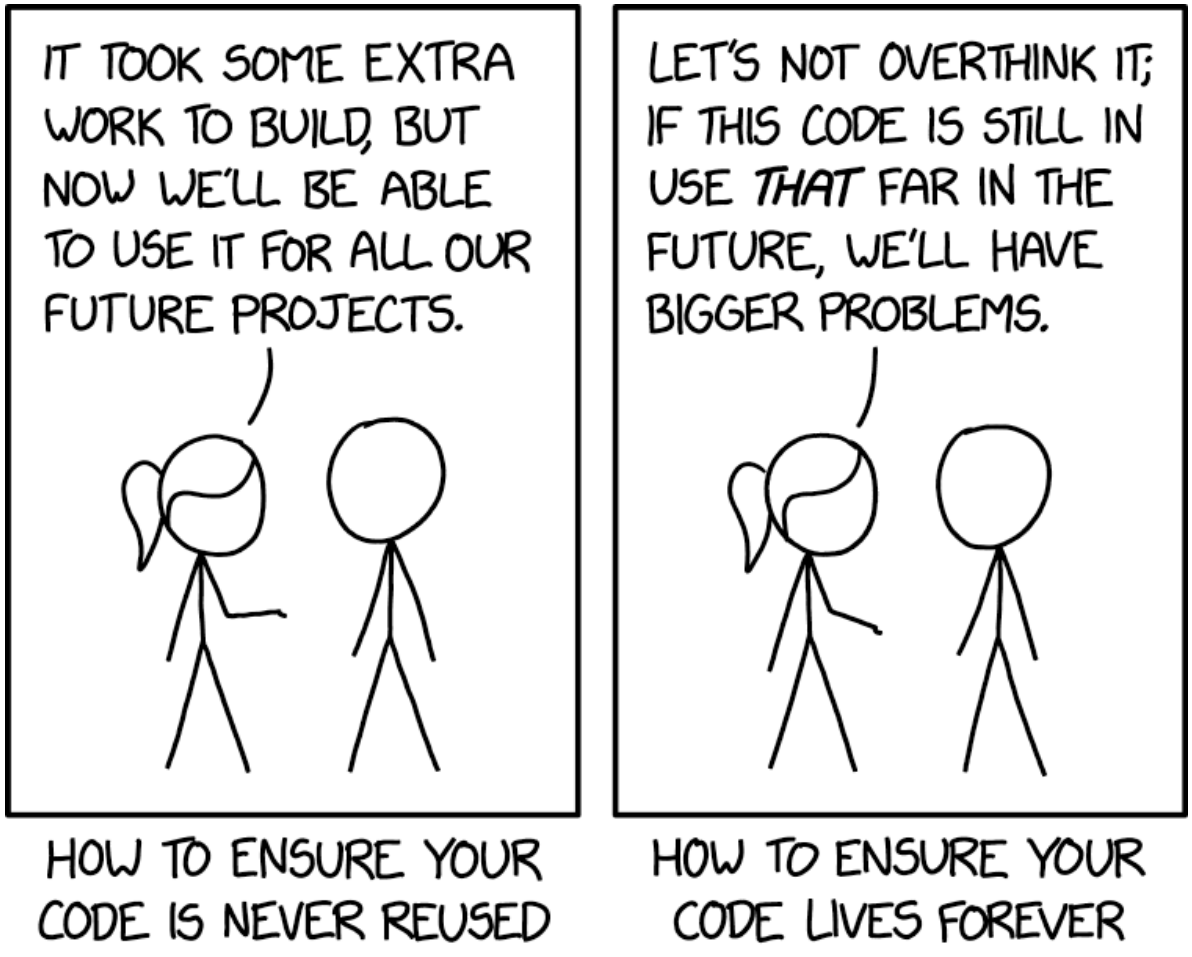
. . . but edit it slowly.

Editing code

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



Editing code



xkcd Explained: https://www.explainxkcd.com/wiki/index.php/File:code_lifespan_2x.png

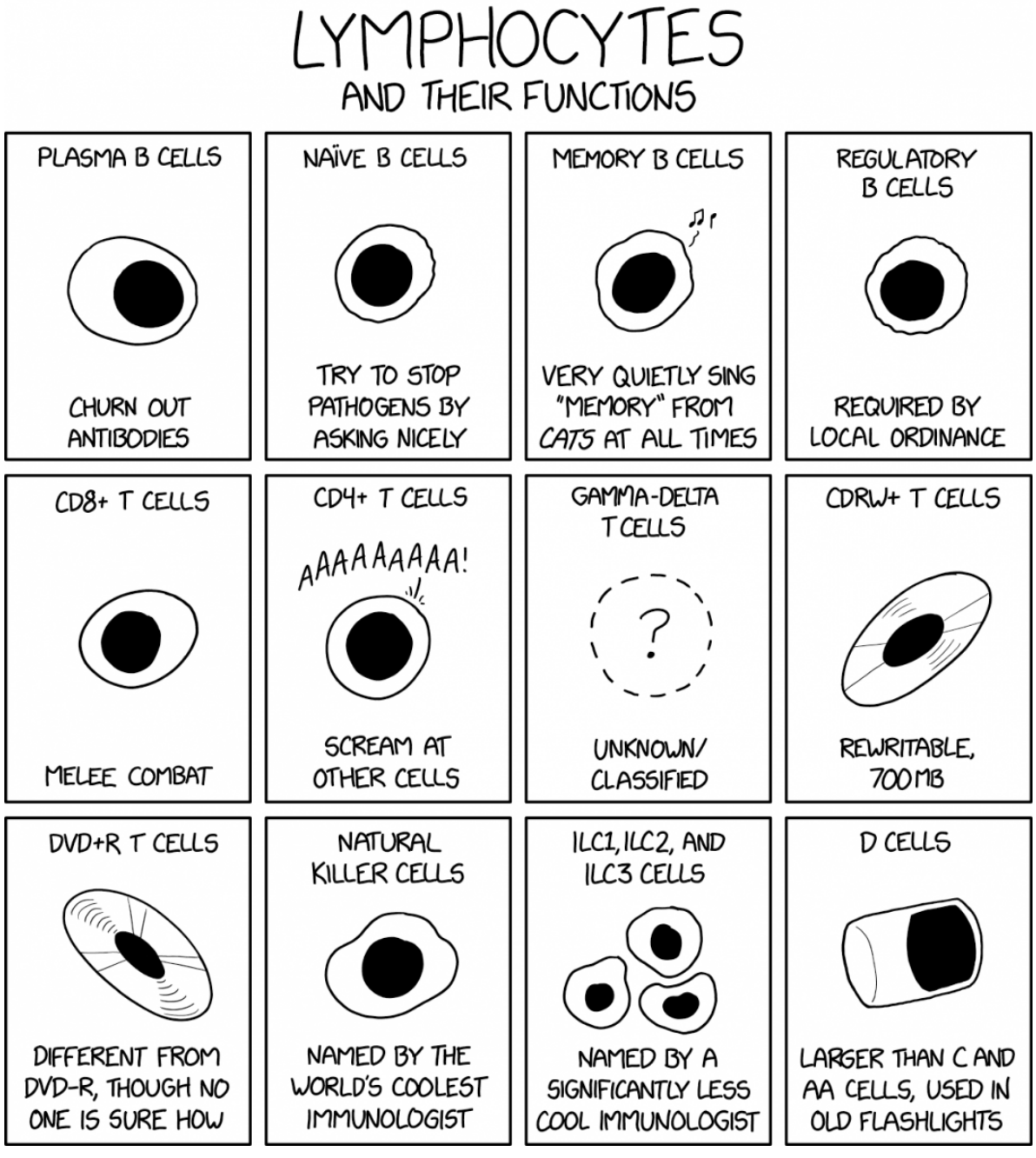
Editing code

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

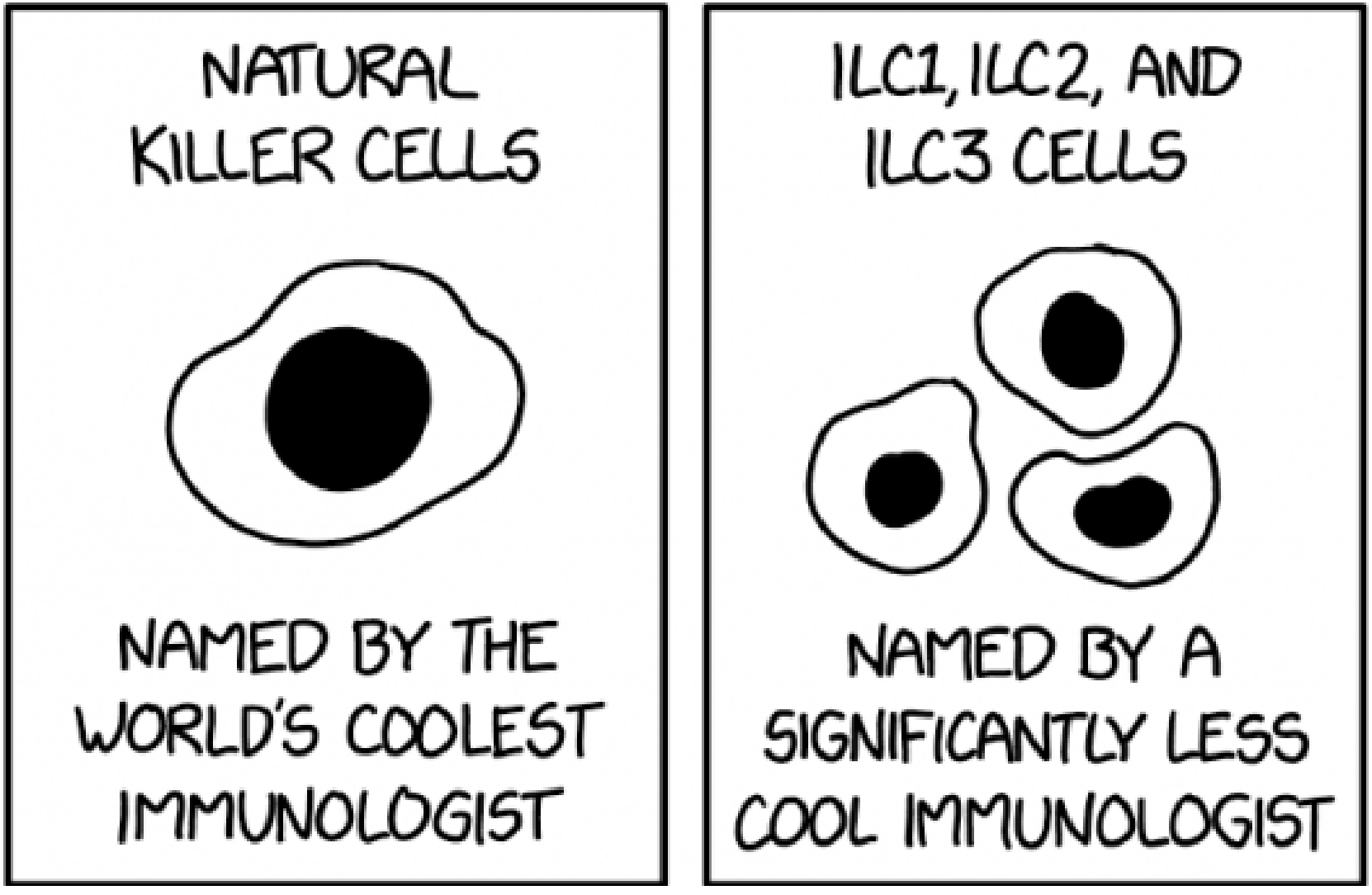
- Improve names
- Break up monolithic code
- Remove dead-end code
- Add useful comments

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



Use good names



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:

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

The tidyverse style guide includes advice for naming things in R: https://style.tidyverse.org/index.html

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`)
- Make names that are 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 dataframe 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:

```
library(janitor)
df <- clean_names(df)
df
```

```
## # 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`:

```
library(dplyr)
df <- df %>%
  rename(longer_name = column_with_long_name)
df
```

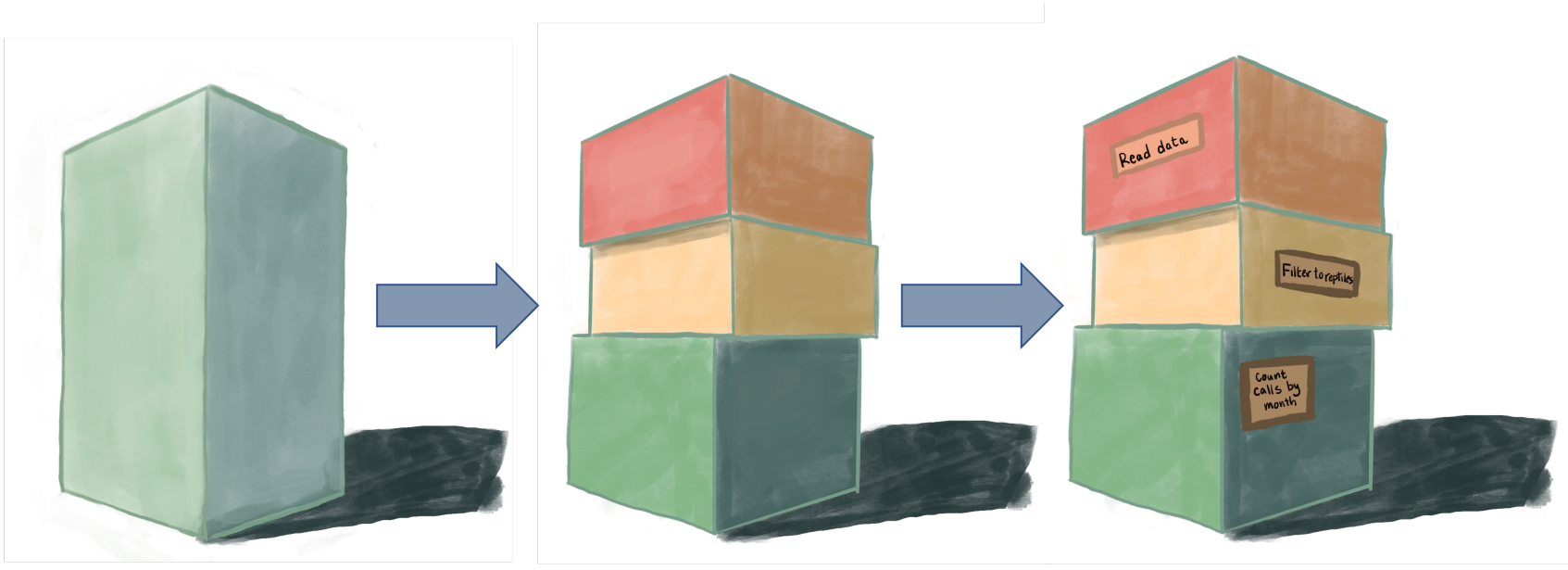
```
## # A tibble: 3 x 2
##   longer_name shorter_name
##         <int> <chr>
## 1           1 a
## 2           2 b
## 3           3 c
```

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.

Break up monolithic code



Code scripts are often "monolithic" when you write them quickly

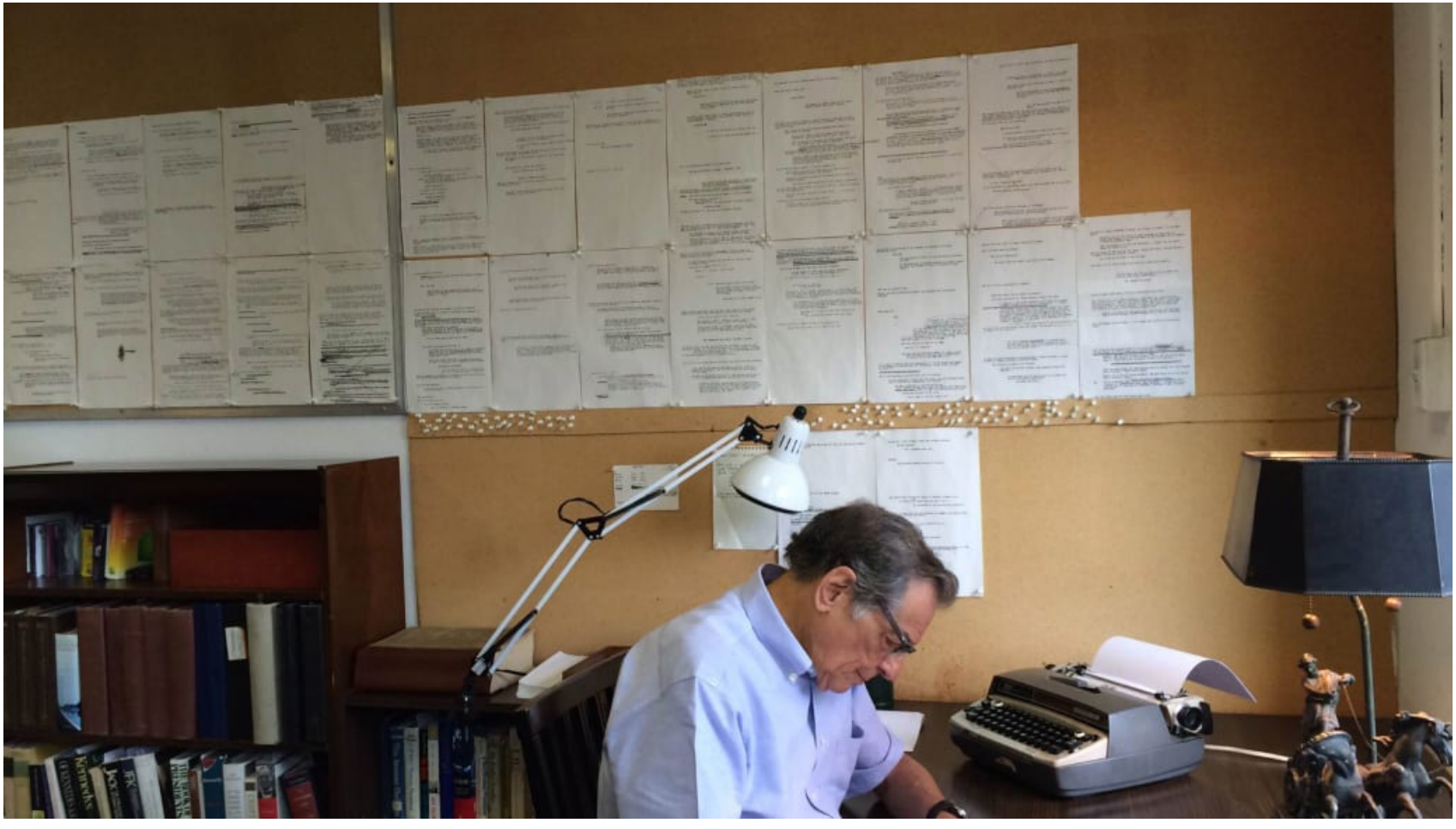
Edit the script to break it up into related subtasks. This provides some structure and order in the script.

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

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

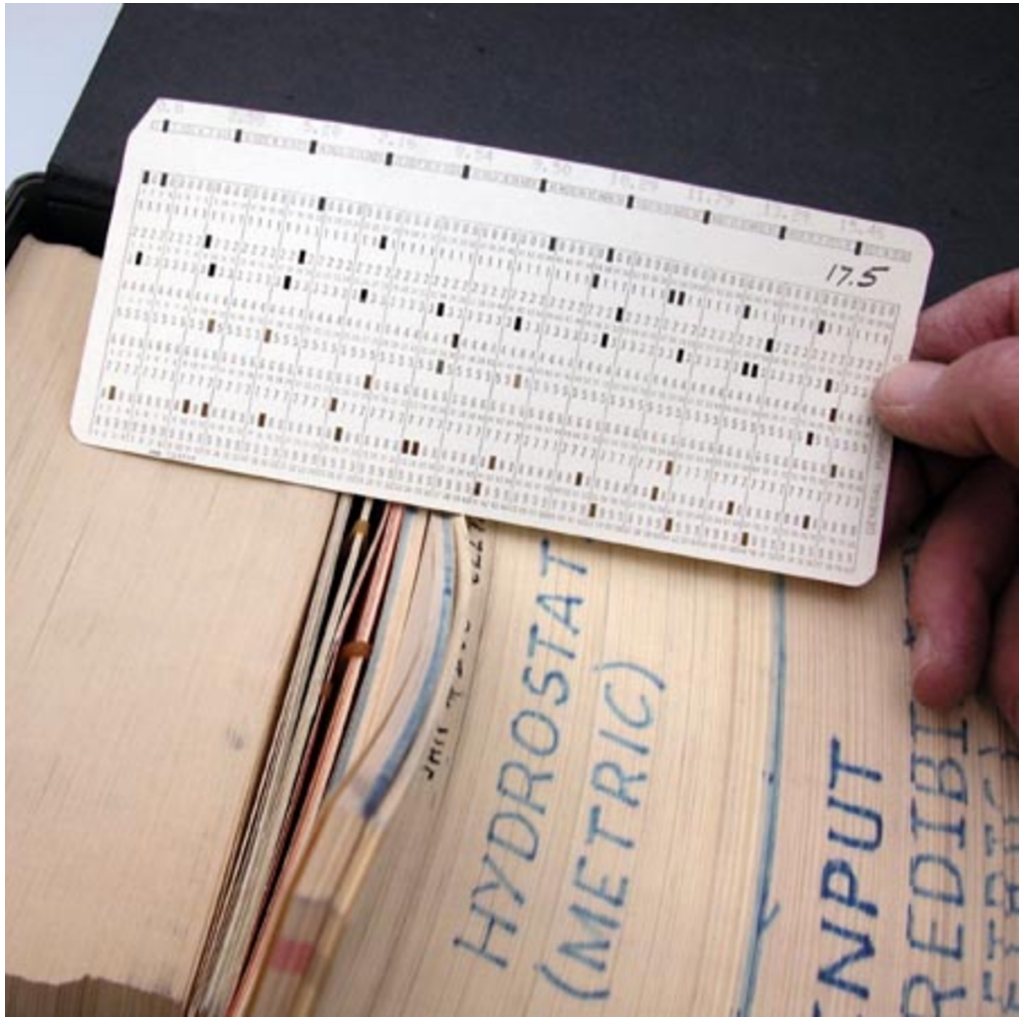


Break up monolithic code

- 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
- Add code comments to document each section

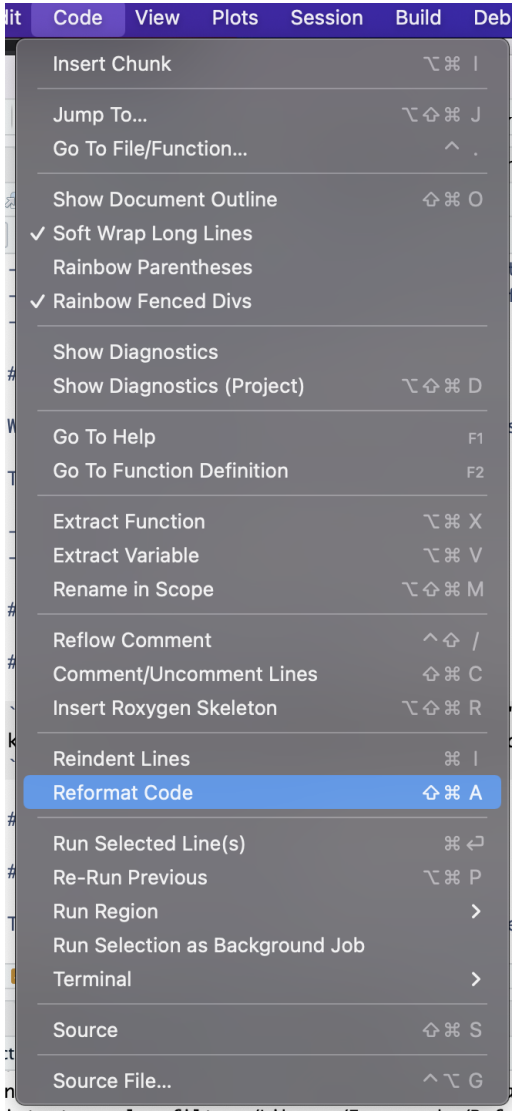
Break up monolithic code

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/

Add useful comments

The next step is to add useful comments describing each section of code. Keep in mind that some comments are more useful than others.

```
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 it's 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)
```

Breaking up monolithic code in RMarkdown

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
```
````

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

As you develop your code, it is common that the script will include function calls that you make to check your progress and explore your data.

Some examples of this type of dead-end code include:

```
my_data
summary(my_data)
View(my_data)
str(my_data)
head(my_data$column_1)
```

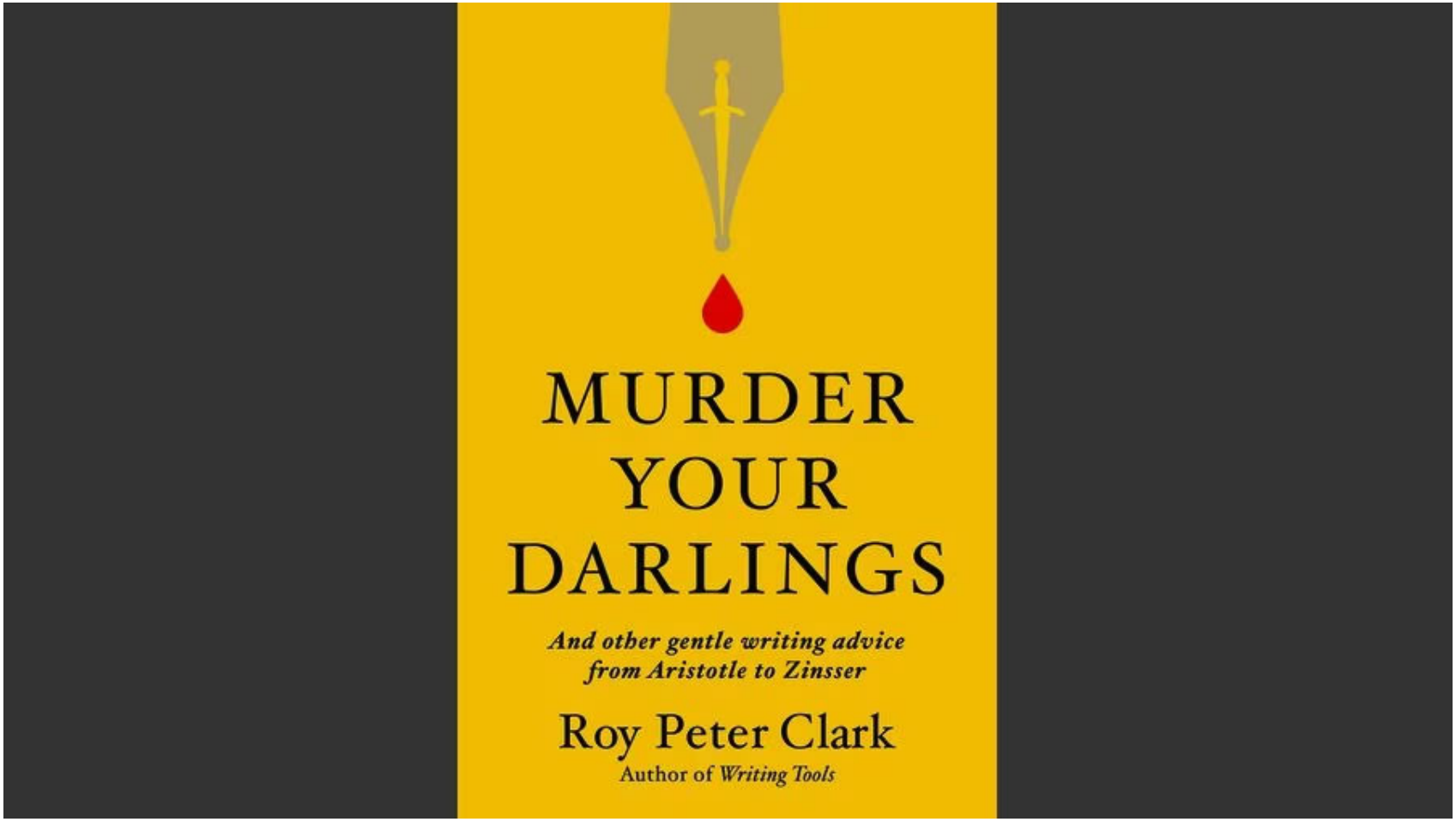
Types of dead-ends: Interactive code

These typically aren't helpful once you've written and edited your code.

You can either plan to edit these out of your script or get in the habit of running these from the console rather than writing them into the script.

Types of dead-ends: Darlings

The name of the other type of dead-end code comes from a famous writing quote.



Types of dead-ends: Darlings

In writing, “darlings” are parts of the text that the writer loves and doesn’t want to remove, but that should be edited out to improve the writing.

One example is a paragraph that doesn’t move the plot forward, but that the author loves because it has a clever turn of phrase that they worked on for a long time.

Types of dead-ends: Darlings

In code, these dead-ends are pieces of code that you pursued but that didn't end up going anywhere. They might involve some very clever code, but they ultimately aren't doing much to make progress on your text.

Often, coders will be reluctant to remove any code that they think might be useful in the future, even if it's not at the moment. However, these clutter up your script, and they send someone trying to understand the code on wild goose chases.