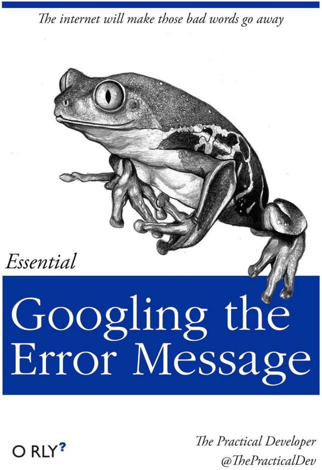
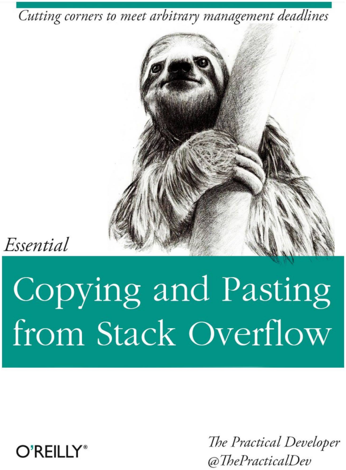
Dissecting and Adapting Example Code

Brooke Anderson and Rachel Severson

Finding example code

Finding example code





The art of Googling for code

- Learn some Google search operators
  - Quotation marks force a match to an exact term or phrase
  - A negative sign excludes a word, returning only matches that don't include that word
- If you get an error message in R, Google it (take out anything specific to your problem, like a filepath or object name)
- Hone your search terms
  - Continue to improve the precision of your coding vocabulary
  - Pay attention to the language advanced coders use
  - Refine your search terms based on what Google returns from your first search

The art of Googling for code

There are a variety of tips that can help you improve your Googling. Some resources include:

- **Advanced Google searching, including search operators**: https://support.google.com/websearch/answer/2466433?hl=en
- **21 Tips and Tricks to Master the Art of Googling as a Developer**: https://www.makeuseof.com/21-tips-and-tricks-to-master-the-art-of-googling-as-a-developer/
- **How to Google effectively as a developer**: https://medium.com/@niamhpower/how-to-google-effectively-as-a-developer-4ebe363afe
- **Googling for code solutions can be tricky—here's how to get started**: https://knightlab.northwestern.edu/2014/03/13/googling-for-code-solutions-can-be-tricky-heres-how-to-get-started/

Where you may find example code

Once you have submitted a good Google search, you should be able to get some promising leads for most R coding tasks.

Some of the types of resources that may have example code include:

- Help forums: StackOverflow, Posit Community
- Package documentation: vignettes, helpfiles (e.g., `?stringr`), cheatsheets
- Blog posts (especially those written with blogdown)
- AI: e.g., ChatGPT (https://openai.com/blog/chatgpt)

Cheatsheets

Work with strings with stringr : : CHEAT SHEET

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches

str_detect(string, **pattern**) Detect the presence of a pattern match in a string. *str_detect(fruit, "a")*

str_which(string, **pattern**) Find the indexes of strings that contain a pattern match. *str_which(fruit, "a")*

str_count(string, **pattern**) Count the number of matches in a string. *str_count(fruit, "a")*

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. *str_locate(fruit, "a")*

Subset Strings

str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. *str_sub(fruit, 1, 3); str_sub(fruit, -2)*

str_subset(string, **pattern**) Return only the strings that contain a pattern match. *str_subset(fruit, "b")*

str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*

str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all**. *str_match(sentences, "(a|the) ([^]+)")*

Manage Lengths

str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). *str_length(fruit)*

str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. *str_trunc(fruit, 3)*

str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. *str_trim(fruit)*

Mutate Strings

str_sub() **<-** value. Replace substrings by identifying the substrings with str_sub() and assigning into the results. *str_sub(fruit, 1, 3) <- "str"*

str_replace(string, **pattern**, replacement) Replace the first matched pattern in each string. *str_replace(fruit, "a", "-")*

str_replace_all(string, **pattern**, replacement) Replace all matched patterns in each string. *str_replace_all(fruit, "a", "-")*

A STRING → a string

str_to_lower(string, locale = "en")[1] Convert strings to lower case. *str_to_lower(sentences)*

a string → A STRING

str_to_upper(string, locale = "en")[1] Convert strings to upper case. *str_to_upper(sentences)*

a string → A String

str_to_title(string, locale = "en")[1] Convert strings to title case. *str_to_title(sentences)*

Join and Split

str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. *str_c(letters, LETTERS)*

str_c(..., sep = "", **collapse = NULL**) Collapse a vector of strings into a single string. *str_c(letters, collapse = "")*

str_dup(string, times) Repeat strings times times. *str_dup(fruit, times = 2)*

str_split_fixed(string, **pattern**, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split** to return a list of substrings. *str_split_fixed(fruit, " ", n=2)*

{xx} {yy}

glue::glue(..., .sep = "", .envir = parent.frame(), .open = "{", .close = "}") Create a string from strings and {expressions} to evaluate. *glue::glue("Pi is {pi}")*

glue::glue_data(.x, ..., .sep = "", .envir = parent.frame(), .open = "{", .close = "}") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. *glue::glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")*

Order Strings

str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Return the vector of indexes that sorts a character vector. *x[str_order(x)]*

str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Sort a character vector. *str_sort(x)*

Helpers

str_conv(string, encoding) Override the encoding of a string. *str_conv(fruit,"ISO-8859-1")*

apple
banana
pear

str_view(string, **pattern**, match = NA) View HTML rendering of first regex match in each string. *str_view(fruit, "[aeiou]")*

apple
banana
pear

str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches. *str_view_all(fruit, "[aeiou]")*

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. *str_wrap(sentences, 20)*

[1] See bit.ly/ISO639-1 for a complete list of locales.

RStudio

Convert string to upper case, lower case, title case, or sentence case

Description

- `str_to_upper()` converts to upper case.
- `str_to_lower()` converts to lower case.
- `str_to_title()` converts to title case, where only the first letter of each word is capitalized.
- `str_to_sentence()` convert to sentence case, where only the first letter of sentence is capitalized.

Usage

```
str_to_upper(string, locale = "en")

str_to_lower(string, locale = "en")

str_to_title(string, locale = "en")

str_to_sentence(string, locale = "en")
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`locale`	Locale to use for comparisons. See `stringi::stri_locale_list()` for all possible options. Defaults to "en" (English) to ensure that default behaviour is consistent across platforms.

Value

A character vector the same length as `string`.

Examples

Run examples

```
dog <- "The quick brown dog"
str_to_upper(dog)
```

Finding R packages

If you are trying to do a specialized task, you may find that you need to find a new R package with the appropriate algorithms. There are steps you can take to identify and evaluate packages for specific tasks.

EDUCATION

Ten simple rules for finding and selecting R packages

Caroline J. Wendt[1,2], **G. Brooke Anderson**[3]*

1 Department of Statistics, Colorado State University, Fort Collins, Colorado, United States of America,
2 Department of Mathematics, Colorado State University, Fort Collins, Colorado, United States of America,
3 Department of Environmental & Radiological Health Sciences, Colorado State University, Fort Collins, Colorado, United States of America

* Brooke.Anderson@colostate.edu

https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1009884

Finding R packages

This paper includes detailed suggestions for how to find and evaluate R packages, so you can pick the best package to tackle your task. Some of the rules that are described include:

- Rule 4: Explore the availability and quality of help
- Rule 5: Quantify how established it is
- Rule 6: Seek evidence of peer acceptance and review
- Rule 8: See how it's developed

Dissecting example code

Getting example code to work

Start by trying the full piece of code.

Does the example run? If not, try these steps first:

- Make sure you have all the packages that are used in the example installed on your computer
- If there is information on the package versions from the example, compare your versions to those used in the example
- Make sure that you have the example data and it's being loaded or set up correctly in R

Dissecting example code

The next step is to go through the example carefully to make sure you understand what it's doing.

- Run through code step-by-step. Take apart pipelines if necessary
- For each step, what does input look like? What does output look like?
- Make sure you understand why each function is being called and why any arguments are being used

Dissecting nested code

In R, you'll often find examples of code (and write them yourself) where a function call is nested within another function call.

For example, in the following example code, a `function1` call is nested inside a `function2` call:

```
function2(function1(my_data, n = 5), verbose = TRUE)
```

Dissecting nested code

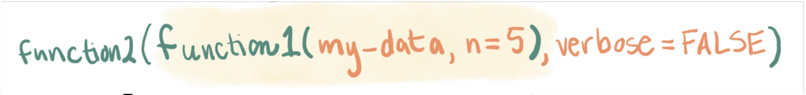
If you are trying to figure out a line of code with nested code, dissect it from the inside out.

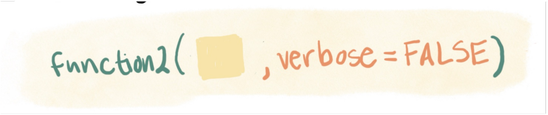
Start by figuring out what the innermost function call is doing. For the moment, ignore everything except the nested function call.

Dissecting nested code

Once you have figured out the innermost function call, you can think of that part of the code line as its output.

Now you can proceed to figure out the outer function call.





Dissecting piped code

Often, an example will include a series of piped functions calls.

```
my_data <- my_data %>%
  rename(better_name = `Bad Name!`) %>%
  mutate(animal_species = fct(animal_species))
```

In this type of piped code, the output of one function call is sent directly as input into the next function call.

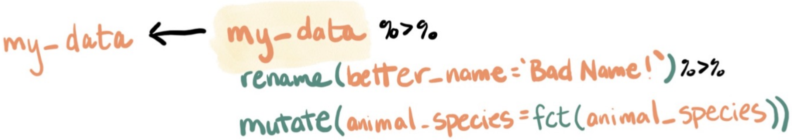
Dissecting piped code

This type of code can be very clear and efficient in performing a series of simple steps. However, it's useful to have some strategies for how to dissect example code that is in a pipeline, as this isn't always as straightforward.

It's a good strategy to dissect the code line by line. Figure out everything up to the first pipe before you look at the next line up to the next pipe, and so on.

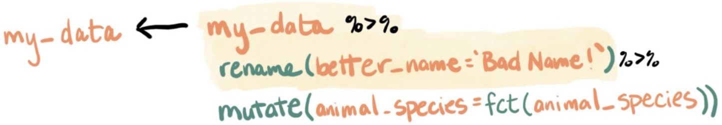
Dissecting piped code

For the example code, you could start just by looking at the data input to the first pipe. In RStudio, you can do this by highlight only that dataframe name (`my_data`) and clicking the Run button (or using the keyboard shortcut for "Run"):



Dissecting piped code

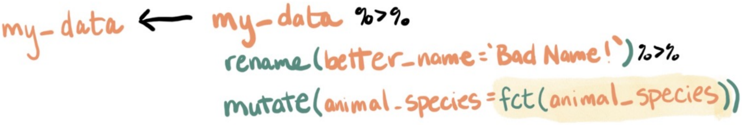
Once you understand that much, you can add on the next function call in the pipe. Highlight the code, starting from the input to the pipe and going up to (but not including) the pipe after the last call you want to assess.



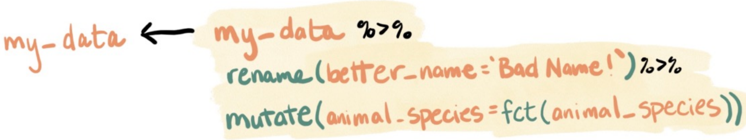
Dissecting piped code

If one of the function calls in the pipeline includes another function call nested inside, use the techniques for nested function calls to dissect that part of the code.

Start with the inner function call:



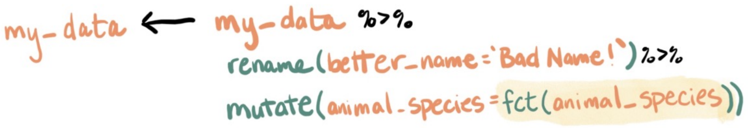
Then dissect the outer function call:



Dissecting piped code

Some nested function calls will operate on a single column of the dataframe, rather than the whole dataframe. In those cases, you can use the `pull` function to investigate the inner function.

For example, to explore this:

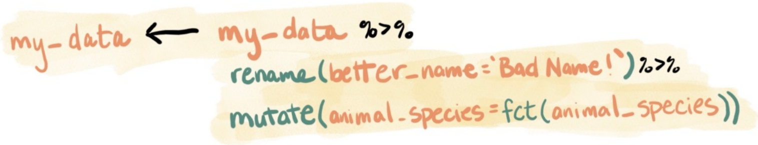


You can use:

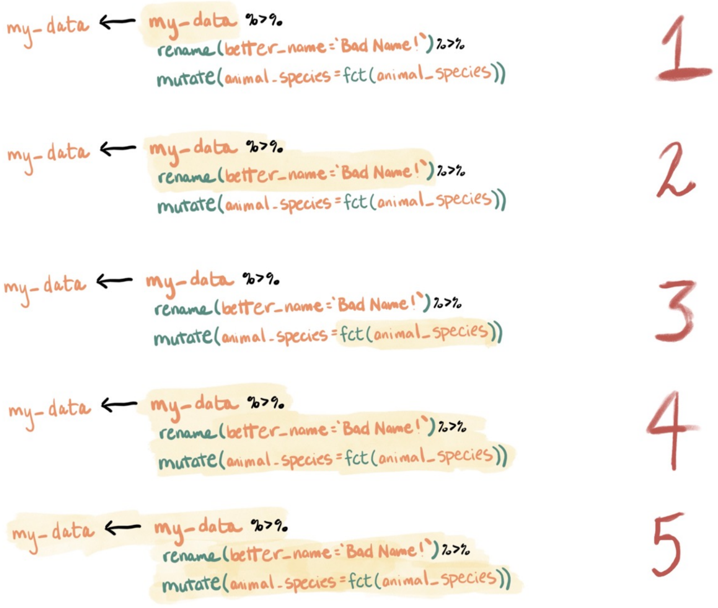
```
my_data %>%
  rename(better_name = `Bad Name!`) %>%
  pull(animal_species) %>%
  fct()
```

Dissecting piped code

If the pipeline overwrites an earlier object, it will usually be best to not run the assignment part of the pipeline until you have dissected all of the rest of the code in the pipeline.



Full process of dissecting piped code



Spot the differences

One method to figure out what a function call does is to see how it changes an object.

Spot the differences

If the function call outputs a revised version of the original object, compare that object before and after the call to make sure you understand how it's changed.

```r
library(stringr)
fruit <- c("apple", "bell pepper", "coconut")

fruit
```

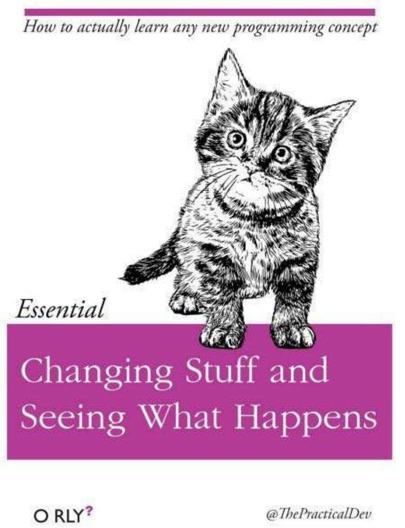
```
## [1] "apple"       "bell pepper" "coconut"
```

```r
str_to_title(fruit)
```

```
## [1] "Apple"       "Bell Pepper" "Coconut"
```

Try changing parameters

Another way to figure out each function call in example code is to try playing around with parameter values in the call, to see how that changes the output.



Adapting example code

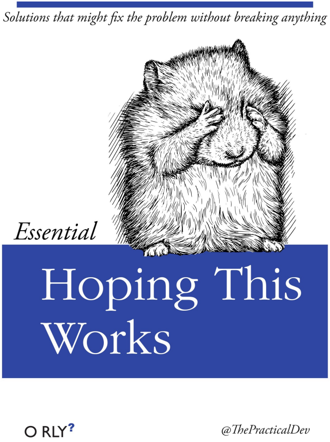
Adapting example code

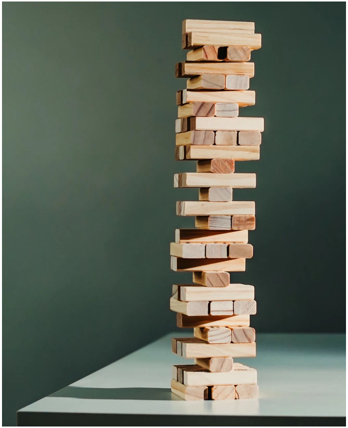
Two steps:

1. Get it to work
2. Edit it to make it more robust and reproducible

Adapting example code

R coders often stop after the first step. This can lead to code that's hard to debug and maintain and that's likely to fail in the future.





Compare data

Compare your data to the example data. How does it compare in terms of:

- Type of data container (vector, dataframe, list)
- Class of data in the object (numeric, character, etc.)
- Object names
- Dimensions of data

Compare data

The `class` function can be helpful in this assessment. Keep in mind that an object can have several levels of this class attribute, moving from a more specific class to a more generic class.

For example, if you have an object named `example_data` that is a tibble (a specialized type of dataframe), when you run `class` you will get:

```r
class(example_data)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Adapt to your tools

We talked in the introductory lecture about how you should have a set of core tools that you know well and use often.

When you pull code from an example, it's useful to edit it to use your set of core tools if it doesn't already.

Adapt to your tools

You gain several advantages by editing code to use your own toolset:

- Bugs are less likely (especially since defaults can be different across R functions that other perform similar actions)
- If there are bugs, you will catch them more quickly
- The code will be easier for you to understand in the future

Adapt to your tools

For example, you might find an example of how to change a column in your data to a factor data class.

The example might be written using base R functions:

```r
df$borough <- as.factor(df$borough)
```

If you tend to use tidyverse tools as your core toolset, you could edit the example to use those tools:

```r
df <- df %>%
  mutate(borough = fct(borough))
```

What is a kluge?

Finally, you should try to identify any kluges in the example data and edit your code to remove them.

From the Jargon File, a kluge is:

> *"A clever programming trick intended to solve a particular nasty case in an expedient, if not clear manner... Often involves ad-hackery..."*

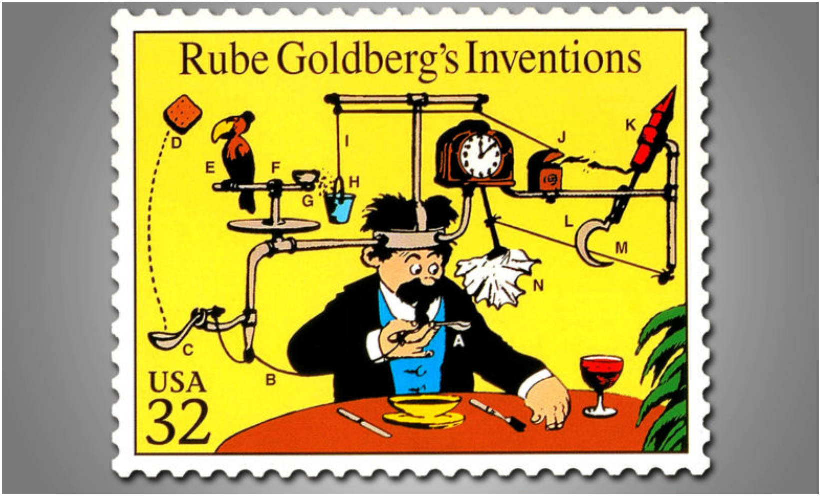
or

> *"Something that works for the wrong reason"*

http://catb.org/jargon/html/K/kluge.html

What is a kluge?

A classic example of a kluge is a Rube Goldberg machine.



What is a kluge?

Kluges can also be clever, but put together in a way that uses materials in unintended ways and that takes a while to understand.

This type of kluge is famous from the TV series MacGyver.

What is a kluge?

Other kluges are clearly going to fall apart at some point, probably in a dramatic and dangerous way.


These types of kluges are often described with, “There I Fixed It”.

Find and fix kluges

> *"The essence of proper kluge building is the designer who is so clever that he outwits himself. —"How to Design a Kluge", Datamation magazine*

You want to edit out kludges because:

- They often use longer code than you need.
- The logic of the code is not clearly linked to the logic of the problem
- They are hard to maintain, understand, and debug
- Some are strongly predisposed to fail unpredictably and dramatically

Don't prioritize concision or efficiency over clarity.