Functional Programming in R

Summer School for Women in Political Methodology

Allison Koh

July 21, 2025

Course Description

This course offers an introduction to functional programming in R, with applications in political science. Through hands-on exercises, participants will learn how to streamline repetitive tasks, implement modular workflows, and document code for replicability[1]. Prior experience with the fundamentals of R and RStudio, as well as data management in R, is recommended.

Setup

Before we get started, let's make sure the following dependencies are installed/loaded for walking through the examples below.

```
library(tidyverse)
library(here) # for path management
library(rlang)
```

Functions 101

Anatomy of a function

`R` functions are made up of three parts:

1. **Formals** (a.k.a. inputs): the parameters or arguments defined by the function
2. **Body:** the code that defines what the function does with its input(s)
3. **Environment:** where the function stores variables and values it uses during execution

[1] We won't get to documentation today, but I have added a link on how to do it in the "Additional resources" section.

You can access the inputs of any function using `formals()` or `args()`:

```
formals(tidytext::get_stopwords)
```

```
$language
[1] "en"

$source
[1] "snowball"
```

```
args(tidytext::get_stopwords)
```

```
function (language = "en", source = "snowball")
NULL
```

You can do the same with the `body` or `environment` of a function.

```
body(tidytext::get_stopwords)
```

```
{
    rlang::check_installed("stopwords", "to use this function.")
    tibble(word = stopwords::stopwords(language = language, source = source),
        lexicon = source)
}
```

```
environment(tidytext::get_stopwords)
```

```
<environment: namespace:tidytext>
```

Discussion Q1 What do we learn about the `stopwords` function from accesing its inputs, body, and environment?

Writing functions

In practice, we consider the following when writing functions:

- **Name:** An intuitive identifier for what the function does
- **Argument(s):** A list of values passed onto the function
- **Body:** The code that runs every time a function is called
- **Return value:** The output that will be sent into the global environment once the function is finished running

You can follow these steps to write functions:

1. Write a script that pertains to a specific task that will form the body of your function.
2. Come up with a good name:
    - Be **explicit** about exactly what the function does.
    - Keep it **concise**.
    - When in doubt, start with a **verb** that is meaningful to what the function does.
3. Add argument(s) to the function's input.
    - Arguments should be based on the parts from the original script that will be removed/abstracted.
4. Make sure you return the output of the function to the global environment.
    - You can save this to the global environment by assigning the output of the function to an object.

Let's write our first function! We'll begin with a relatively simple example: generating synthetic data. This sort of function can be useful for developing research designs and forming pre-analysis plans.

We will first write a function that generates a vector of ages for individuals. First, starting with the script that will turn into the body of the function:

```
age <- rnorm(n = 25, mean = 40, sd = 15) %>%
  round() %>%
  pmax(0) %>% # ensures values are not less than 0
  pmin(100) # ensures that values are not greater than 100
```

Now let's turn this into a function:

1. Let's name it `generate_age_vec`.
2. Following the code in our original script, our arguments will match those for the `rnorm` function:

- `n` for number of individuals in the vector
- `mean` to establish the average age represented
- `sd` for the standard deviation that will inform the distribution of the vector

```
generate_age_vec <- function(n_people, mean_age, sd_age){
  age <- rnorm(n = n_people, mean = mean_age, sd = sd_age) %>%
    round() %>%
  pmax(0) %>%
  pmin(100)
  return(age)
}
```

Note: Saving `age` to an object is not necessary for putting together this function, but assigning parts of the function's body to locally defined objects will be key to writing more complicated functions.

Now let's try out our function! Let's say we want to generate synthetic data for a study targeting university students:

```
generate_age_vec(n_people = 100, mean_age = 20, sd_age = 1)
```

```
  [1] 21 19 21 21 19 18 21 19 20 20 22 22 19 20 20 21 18 21 18 20 21 21 19 20 21
 [26] 20 21 21 21 21 21 20 20 20 19 21 22 19 20 20 20 20 20 19 19 20 20 20 19 20
 [51] 19 19 20 20 19 21 19 20 22 20 21 19 20 20 22 19 20 20 18 20 22 21 20 20 21
 [76] 20 19 19 21 19 21 20 19 22 22 20 21 20 20 20 20 20 21 20 18 19 22 20 20 20
```

Looks good! Let's say we want the function to default to generating cohorts of synthetic information on 100 college-age individuals. We can add *values* to the arguments in our original function to do this.

```
generate_age_vec <- function(n_people = 100, mean_age = 20, sd_age = 1){
  age <- rnorm(n = n_people, mean = mean_age, sd = sd_age) %>%
    round() %>%
  pmax(0) %>%
  pmin(100)
  return(age)
}
```

We can then just call the function without specifying any arguments/values:

```
generate_age_vec()
```

```
  [1] 21 19 21 19 20 19 19 20 20 19 20 20 20 22 19 19 19 19 20 20 21 20 21 20 21
 [26] 21 18 20 20 19 20 21 20 21 21 20 19 20 22 20 20 18 19 19 20 21 20 20 22 19
 [51] 20 19 19 20 22 20 21 20 21 20 20 21 20 21 22 19 20 21 19 21 19 21 19 21 20
 [76] 21 21 20 21 20 19 22 21 19 20 19 21 20 22 21 20 19 19 22 19 20 18 20 20 20
```

And if we want to focus on a different age range/cohort, we can override these default values:

```
generate_age_vec(n_people = 25, mean_age = 35, sd_age = 15)
```

```
 [1] 17 31 35 14 19 18 32 47  6 41 38 37 27 36 57 29 30 42 18 41 36 20 26  6 50
```

Building on this, let's say you want to create a new variable based on this vector that specifies which `generation` an individual belongs to:

- `genX`: 43-58 years old
- `genY`: 27-42 years old
- `genZ`: 11-26 years old

First, using the function for generating synthetic age data above, let's create a vector of numbers to work with.

```
set.seed(77976)
age <- generate_age_vec(n_people = 50, mean_age = 40, sd_age = 10)
age
```

```
 [1] 33 42 45 35 56 19 19 27 24 32 19 35 56 41 42 36 37 30 57 25 43 39 24 45 29
[26] 40 29 55 46 44 40 35 33 51 50 37 49 30 29 36 44 31 36 52 20 11 35 40 37 37
```

How would we do it for this specific age vector?

```
generation <- cut(
  age,
  breaks = c(10.5, 26.5, 42.5, 58.5),
  labels = c("genZ", "genY", "genX")
)
```

Exercise 1 Write a function that codes generations based on a vector of ages, then apply it to our age vector:

```
code_generation <- function(age){
  ADD FUNCTION BODY HERE
}
```

```
code_generation(age)
```

```
 [1] genY genY genX genY genX genZ genZ genY genZ genY genZ genY genX genY genY
[16] genY genY genY genX genZ genX genY genZ genX genY genY genY genX genX genX
[31] genY genY genY genX genX genY genX genY genY genY genX genY genY genX genZ
[46] genZ genY genY genY genY
Levels: genZ genY genX
```

Applying functions to data

For applied examples, we will use the Crowd Counting Consortium's U.S. Protest Event Data (Ulfelder 2025) as our basis for writing data frame functions. This dataset provides event data on contentious politics in the United States. Events of interest comprise of "any type of activity that…is carried out with the explicit purpose of articulating a grievance against a [political] target, or expressing support of a [political] target" and meet the following inclusion criteria:

- Open to the public and free of charge
- Nonviolent: not primarily organized to cause direct harm to any persons
- Occur in the United States, including Puerto Rico, Guam, and the U.S. Virgin Islands

The data comprise of three phases. We will be working with data from Phase 2 of the project, which covers events from 2021 to 2024.

If you were to use this data in your work IRL, writing functions that apply to this dataset would make a lot of sense since there are two other "waves" of event data that are similarly structured.

Let's start by importing the dataset.

```
ccc_data <- read_csv(here("data", "ccc_compiled_20212024.csv"), show_col_types = FALSE)
```

Let's take an initial peek at the data.

```
str(ccc_data)
```

```
spc_tbl_ [139,823 x 72] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ date                : Date[1:139823], format: "2021-01-01" "2021-01-01" ...
 $ locality            : chr [1:139823] "Montgomery" "Tucson" "Lafayette" "Palo Alto" ...
 $ state               : chr [1:139823] "AL" "AZ" "CA" "CA" ...
 $ location_detail     : chr [1:139823] "statewide" "E Speedway Blvd and N Euclid Ave" "El Cu
 $ online              : num [1:139823] 0 0 0 0 0 0 0 0 0 0 ...
 $ type                : chr [1:139823] "strike; boycott" "vigil" "demonstration" "vigil" ..
 $ title               : chr [1:139823] NA NA NA NA ...
 $ macroevent          : chr [1:139823] NA NA NA NA ...
 $ organizations       : chr [1:139823] "Free Alabama Movement" "Women in Black" "Contra Cost
 $ participants        : chr [1:139823] "prisoners" NA NA NA ...
 $ claims              : chr [1:139823] "against prison labor, for safer conditions in Alaba
 $ claims_summary      : chr [1:139823] "against prison labor;for safer conditions in Alabama
 $ claims_verbatim     : chr [1:139823] NA NA NA NA ...
 $ issue_tags_summary  : chr [1:139823] "covid; criminal justice; labor" "foreign affairs" "p
 $ issue_tags_verbatim : chr [1:139823] NA NA NA NA ...
 $ issue_tags          : chr [1:139823] "covid;criminal justice;labor" "foreign affairs" "pre
 $ valence             : num [1:139823] 1 0 2 0 2 1 0 0 0 1 ...
 $ size_text           : chr [1:139823] NA NA NA NA ...
 $ size_low            : num [1:139823] NA NA NA NA 1000 NA NA NA 21 NA ...
 $ size_high           : num [1:139823] NA NA NA NA 1000 NA NA NA 21 NA ...
 $ size_mean           : num [1:139823] NA NA NA NA 1000 NA NA NA 21 NA ...
 $ size_cat            : num [1:139823] 0 0 0 0 3 0 0 0 1 0 ...
 $ arrests             : chr [1:139823] NA NA NA NA ...
 $ arrests_any         : num [1:139823] 0 0 0 0 0 0 0 0 0 0 ...
 $ injuries_crowd      : chr [1:139823] NA NA NA NA ...
 $ injuries_crowd_any  : num [1:139823] 0 0 0 0 0 0 0 0 0 0 ...
 $ injuries_police     : logi [1:139823] NA NA NA NA NA NA ...
 $ injuries_police_any : num [1:139823] 0 0 0 0 0 0 0 0 0 0 ...
 $ property_damage     : chr [1:139823] NA NA NA NA ...
 $ property_damage_any : num [1:139823] 0 0 0 0 0 1 0 0 0 0 ...
 $ chemical_agents     : num [1:139823] 0 0 0 0 0 0 0 0 0 0 ...
 $ participant_measures: chr [1:139823] NA NA NA NA ...
 $ police_measures     : chr [1:139823] NA NA NA NA ...
 $ participant_deaths  : logi [1:139823] NA NA NA NA NA NA ...
 $ police_deaths       : logi [1:139823] NA NA NA NA NA NA ...
 $ source_1            : chr [1:139823] "https://sfbayview.com/2020/12/fam-launches-30-day-e
 $ source_2            : chr [1:139823] "https://twitter.com/ShutDownRacism/status/134619390
 $ source_3            : chr [1:139823] NA NA NA NA ...
 $ source_4            : chr [1:139823] NA NA NA NA ...
 $ source_5            : chr [1:139823] NA NA NA NA ...
 $ source_6            : chr [1:139823] NA NA NA NA ...
 $ source_7            : chr [1:139823] NA NA NA NA ...
```

```
$ source_8            : chr [1:139823] NA NA NA NA ...
$ source_9            : chr [1:139823] NA NA NA NA ...
$ source_10           : chr [1:139823] NA NA NA NA ...
$ source_11           : chr [1:139823] NA NA NA NA ...
$ source_12           : chr [1:139823] NA NA NA NA ...
$ source_13           : chr [1:139823] NA NA NA NA ...
$ source_14           : chr [1:139823] NA NA NA NA ...
$ source_15           : chr [1:139823] NA NA NA NA ...
$ source_16           : chr [1:139823] NA NA NA NA ...
$ source_17           : chr [1:139823] NA NA NA NA ...
$ source_18           : chr [1:139823] NA NA NA NA ...
$ source_19           : chr [1:139823] NA NA NA NA ...
$ source_20           : chr [1:139823] NA NA NA NA ...
$ source_21           : chr [1:139823] NA NA NA NA ...
$ source_22           : chr [1:139823] NA NA NA NA ...
$ source_23           : chr [1:139823] NA NA NA NA ...
$ source_24           : chr [1:139823] NA NA NA NA ...
$ source_25           : chr [1:139823] NA NA NA NA ...
$ source_26           : chr [1:139823] NA NA NA NA ...
$ source_27           : chr [1:139823] NA NA NA NA ...
$ source_28           : logi [1:139823] NA NA NA NA NA NA ...
$ source_29           : logi [1:139823] NA NA NA NA NA NA ...
$ source_30           : logi [1:139823] NA NA NA NA NA NA ...
$ notes               : chr [1:139823] "Scheduled to run 30 days." "every Friday since at le
$ lat                 : num [1:139823] 32.4 32.3 37.9 37.4 34.1 ...
$ lon                 : num [1:139823] -86.3 -111 -122.1 -122.1 -118.1 ...
$ resolved_locality   : chr [1:139823] "Montgomery" "Tucson" "Lafayette" "Palo Alto" ...
$ resolved_county     : chr [1:139823] "Montgomery County" "Pima County" "Contra Costa Count
$ resolved_state      : chr [1:139823] "AL" "AZ" "CA" "CA" ...
$ fips_code           : chr [1:139823] "01101" "04019" "06013" "06085" ...
- attr(*, "spec")=
 .. cols(
 ..   date = col_date(format = ""),
 ..   locality = col_character(),
 ..   state = col_character(),
 ..   location_detail = col_character(),
 ..   online = col_double(),
 ..   type = col_character(),
 ..   title = col_character(),
 ..   macroevent = col_character(),
 ..   organizations = col_character(),
 ..   participants = col_character(),
 ..   claims = col_character(),
```

```
..   claims_summary = col_character(),
..   claims_verbatim = col_character(),
..   issue_tags_summary = col_character(),
..   issue_tags_verbatim = col_character(),
..   issue_tags = col_character(),
..   valence = col_double(),
..   size_text = col_character(),
..   size_low = col_double(),
..   size_high = col_double(),
..   size_mean = col_double(),
..   size_cat = col_double(),
..   arrests = col_character(),
..   arrests_any = col_double(),
..   injuries_crowd = col_character(),
..   injuries_crowd_any = col_double(),
..   injuries_police = col_logical(),
..   injuries_police_any = col_double(),
..   property_damage = col_character(),
..   property_damage_any = col_double(),
..   chemical_agents = col_double(),
..   participant_measures = col_character(),
..   police_measures = col_character(),
..   participant_deaths = col_logical(),
..   police_deaths = col_logical(),
..   source_1 = col_character(),
..   source_2 = col_character(),
..   source_3 = col_character(),
..   source_4 = col_character(),
..   source_5 = col_character(),
..   source_6 = col_character(),
..   source_7 = col_character(),
..   source_8 = col_character(),
..   source_9 = col_character(),
..   source_10 = col_character(),
..   source_11 = col_character(),
..   source_12 = col_character(),
..   source_13 = col_character(),
..   source_14 = col_character(),
..   source_15 = col_character(),
..   source_16 = col_character(),
..   source_17 = col_character(),
..   source_18 = col_character(),
..   source_19 = col_character(),
```

```
  ..   source_20 = col_character(),
  ..   source_21 = col_character(),
  ..   source_22 = col_character(),
  ..   source_23 = col_character(),
  ..   source_24 = col_character(),
  ..   source_25 = col_character(),
  ..   source_26 = col_character(),
  ..   source_27 = col_character(),
  ..   source_28 = col_logical(),
  ..   source_29 = col_logical(),
  ..   source_30 = col_logical(),
  ..   notes = col_character(),
  ..   lat = col_double(),
  ..   lon = col_double(),
  ..   resolved_locality = col_character(),
  ..   resolved_county = col_character(),
  ..   resolved_state = col_character(),
  ..   fips_code = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

```
head(ccc_data)
```

```
# A tibble: 6 x 72
  date       locality      state location_detail   online type  title macroevent
  <date>     <chr>         <chr> <chr>              <dbl> <chr> <chr> <chr>
1 2021-01-01 Montgomery    AL    statewide              0 stri~ <NA>  <NA>
2 2021-01-01 Tucson        AZ    E Speedway Blvd ~      0 vigil <NA>  <NA>
3 2021-01-01 Lafayette     CA    El Curtola Blvd ~      0 demo~ <NA>  <NA>
4 2021-01-01 Palo Alto     CA    El Camino Real a~      0 vigil <NA>  <NA>
5 2021-01-01 Pasadena      CA    Rose Bowl Stadium      0 rall~ Patr~ <NA>
6 2021-01-01 San Francisco CA    home of U.S. Hou~      0 dire~ <NA>  <NA>
# i 64 more variables: organizations <chr>, participants <chr>, claims <chr>,
#   claims_summary <chr>, claims_verbatim <chr>, issue_tags_summary <chr>,
#   issue_tags_verbatim <chr>, issue_tags <chr>, valence <dbl>,
#   size_text <chr>, size_low <dbl>, size_high <dbl>, size_mean <dbl>,
#   size_cat <dbl>, arrests <chr>, arrests_any <dbl>, injuries_crowd <chr>,
#   injuries_crowd_any <dbl>, injuries_police <lgl>, injuries_police_any <dbl>,
#   property_damage <chr>, property_damage_any <dbl>, ...
```

Inspecting data

There's still a *lot* of noise here. Let's write some functions that can help us **inspect** the data in a way that will be more informative for analyzing event data on contentious politics.

We can see from an initial inspection of the data that there are a *lot* of missing values. We are likely to repeatedly address this issue across other phases of this data project, and potentially across sub-analyses of the same dataset. Let's write a function to make this less redundant.

First, let's write a script that **inspects** the issue of missing values.

```
na_counts <- ccc_data %>%
  summarise_all(~ sum(is.na(.))) %>%
  pivot_longer(cols = everything(), names_to = "var", values_to = "n_missing") %>%
  filter(n_missing > 0) %>%
  arrange(desc(n_missing))

na_counts
```

```
# A tibble: 65 x 2
   var                n_missing
   <chr>                  <int>
 1 source_28             139823
 2 source_29             139823
 3 source_30             139823
 4 police_deaths         139822
 5 participant_deaths    139809
 6 injuries_police       139803
 7 source_27             139786
 8 source_26             139780
 9 source_25             139775
10 source_24             139762
# i 55 more rows
```

Great, let's turn the above script into a function.

```
count_missing <- function(data){
  na_counts <- data %>%
    summarise_all(~ sum(is.na(.))) %>%
    pivot_longer(cols = everything(), names_to = "var", values_to = "n_missing") %>%
    filter(n_missing > 0) %>%
    arrange(desc(n_missing))
```

```
  return(na_counts)
}
```

```
count_missing(ccc_data)
```

```
# A tibble: 65 x 2
   var                n_missing
   <chr>                  <int>
 1 source_28             139823
 2 source_29             139823
 3 source_30             139823
 4 police_deaths         139822
 5 participant_deaths    139809
 6 injuries_police       139803
 7 source_27             139786
 8 source_26             139780
 9 source_25             139775
10 source_24             139762
# i 55 more rows
```

Manipulating data

Now we know what we're working with. Let's write a function that allows us to **manipulate** the data for our purposes:

- We are not interested in columns that start with "`source_`".
- We are not interested in keeping any variables with a high number of missings (>500).
- Among the selected columns with <500 missing values, we are not interested in keeping any rows with missing values.

```
na_counts_subset <- na_counts %>%
  filter(
    !grepl("^source_", var),
    n_missing < 500
  )

na_counts_subset
```

```
# A tibble: 10 x 2
   var            n_missing
```

```
   <chr>              <int>
 1 fips_code            126
 2 resolved_state        93
 3 online                68
 4 lat                   61
 5 lon                   61
 6 locality              53
 7 type                  49
 8 valence               34
 9 state                 16
10 claims                 9
```

Now we have a list of variables with limited missing values. Let's `select` for variables with limited missing values and then `filter` out any remaining rows with missing values.

This is what the script for that would look like, building off the code we've already written:

```
# Vectors that specify which variables to keep/drop
missings_vars_all <- na_counts$var
missings_vars_keep <- na_counts_subset$var
missings_vars_drop <- setdiff(missings_vars_all, missings_vars_keep)

missings_vars_drop
```

```
 [1] "source_28"           "source_29"           "source_30"
 [4] "police_deaths"       "participant_deaths"  "injuries_police"
 [7] "source_27"           "source_26"           "source_25"
[10] "source_24"           "source_23"           "source_22"
[13] "source_21"           "source_20"           "source_19"
[16] "source_18"           "source_17"           "source_16"
[19] "source_15"           "source_14"           "injuries_crowd"
[22] "source_13"           "source_12"           "property_damage"
[25] "source_11"           "source_10"           "source_9"
[28] "arrests"             "source_8"            "source_7"
[31] "source_6"            "macroevent"          "source_5"
[34] "police_measures"     "source_4"            "source_3"
[37] "issue_tags_verbatim" "participants"        "participant_measures"
[40] "claims_verbatim"     "size_text"           "size_high"
[43] "size_low"            "size_mean"           "title"
[46] "source_2"            "notes"               "organizations"
[49] "resolved_county"     "location_detail"     "issue_tags_summary"
[52] "issue_tags"          "resolved_locality"   "claims_summary"
[55] "source_1"
```

```r
# Use `select()` to drop variables with too many missings
ccc_clean <- ccc_data %>%
  select(-one_of(missings_vars_drop))

ncol(ccc_data)
```

```
[1] 72
```

```r
ncol(ccc_clean)
```

```
[1] 17
```

```r
# And now filtering out all remaining rows with missing values
ccc_clean <- ccc_clean %>%
  drop_na()

nrow(ccc_data)
```

```
[1] 139823
```

```r
nrow(ccc_clean)
```

```
[1] 139580
```

Exercise 2a Write a function that handles missing values. It should:

- `select` for variables with limited missing values and
- `filter` out any remaining rows with missing values.

Hint: we can integrate the other function we just wrote into this function.

```r
handle_missing_values <- function(data){
  na_counts <- count_missing(data)

  #' Select for variables that are not of interest
  #'  - too many missings
  #'  - contains the link to a `source`
  na_counts_subset <- na_counts %>%
    filter(
```

```
      !grepl("^source_", var),
      n_missing < 500
    )

  missings_vars_all <- na_counts$var
  missings_vars_keep <- na_counts_subset$var
  missings_vars_drop <- setdiff(missings_vars_all, missings_vars_keep)

  missings_vars_drop

  # Use `select()` to drop variables with too many missings
  data_subset <- data %>%
    select(-one_of(missings_vars_drop))

  # And now filtering out all remaining rows with missing values
  data_subset <- data_subset %>%
    drop_na()

  return(data_subset)
}
```

Exercise 2b: Apply this function to create a dataset called `ccc_clean`.

```
ccc_clean <- handle_missing_values(ccc_data)

nrow(ccc_clean)
```

```
[1] 139580
```

```
ncol(ccc_clean)
```

```
[1] 17
```

Great, we have a complete dataset to work with! The 17 remaining variables are as follows:

- `date`: Date of event in YYYY-MM-DD format; multi-day events are recorded as one-day-per-row
- `locality`:Name of the locality (i.e., city or town) in which the event took place
- `state`: Two-letter U.S. postal abbreviation for the state or U.S. territory in which the event took place

- `online`: Binary indicator for online-only events. 1 = online, 0 = in-person. Generated from location and event type information in the source data
- `type`: Type(s) of protest action (e.g. march, protest, demonstration, strike, counter-protest, sit-in), separated with semicolons or commas when more than one
- `claims`: Comma-separated text phrases describing what the event was about
- `valence`: Political valence of the event (2=right wing, 1=left wing, 0=other or neither)
- `size_cat`: categorical indicator of crowd size
  - 0 = unknown
  - 1 = 1-99
  - 2 = 100-999
  - 3 = 1,000 - 9,999
  - 4 = 10,000+
- `arrests_any`: Binary indicator for whether or not any arrests occurred. 1 = yes, 0 = no.
- `injuries_crowd_any`: Binary indicator for whether or not any protesters were reportedly injured. 1 = yes, 0 = no.
- `injuries_police_any`: Binary indicator for whether or not any police officers were reportedly injured. 1 = yes, 0 = no.
- `property_damage_any`: Binary indicator for whether or not protesters reportedly caused any property damage. 1 = yes, 0 = no.
- `chemical_agents`: Binary indicator for whether police or other state security forces used tear gas or other chemical irritants, such as pepper spray or pepper balls, during the protest event. 1 = yes, 0 = no.
- `lat` and `lon`: latitude and longitude of the locality in which the event took place as resolved by the Google Maps Geocoding API
- `resolved_state`: Postal abbreviation of the state or territory in which the event occurred, as resolved by the Google Maps Geocoding API.
- `fips_code`: Five-digit FIPS code for the county in which the event took place

Now let's say we're interested in the *type* of events represented in the dataset. Let's have a look at the first few values in the `type` variable:

```
ccc_clean$type[1:5]
```

```
[1] "strike; boycott" "vigil"           "demonstration"   "vigil"
[5] "rally; parade"
```

As shown here, some events have more than one type indicated. Let's write up some code that expands the data so each row represents an event-type pairing:

```
ccc_types <- ccc_clean %>%
  separate_rows(type, sep = ";") %>%
  mutate(type == gsub("\\s+", "", type))

nrow(ccc_clean)
```

```
[1] 139580
```

```
nrow(ccc_types)
```

```
[1] 165730
```

We'll take the increased number of rows in `ccc_types` as an indication that the data is doing what we want it to do.

Now let's turn it into a function that separates rows by type:

```
separate_rows_by_type <- function(data){
  data_expanded <- data %>%
    separate_rows(type, sep = ";") %>%
  mutate(type == gsub("\\s+", "", type))
  return(data_expanded)
}
```

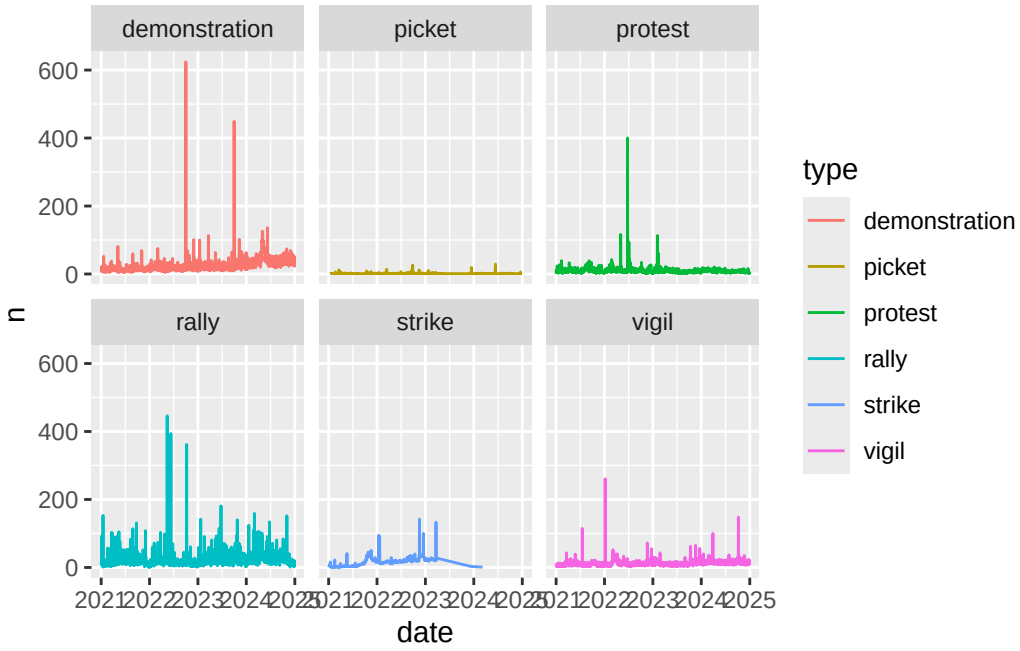
Even if we do not end up using this function for other datasets, it could be useful as a *helper function* so we can use some shorthand in other functions that work with this expanded version of the dataset.

Visualizing data

Functions can also be helpful for streamlining data visualization. For example, functions that visualize user-specified subsets of data can be especially useful for creating dashboards.

Let's say we're interested in generating plots that illustrate event types over time for a given date range. Let's write a script for illustrating six event types over time: `demonstration`, `rally`, `vigil`, `protest`, `strike`, and `picket`.

```
event_types <- c("demonstration", "rally", "vigil", "protest", "strike", "picket")

ccc_types_over_time <- separate_rows_by_type(ccc_clean) %>%
  filter(type %in% event_types) %>%
  count(date, type)

ccc_types_over_time %>%
  ggplot(aes(x = date, y = n, color = type)) +
  geom_line() +
  facet_wrap(~type)
```

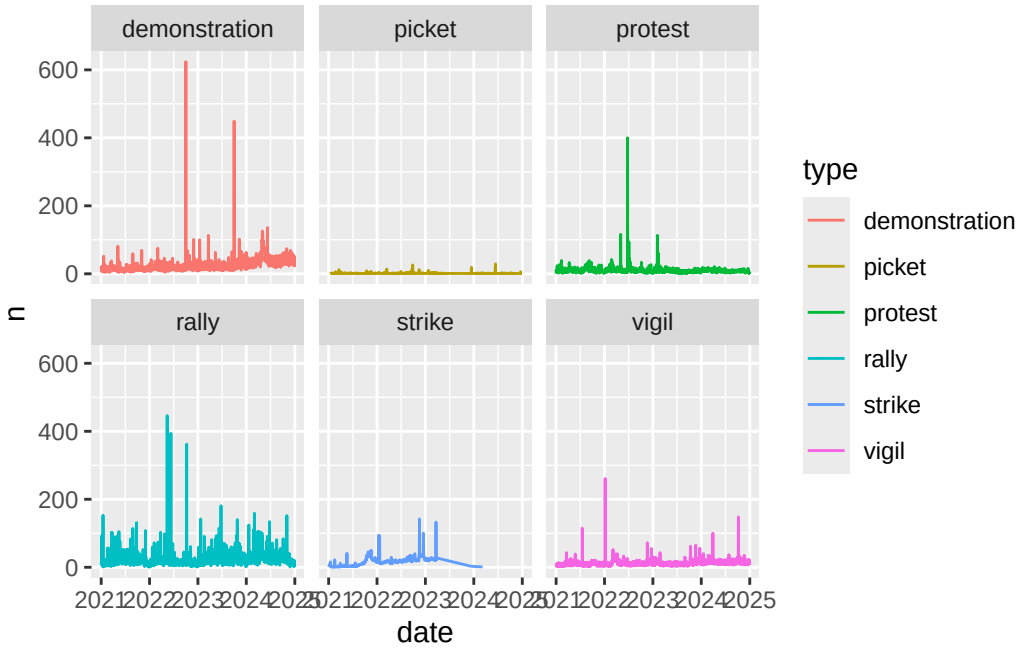


Now let's write a function that takes a vector of events and a data frame as input, then outputs this data visualization on types of events.

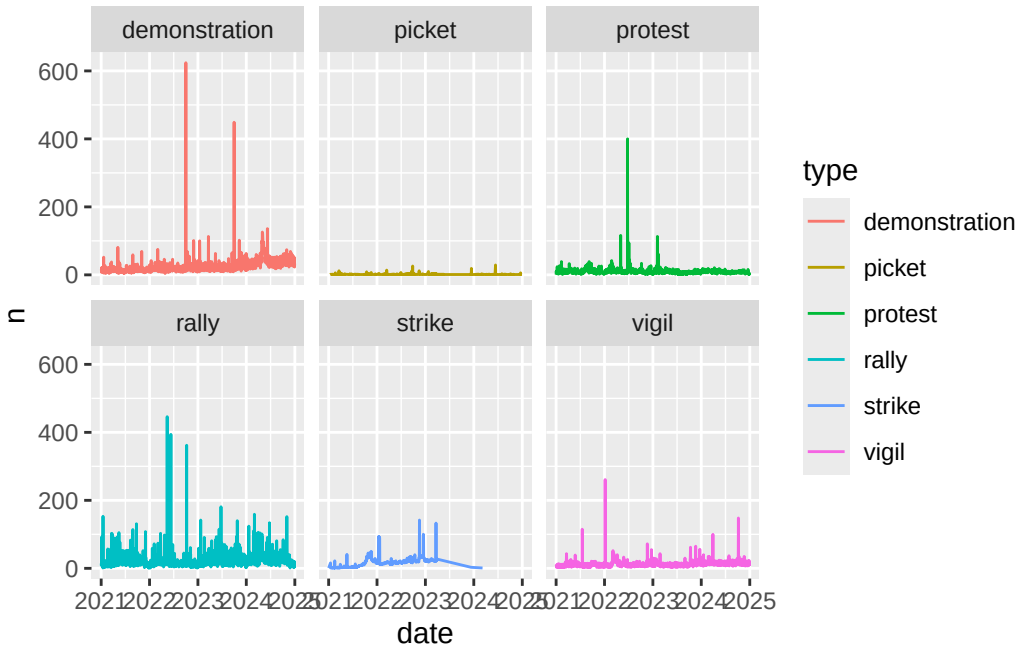
Here's what the body of the function will be based on:

```
event_types <- c("demonstration", "rally", "vigil", "protest", "strike", "picket")

ccc_types_over_time <- separate_rows_by_type(ccc_clean) %>%
  filter(type %in% event_types) %>%
  count(date, type) %>%
  arrange(desc(n))

ccc_types_over_time %>%
```

```
  ggplot(aes(x = date, y = n, color = type)) +
  geom_line() +
  facet_wrap(~type)
```

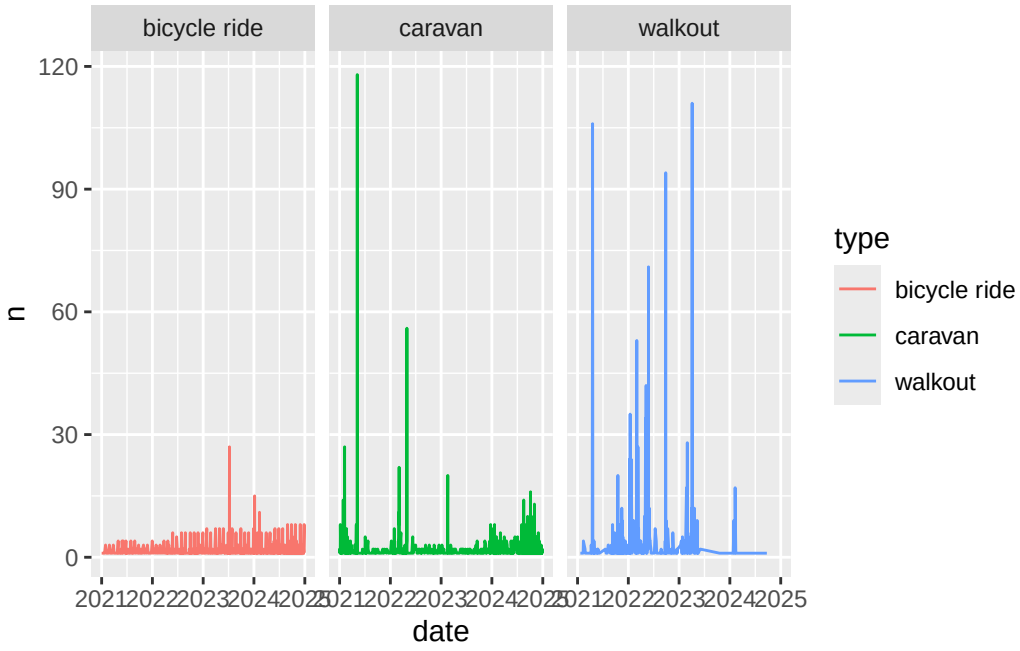



```
visualize_types_over_time <- function(df, event_types = c("demonstration", "rally", "vigil",

  df_types_over_time <- separate_rows_by_type(df) %>%
    filter(type %in% event_types) %>%
    count(date, type)

  df_types_over_time %>%
    ggplot(aes(x = date, y = n, color = type)) +
    geom_line() +
    facet_wrap(~type)
}

visualize_types_over_time(ccc_clean)
```



Since we include a vector of event types as input for the function, we can use this function to visualize other event types:

```
visualize_types_over_time(ccc_clean, event_types = c("caravan", "bicycle ride", "walkout"))
```



Discussion Q2 What other arguments could we add to this function?

Iteration

`purrr::map()`

Now that we understand how functions work, we can start applying them repeatedly across multiple elements. This process is known as *iteration*.

In `R`, iteration refers to the application of the same operation to every item in a vector, list, or column. This includes:

- Applying a function to every row/column of a dataset
- Calculating summary statistics for every country in a panel dataset
- Cleaning a batch of interview transcripts

The classic way of iterating involves for loops. Let's say we want to set every word in the following vector to lowercase:

```
regime_types <- c("Democratic", "Authoritarian", "Hybrid")

for (word in regime_types) {
  print(tolower(word))
}
```

```
[1] "democratic"
[1] "authoritarian"
[1] "hybrid"
```

While this *technically* would work, it can get verbose when applying code across lists of models, multiple text documents, grouped datasets, and the like. At this scale, for loops can be really hard to read, debug, or reuse. Which is the opposite of what we want to do!

The `purrr` package offers a *tidier* toolkit for iteration. Instead of writing out the for loops, `purrr`'s functions will do the iteration for you.

The most common function used from the `purrr` package is `map()`.

Let's revisit our text cleaning example from before:

```
map(regime_types, tolower)
```

```
[[1]]
[1] "democratic"

[[2]]
```

```
[1] "authoritarian"

[[3]]
[1] "hybrid"
```

So much easier! It more or less does the same thing, but returns a list.

If you wanted a character vector instead of a list, you can use the `map_chr()` function:

```
map_chr(regime_types, tolower)
```

```
[1] "democratic"    "authoritarian" "hybrid"
```

`purrr` also has other variants of `map()` that produce other types of vectors:

- `map_int()`: integer vector
- `map_dbl()`: double vector
- `map_chr()`: character vector
- `map_lgl()`: logical vector

Let's try some more examples with the CCC data.

Let's start with a tibble containing information on whether arrests, injuries to protesters, or use of chemical agents took place:

```
incident_df <- ccc_clean %>%
  select(
    arrests_any,
    injuries_crowd_any,
    chemical_agents
  )
```

```
str(incident_df)
```

```
tibble [139,580 x 3] (S3: tbl_df/tbl/data.frame)
 $ arrests_any       : num [1:139580] 0 0 0 0 0 0 0 0 0 0 ...
 $ injuries_crowd_any: num [1:139580] 0 0 0 0 0 0 0 0 0 0 ...
 $ chemical_agents   : num [1:139580] 0 0 0 0 0 0 0 0 0 0 ...
```

All of these variables are binary indicators of whether one of these events took place.

Let's say that we want to generate a vector that counts how many incidents of each type occurred:

```
map_int(incident_df, sum)
```

```
       arrests_any injuries_crowd_any    chemical_agents
              1776                441                124
```

The map variants will throw an error if your function doesn't output the correct type of data.

```
map_lgl(incident_df, sum)
```

This is what the error message should look like:

```
Error in `map_lgl()`:
i In index: 1.
i With name: arrests_any.
Caused by error:
! Can't coerce from a number to a logical.
Run `rlang::last_trace()` to see where the
error occurred.
```

```
Error in map_lgl(incident_df,     Show Traceback
 sum) :
 i With name: arrests_any.
 Caused by error:
 ! Can't coerce from a number to a logical.
```

If you're getting this sort of "Can't coerce" error, it means that you are using the wrong variant of map function.

Anonymous functions

Until now, we have been only working with *named* functions. Naming a function is useful when you need to run something more than once. If you only need to apply a function one time, you may want to consider using *anonymous functions* instead. We can just write the function body inside `map()` instead.

Remember the `count_missings` function we wrote earlier? Let's find a more streamlined way to count the number of missings across variables using `map()` and anonymous functions:

```
ccc_data %>% map_int(~ sum(is.na(.)))
```

date	locality	state
0	53	16
location_detail	online	type
9804	68	49
title	macroevent	organizations
84363	132123	38419
participants	claims	claims_summary
104366	9	671
claims_verbatim	issue_tags_summary	issue_tags_verbatim
97272	3525	105490
issue_tags	valence	size_text
2371	34	92885
size_low	size_high	size_mean
92783	92793	92783
size_cat	arrests	arrests_any
0	138045	0
injuries_crowd	injuries_crowd_any	injuries_police
139382	0	139803
injuries_police_any	property_damage	property_damage_any
0	139035	0
chemical_agents	participant_measures	police_measures
0	102520	129013
participant_deaths	police_deaths	source_1
139809	139822	80
source_2	source_3	source_4
78516	110953	125379
source_5	source_6	source_7
131851	134983	136451
source_8	source_9	source_10
137725	138421	138769
source_11	source_12	source_13
139009	139183	139309
source_14	source_15	source_16
139403	139463	139530
source_17	source_18	source_19
139577	139617	139647
source_20	source_21	source_22
139685	139709	139735
source_23	source_24	source_25
139754	139762	139775
source_26	source_27	source_28
139780	139786	139823
source_29	source_30	notes

```
          139823           139823            59756
             lat              lon  resolved_locality
              61               61              812
resolved_county     resolved_state        fips_code
           13546               93              126
```

If we are trying to calculate missing across multiple datasets, writing a named function may be helpful. But if we just want to count missings on one dataset, this is the way to go.

`dplyr::across()`

In addition to the map functions, `dplyr::across()` also offers options for iteration. It is typically used for data wrangling and applying the same function(s) across columns in a dataset.

Let's have a look at the documentation for this function:

```
?across
```

The three arguments that we should account for are:

- `.cols`: This tells R which columns in the data frame the function(s) should be applied to. You can pass a character vector, `tidyselect` helpers (e.g. starts_with("foo")), or use column positions.
- `.fns`:
- `.names` (optional): controls how the names of newly constructed columns will be constructed

Building on our example from before, let's say we are interested in inspecting the number of events where arrests occurred, protesters/civilians were injured, or chemical agents were deployed.

Let's first specify the three columns within the `ccc_clean` database that this would apply to, then apply the `sum` function `across` these columns

```
incident_cols <- c("arrests_any", "injuries_crowd_any", "chemical_agents")

incidents_by_state <- ccc_clean %>%
  group_by(state) %>%
  summarize(
    n_events = n(),
    across(all_of(incident_cols), sum)
  )
```

```r
head(incidents_by_state)
```

```
# A tibble: 6 x 5
  state n_events arrests_any injuries_crowd_any chemical_agents
  <chr>    <int>       <dbl>              <dbl>           <dbl>
1 AK         267           1                  0               0
2 AL        1698           7                  4               0
3 AR         485           4                  2               0
4 AZ        2445          35                 11               4
5 CA       21554         209                 96              16
6 CO        2838          17                  5               2
```

Let's build a custom function to apply to this state-level dataset. Let's say we are interested in the percentage of events for each type of incident relative to the total number of events in each state.

```r
calculate_percentage_col <- function(x, n_events) {
  x / n_events * 100
}
```

Now let's apply it to our `incidents_by_state` dataset

```r
incidents_tmp <- incidents_by_state %>%
  mutate(across(
    .cols = all_of(incident_cols),
    .fns = ~ .x / n_events * 100,
    .names = "perc_{.col}"
  ))
```

Finally, let's check the structure of this dataset to double check that it's working:

```r
str(incidents_tmp)
```

```
tibble [53 x 8] (S3: tbl_df/tbl/data.frame)
 $ state                 : chr [1:53] "AK" "AL" "AR" "AZ" ...
 $ n_events              : int [1:53] 267 1698 485 2445 21554 2838 1657 5867 172 4104 ...
 $ arrests_any           : num [1:53] 1 7 4 35 209 17 21 188 3 59 ...
 $ injuries_crowd_any    : num [1:53] 0 4 2 11 96 5 4 24 0 12 ...
 $ chemical_agents       : num [1:53] 0 0 0 4 16 2 0 7 0 2 ...
```

```
 $ perc_arrests_any       : num [1:53] 0.375 0.412 0.825 1.431 0.97 ...
 $ perc_injuries_crowd_any: num [1:53] 0 0.236 0.412 0.45 0.445 ...
 $ perc_chemical_agents   : num [1:53] 0 0 0 0.1636 0.0742 ...
```

Exercise 3a Based on `ccc_data`, use `across()` and an anonymous function to convert all of the the text in the `claims`, `organizations`, and `title` columns to lowercase.

```
text_vars <- c("claims", "organizations", "title")

ccc_data_lower <- ccc_data %>%
  mutate(across(all_of(text_vars), ~ tolower(.x)))
```

```
ccc_data_lower %>%
  select(claims, organizations, title)
```

```
# A tibble: 139,823 x 3
   claims                                                  organizations title
   <chr>                                                   <chr>         <chr>
 1 against prison labor, for safer conditions in alabama pr~ free alabama~ <NA>
 2 for ending israel's occupation of palestine, for palesti~ women in bla~ <NA>
 3 for president trump                                     contra costa~ <NA>
 4 for peace, against war, for banning nuclear weapons     wilpf         <NA>
 5 for president trump, against election fraud, against gov~ trump unity ~ patr~
 6 for canceling rent and providing financial relief during~ <NA>          <NA>
 7 for safer streets for bicyclists, for community-building  san francisc~ <NA>
 8 for peace                                               <NA>          frid~
 9 for free phone calls and video visitations and increased~ <NA>          <NA>
10 black lives matter, against police brutality            the valley o~ <NA>
# i 139,813 more rows
```

Exercise 3b Use `map_int()` to create a new column in the `ccc_clean` data frame that contains the number of **characters** in each entry of the `claims` column.

Hint: You can use the `nchar()` function to count characters in a string.

```
ccc_clean2 <- ccc_clean %>%
  mutate(n_chars = map_int(claims, ~ nchar(.x)))
```

If the function was written correctly, the following code should work:

```
ccc_clean2$n_chars[1:10]
```

```
 [1]  89  71  19  51 104  77  56   9 112  44
```

Tidy evaluation

If you mainly use `tidyverse` functions in your `R` workflow, functional programming will be a little more complicated. This is because, when it comes to working with variables programmatically, the tidyverse uses a system called **tidy evaluation**.

Writing functions that accept variables as arguments introduces a new challenge. This is because they are treated differently between base `R` and the `tidyverse`:

- In base `R`, variable names are typically treated as placeholders that refer to objects in the global environment.
- In contrast, `tidyverse` functions treat variable names as expressions that are evaluated inside the context of the data frame itself. This is what makes the tidyverse more user-friendly and dynamic; if a variable is treated as an expression, you can chain multiple functions together to manipulate data at scale.

This difference introduces a tension when we try to write our own functions. Inside our function, we want to let the user pass in column names, and use those column names in tidyverse functions such as `filter()`, `ggplot()`, and `group_by()`. But `R` doesn't automatically know that those names refer to columns inside a data frame, rather than variables in the function environment.

This is where **tidy evaluation** comes in. Tidy evaluation is the programming system that powers tidyverse functions, bridging the gap between how variables are interpreted in the global environment and how they are interpreted inside a tidy data frame.

The core tools of tidy evaluation are:

1. **{{}} (a.k.a. embracing):** Tells `R` to treat the argument as a column reference inside a data frame, e.g.:

```
group_and_summarize <- function(data, group_var) {
  data %>%
    group_by({{ group_var }}) %>%
    summarise(
      n_events = n(),
    )
}
```

2. **...:** Flexible arguments

When you want to pass a function though several expressions (e.g. `filter()` conditions or `group_by()` variables), e.g.:

```
df_filter <- function(...) {
  df %>% filter(...)
}
```

3. **:= (a.k.a. the "walrus operator")**: Allows the output name to be created from the input of a function, e.g.:

```
lowercase_column <- function(data, input_col, output_col) {
  data %>%
    mutate({{ output_col }} := tolower({{ input_col }}))
}
```

4. **!!!:** unpacks and passes multiple arguments in as if they were typed out manually, e.g.:

```
valence_map <- c(
  "0" = "none/other",
  "1" = "left-wing",
  "2" = "right-wing"
)

ccc_recoded <- ccc_data %>%
  mutate(valence_label = recode(as.character(valence), !!!valence_map))
```

Here's a table that can serve as a small cheat sheet for these four tools:

Tool	What it does	When to use it
`{ }`	Treats argument as a column name	Always, for tidy column inputs
`...`	Forwards user expressions	For flexible filters, groupings
`:=`	Assigns dynamic column names	When user names the output
`!!!`	Unpacks a list of named arguments	When arguments are stored in a list

Let's apply some tidy evaluation to data. There are two variables in the CCC data that include text strings that are separated by semi-colons: `type` and `claims`. Let's say we are interested in tabulating the types of events or claims over time:

Let's revisit the code we wrote for doing this based on `type`:

```r
ccc_types <- ccc_clean %>%
  separate_rows(type, sep = ";") %>%
  mutate(type == gsub("\\s+", "", type))
```

Now let's turn it into a function:

```r
separate_rows_by_var <- function(data, var){
  data_expanded <- data %>%
    separate_rows({{var}}, sep = ";") %>%
  mutate({{var}} == gsub("\\s+", "", {{var}}))
  return(data_expanded)
}
```

Great! Now we can separate the rows based on `type` or `claim`.

```r
separate_rows_by_var(ccc_clean, type)
separate_rows_by_var(ccc_clean, claims)
```

Exercise 4 Using a dataset of your choice, write a function that applies tidy evaluation.

Bonus exercise (if time)

Exercise 5 Practice writing functions based on an `R` script/dataset from your own work.

Additional resources

While we cover a lot of ground on applied examples in this short course, we will not be able to cover everything there is to know about functions. Here are some resources that you can consult on your own time in the event that you want to build on what we learned today!

- Debugging, condition handling, and defensive programming by Hadley Wickham (Advanced R4DS)
- Error Handling in R by Alboukadel Kassambara, PhD
- Functional programming by Sara Altman, Bill Behrman, Hadley Wickham
- Function Documentation by Hadley Wickham and Jennifer Bryan

References

Ulfelder, Jay. 2025. "Crowd Counting Consortium U.S. Protest Event Data, 2021-2024." Harvard Dataverse. https://doi.org/10.7910/DVN/9MMYDI.