Functions and tidy evaluation

Based on Chapter 25 from *R for Data Science*

You can download this .qmd file from here. Just hit the Download Raw File button.

Introduction (from Ch 25 of R4DS)

One of the best ways to improve your reach as a data scientist is to write functions. Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting. Writing a function has four big advantages over using copy-and-paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).
- It makes it easier to reuse work from project-to-project, increasing your productivity over time.

A good rule of thumb is to consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code). We'll learn about three useful types of functions:

- Vector functions take one or more vectors as input and return a vector as output.
- Data frame functions take a data frame as input and return a data frame as output.
- Plot functions that take a data frame as input and return a plot as output.

```r
# Initial packages required (we'll be adding more)
library(tidyverse)
library(nycflights13)
```

Do not Repeat Yourself: Also known as DRY, if you copy or paste code more than twice, you should write a function instead.

When writing a function, it is usually best to start with the code you know works for one instance, and then "function-ize" it.

Vector functions

Example 1: Rescale variables from 0 to 1.

This code creates a 10 x 4 tibble filled with random values taken from a normal distribution with mean 0 and SD 1

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
df
```

```
# A tibble: 10 x 4
         a       b      c      d
     <dbl>   <dbl>  <dbl>  <dbl>
 1  0.296   0.0247 -0.228 -0.392
 2  0.819  -1.68    0.978  0.172
 3 -0.0177  1.44    1.43  -0.325
 4  0.924   0.437  -0.450 -1.01
 5  1.23   -0.553   0.812  0.188
 6 -2.22    0.0649  0.836  2.78
 7  1.35   -1.34   -0.811 -1.45
 8 -0.884   1.24   -0.536 -1.14
 9  1.12   -0.246   1.02   0.147
10 -0.344  -1.71   -2.23  -2.21
```

This code below for rescaling variables from 0 to 1 is ripe for functions... we did it four times!

It's easiest to start with working code and turn it into a function.

```
df$a <- (df$a - min(df$a)) / (max(df$a) - min(df$a))
df$b <- (df$b - min(df$b)) / (max(df$b) - min(df$b))
df$c <- (df$c - min(df$c)) / (max(df$c) - min(df$c))
df$d <- (df$d - min(df$d)) / (max(df$d) - min(df$d))
df
```

```
# A tibble: 10 x 4
       a      b     c     d
   <dbl>  <dbl> <dbl> <dbl>
```

```
 1 0.704 0.551  0.548 0.364
 2 0.850 0.0117 0.877 0.477
 3 0.616 1      1     0.378
 4 0.880 0.681  0.487 0.241
 5 0.965 0.368  0.832 0.480
 6 0     0.564  0.838 1
 7 1     0.118  0.388 0.154
 8 0.374 0.937  0.464 0.214
 9 0.935 0.465  0.889 0.472
10 0.525 0      0     0
```

Notice first what changes and what stays the same in each line. Then, if we look at the first line above, we see we have one value we're using over and over: `df$a`. So our function will have one input. We'll start with our code from that line, then replace the input (df$a) with x. We should give our function a name that explains what it does. The name should be a verb.

```
# I'm going to show you how to write the function in class!
# I have it in the code already below, but don't look yet!
# Let's try to write it together first!
```

. . . .

```
# Our function (first draft!)
rescale01 <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}
```

Note the **general form of a function**:

```
name <- function(arguments) {
  body
}
```

Every function contains 3 essential components:

- A name. The name should clearly evoke what the function does; hence, it is often a verb (action). Here we'll use rescale01 because this function rescales a vector to lie between 0 and 1. snake_case is good; CamelCase is just okay.
- The arguments. The arguments are things that vary across calls and they are usually nouns - first the data, then other details. Our analysis above tells us that we have just one; we'll call it x because this is the conventional name for a numeric vector, but you can use any word.

- The body. The body is the code that's repeated across all the calls. By default a function will return the last statement; use `return()` to specify a return value

Summary: Functions should be written for both humans and computers!

Once we have written a function we like, then we need to test it with different inputs!

```r
temp <- c(4, 6, 8, 9)
rescale01(temp)
```

```
[1] 0.0 0.4 0.8 1.0
```

```r
temp0 <- c(4, 6, 8, 9, NA)
rescale01(temp0)
```

```
[1] NA NA NA NA NA
```

OK, so NA's don't work the way we want them to.

```r
rescale01 <- function(x) {
  (x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
}
rescale01(temp)
```

```
[1] 0.0 0.4 0.8 1.0
```

```r
rescale01(temp0)
```

```
[1] 0.0 0.4 0.8 1.0  NA
```

We can continue to improve our function. Here is another method, which uses the existing `range` function within R to avoid 3 max/min executions:

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(temp)
```

```
[1] 0.0 0.4 0.8 1.0
```

```
rescale01(c(0, 5, 10))
```

```
[1] 0.0 0.5 1.0
```

```
rescale01(c(-10, 0, 10))
```

```
[1] 0.0 0.5 1.0
```

```
rescale01(c(1, 2, 3, NA, 5))
```

```
[1] 0.00 0.25 0.50   NA 1.00
```

We should continue testing unusual inputs. Think carefully about how you want this function to behave… the current behavior is to include the Inf (infinity) value when calculating the range. You get strange output everywhere, but it's pretty clear that there is a problem right away when you use the function. In the example below (rescale1), you ignore the infinity value when calculating the range. The function returns Inf for one value, and sensible stuff for the rest. In many cases this may be useful, but it could also hide a problem until you get deeper into an analysis.

```
x <- c(1:10, Inf)
rescale01(x)
```

```
 [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

```
rescale1 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale1(x)
```

```
 [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
 [8] 0.7777778 0.8888889 1.0000000       Inf
```

Now we've used functions to simplify original example. We will learn to simplify further in iterations (Ch 26)

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
# add a little noise
df$a[5] = NA
df$b[6] = Inf
df
```

```
# A tibble: 10 x 4
        a       b      c       d
    <dbl>   <dbl>  <dbl>   <dbl>
 1  1.04    1.33    1.03  -2.03
 2  0.703  -1.09   -0.755 -0.845
 3  1.49   -0.168  -1.22   0.104
 4  0.721   0.792   1.28   1.36
 5 NA      -1.71    2.53   0.568
 6 -1.39  Inf       0.286 -0.0297
 7 -0.212   0.503   0.869 -1.58
 8  1.31    1.66   -0.892 -0.662
 9 -1.01   -0.0316 -0.417 -0.429
10  1.17   -0.187  -0.264  1.45
```

```
df$a_new <- rescale1(df$a)
df$b_new <- rescale1(df$b)
df$c_new <- rescale1(df$c)
df$d_new <- rescale1(df$d)
df
```

```
# A tibble: 10 x 8
        a       b      c       d  a_new   b_new  c_new d_new
    <dbl>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>  <dbl> <dbl>
 1  1.04    1.33    1.03  -2.03    0.846   0.902 0.600 0
 2  0.703  -1.09   -0.755 -0.845   0.727   0.185 0.125 0.340
 3  1.49   -0.168  -1.22   0.104   1       0.457 0     0.613
 4  0.721   0.792   1.28   1.36    0.734   0.742 0.667 0.973
 5 NA      -1.71    2.53   0.568  NA       0     1     0.746
 6 -1.39  Inf       0.286 -0.0297  0     Inf     0.402 0.574
 7 -0.212   0.503   0.869 -1.58    0.409   0.657 0.557 0.128
 8  1.31    1.66   -0.892 -0.662   0.938   1     0.0879 0.393
 9 -1.01   -0.0316 -0.417 -0.429   0.131   0.498 0.215  0.460
10  1.17   -0.187  -0.264  1.45    0.890   0.452 0.255  1
```

```
df %>%
  mutate(a_new = rescale1(a),
         b_new = rescale1(b),
         c_new = rescale1(c),
         d_new = rescale1(d))
```

```
# A tibble: 10 x 8
        a       b      c       d  a_new   b_new  c_new d_new
    <dbl>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>  <dbl> <dbl>
 1  1.04    1.33    1.03  -2.03    0.846   0.902 0.600  0
 2  0.703  -1.09   -0.755 -0.845   0.727   0.185 0.125  0.340
 3  1.49   -0.168  -1.22   0.104   1       0.457 0      0.613
 4  0.721   0.792   1.28   1.36    0.734   0.742 0.667  0.973
 5 NA      -1.71    2.53   0.568  NA       0     1      0.746
 6 -1.39   Inf      0.286 -0.0297  0     Inf     0.402  0.574
 7 -0.212   0.503   0.869 -1.58    0.409   0.657 0.557  0.128
 8  1.31    1.66   -0.892 -0.662   0.938   1     0.0879 0.393
 9 -1.01   -0.0316 -0.417 -0.429   0.131   0.498 0.215  0.460
10  1.17   -0.187  -0.264  1.45    0.890   0.452 0.255  1
```

```
# Even better - from Chapter 26
df |> mutate(across(a:d, rescale1))
```

```
# A tibble: 10 x 8
        a       b      c     d  a_new   b_new  c_new d_new
    <dbl>   <dbl>  <dbl> <dbl>  <dbl>   <dbl>  <dbl> <dbl>
 1  0.846   0.902 0.600  0      0.846   0.902 0.600  0
 2  0.727   0.185 0.125  0.340  0.727   0.185 0.125  0.340
 3  1       0.457 0      0.613  1       0.457 0      0.613
 4  0.734   0.742 0.667  0.973  0.734   0.742 0.667  0.973
 5 NA       0     1      0.746 NA       0     1      0.746
 6  0     Inf     0.402  0.574  0     Inf     0.402  0.574
 7  0.409   0.657 0.557  0.128  0.409   0.657 0.557  0.128
 8  0.938   1     0.0879 0.393  0.938   1     0.0879 0.393
 9  0.131   0.498 0.215  0.460  0.131   0.498 0.215  0.460
10  0.890   0.452 0.255  1      0.890   0.452 0.255  1
```

Options for handling NAs in functions

Before we try some practice problems, let's consider various options for handling NAs in functions. We used the `na.rm` option within functions like `min`, `max`, and `range` in order to take care of missing values. But there are alternative approaches:

- filter/remove the NA values before rescaling
- create an if statement to check if there are NAs; return an error if NAs exist
- create a removeNAs option in the function we are creating

Let's take a look at each alternative approach in turn:

Filter/remove the NA values before rescaling

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
df$a[5] = NA
df
```

```
# A tibble: 10 x 4
          a       b       c      d
      <dbl>   <dbl>   <dbl>  <dbl>
 1 -0.0257  -1.23   -1.10    0.327
 2 -1.74     2.30   -0.377   1.03
 3  1.44    -0.0767 -0.716   0.841
 4 -1.01     0.129  -0.704   1.51
 5 NA       -0.0936  0.407   1.40
 6  0.638   -0.0877  0.0898 -0.527
 7  0.00280 -0.102   0.860  -0.568
 8 -1.01    -0.247   0.810  -0.209
 9 -0.768   -0.224   0.185   1.04
10 -1.42     0.311   1.70    0.146
```

```
rescale_basic <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}
```

```
df %>%
  filter(!is.na(a)) %>%
  mutate(new_a = rescale_basic(a))
```

```
# A tibble: 9 x 5
        a       b       c      d new_a
    <dbl>   <dbl>   <dbl>  <dbl> <dbl>
1 -0.0257  -1.23   -1.10    0.327 0.539
2 -1.74     2.30   -0.377   1.03  0
3  1.44    -0.0767 -0.716   0.841 1
4 -1.01     0.129  -0.704   1.51  0.230
5  0.638   -0.0877  0.0898 -0.527 0.748
6  0.00280 -0.102   0.860  -0.568 0.548
7 -1.01    -0.247   0.810  -0.209 0.230
8 -0.768   -0.224   0.185   1.04  0.306
9 -1.42     0.311   1.70    0.146 0.101
```

[**Pause to Ponder:**] Do you notice anything in the output above that gives you pause?

- Filtering ahead of time takes out the entire row where a had an NA value.

Create an if statement to check if there are NAs; return an error if NAs exist

First, here's an example involving weighted means:

```
# Create function to calculate weighted mean
wt_mean <- function(x, w) {
  sum(x * w) / sum(w)
}
wt_mean(c(1, 10), c(1/3, 2/3))
```

```
[1] 7
```

```
wt_mean(1:6, 1:3)
```

```
[1] 7.666667
```

[**Pause to Ponder:**] Why is the answer to the last call above 7.67? Aren't we taking a weighted mean of 1-6, all of which are below 7? - The data and weights are unequal length, so R is not calculating it like we want.

```r
# update function to handle cases where data and weights of unequal length
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  } else {
  sum(w * x) / sum(w)
  }
}
wt_mean(1:6, 1:3)
```

```
Error: `x` and `w` must be the same length
```

```r
# should produce an error now if weights and data different lengths
#  - nice example of if and else
```

[**Pause to Ponder:**] What does the `call.` option do? - The `call.` option tells you in the error message where in the code the error is coming from.

Now let's apply this to our rescaling function

```r
rescale_w_error <- function(x) {
  if (is.na(sum(x))) {
    stop("`x` cannot have NAs", call. = FALSE)
  } else {
  (x - min(x)) / (max(x) - min(x))
  }
}

temp <- c(4, 6, 8, 9)
rescale_w_error(temp)
```

```
[1] 0.0 0.4 0.8 1.0
```

```r
temp <- c(4, 6, 8, 9, NA)
rescale_w_error(temp)
```

```
Error: `x` cannot have NAs
```

[**Pause to Ponder:**] Why can't we just use `if (is.na(x))` instead of `is.na(sum(x))`? - x is a vector so there is the possibility to have a list of values where some are na and some aren't. If at least one of the values is NA, taking the sum of x will result in NA and appropriately prompt the message that the vector x cannot include NAs.

Create a removeNAs option in the function we are creating

```r
rescale_NAoption <- function(x, removeNAs = FALSE) {
  (x - min(x, na.rm = removeNAs)) /
    (max(x, na.rm = removeNAs) - min(x, na.rm = removeNAs))
}

temp <- c(4, 6, 8, 9)
rescale_NAoption(temp)
```

```
[1] 0.0 0.4 0.8 1.0
```

```r
temp <- c(4, 6, 8, 9, NA)
rescale_NAoption(temp, removeNAs = TRUE)
```

```
[1] 0.0 0.4 0.8 1.0  NA
```

OK, but all the other summary stats functions use na.rm as the input, so to be consistent, it's probably better to do something slightly awkward like this:

```r
rescale_NAoption <- function(x, na.rm = FALSE) {
  (x - min(x, na.rm = na.rm)) /
    (max(x, na.rm = na.rm) - min(x, na.rm = na.rm))
}

temp <- c(4, 6, 8, 9, NA)
rescale_NAoption(temp, na.rm = TRUE)
```

```
[1] 0.0 0.4 0.8 1.0  NA
```

`wt_mean()` is an example of a "summary function (single value output) instead of a"mutate function" (vector output) like `rescale01()`. Here's another summary function to produce the mean absolute percentage error:

```r
mape <- function(actual, predicted) {
  sum(abs((actual - predicted) / actual)) / length(actual)
}

y <- c(2,6,3,8,5)
yhat <- c(2.5, 5.1, 4.4, 7.8, 6.1)
mape(actual = y, predicted = yhat)
```

```
[1] 0.2223333
```

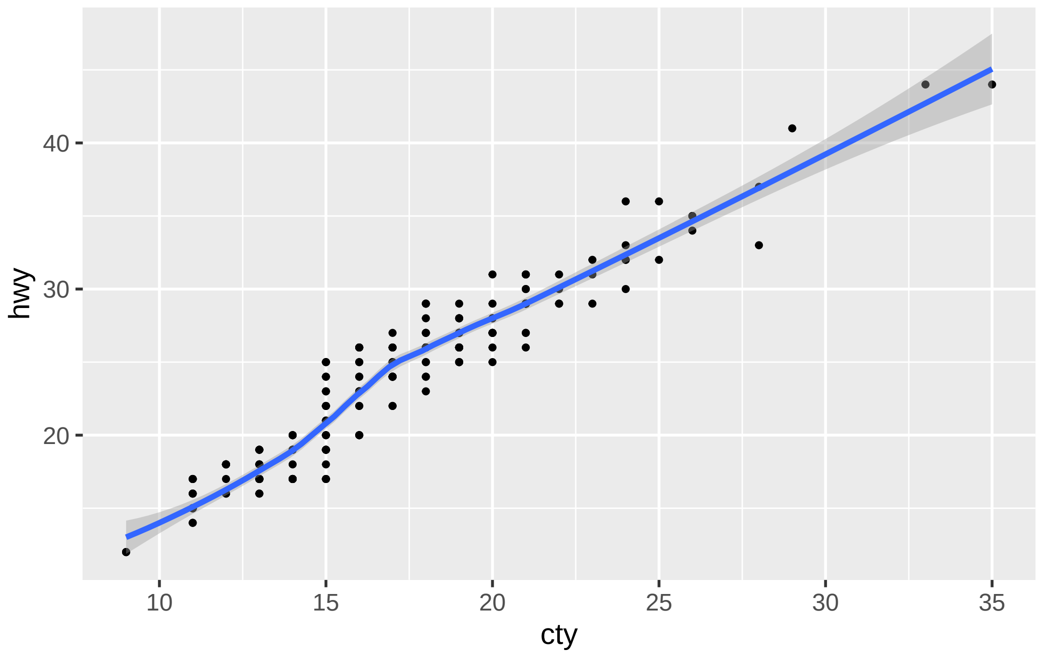
Data frame functions

These work like dplyr verbs, taking a data frame as the first argument, and then returning a data frame or a vector.

Demonstration of tidy evaluation in functions

```r
# Start with working code then functionize
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(size = 0.75) +
  geom_smooth()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



```r
make_plot <- function(dataset, xvar, yvar, pt_size = 0.75)  {
  ggplot(data = dataset, mapping = aes(x = xvar, y = yvar)) +
    geom_point(size = pt_size) +
    geom_smooth()
}

make_plot(dataset = mpg, xvar = cty, yvar = hwy)  # Error!
```

```
Error in `geom_point()`:
! Problem while computing aesthetics.
i Error occurred in the 1st layer.
Caused by error:
! object 'cty' not found
```

The problem is tidy evaluation, which makes most common coding easier, but makes some less common things harder. Key terms to understand tidy evaluation:

- env-variables = live in the environment (mpg)
- data-variables = live in data frame or tibble (cty)
- data masking = tidyverse use data-variables as if they are env-variables. That is, you don't always need `mpg$cty` to access `cty` in tidyverse

The key idea behind data masking is that it blurs the line between the two different meanings of the word "variable":

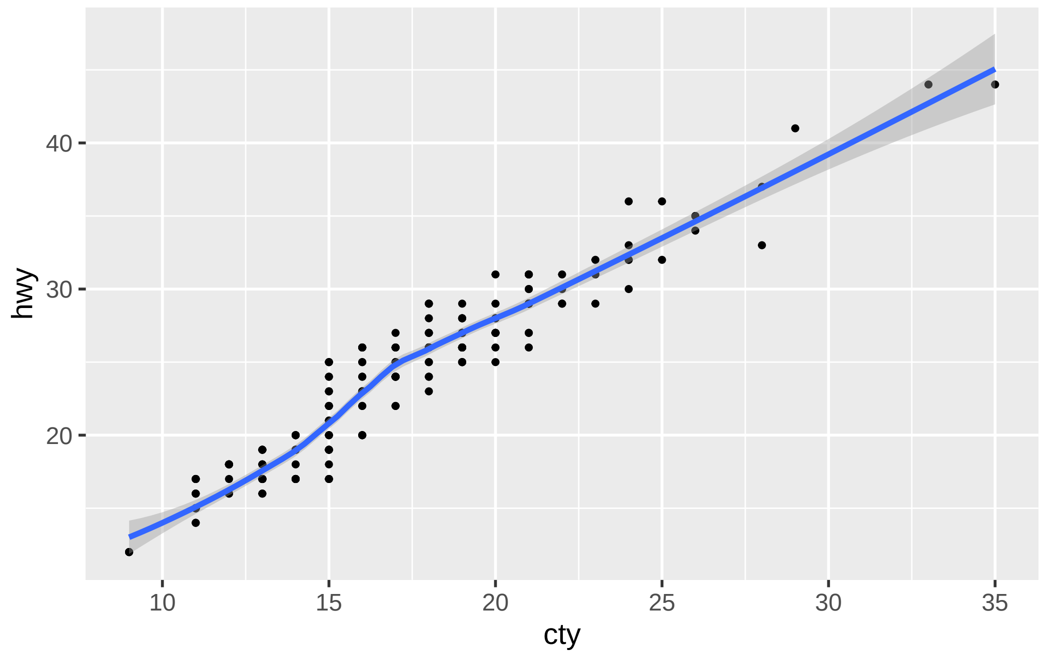
- env-variables are "programming" variables that live in an environment. They are usually created with <-.
- data-variables are "statistical" variables that live in a data frame. They usually come from data files (e.g. .csv, .xls), or are created manipulating existing variables.

The solution is to embrace {{ }} data-variables which are user inputs into functions. One way to remember what's happening, as suggested by our book authors, is to think of {{ }} as looking down a tunnel — {{ var }} will make a dplyr function look inside of `var` rather than looking for a variable called `var`. Thus, embracing a variable tells dplyr to use the value stored inside the argument, not the argument as the literal variable name.

See Section 25.3 of R4DS for more details (and there are plenty!).

```r
# This will work to make our plot!
make_plot <- function(dataset, xvar, yvar, pt_size = 0.75)  {
  ggplot(data = dataset, mapping = aes(x = {{ xvar }}, y = {{ yvar }})) +
    geom_point(size = pt_size) +
    geom_smooth()
}

make_plot(dataset = mpg, xvar = cty, yvar = hwy)
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



I often wish it were easier to get my own custom summary statistics for numeric variables in EDA rather than using `mosaic:favstats()`. Using `group_by()` and `summarise()` from the tidyverse reads clearly but takes so many lines, but if I only had to write the code once…

```
summary6 <- function(data, var) {
  data |> summarize(
    min = min({{ var }}, na.rm = TRUE),
    mean = mean({{ var }}, na.rm = TRUE),
    median = median({{ var }}, na.rm = TRUE),
    max = max({{ var }}, na.rm = TRUE),
    n = n(),
    n_miss = sum(is.na({{ var }})),
    .groups = "drop"    # to leave the data in an ungrouped state
  )
}

mpg |> summary6(hwy)
```

```
# A tibble: 1 x 6
    min  mean median   max     n n_miss
  <int> <dbl>  <dbl> <int> <int>  <int>
1    12  23.4     24    44   234      0
```

Even cooler, I can use my new function with `group_by()`!

```
mpg |>
  group_by(drv) |>
  summary6(hwy)
```

```
# A tibble: 3 x 7
  drv     min  mean median   max     n n_miss
  <chr> <int> <dbl>  <dbl> <int> <int>  <int>
1 4        12  19.2     18    28   103      0
2 f        17  28.2     28    44   106      0
3 r        15  21       21    26    25      0
```

You can even pass conditions into a function using the embrace:

[**Pause to Ponder:**] Predict what the code below will do, and (only) then run it to check. Think about: why do we have `sort = sort`? why not embrace `df`? why didn't we need `n` in the arguments?

- `sort = sort` arranges from the most to least according to the variable in the count function. We put `sort = TRUE` in the argument for the function, so we add `sort = sort` to the count to indicate that we are interested in the sort being applied here.

- We don't embrace `df` because the dataframe lives in the environment (it is an env-variable). Only variables within a dataset (data variables) need to be embraced because they don't live directly in the R environment.
- We don't need `n` in the arguments because it is generated by the count function.

```
new_function <- function(df, var, condition, sort = TRUE) {
  df |>
    filter({{ condition }}) |>
    count({{ var }}, sort = sort) |>
    mutate(prop = n / sum(n))
}

mpg |> new_function(var = manufacturer,
                    condition = manufacturer %in% c("audi", "honda", "hyundai", "nissan",
```

Data-masking vs. tidy-selection (Section 25.3.4)

Why doesn't the following code work?

```r
count_missing <- function(df, group_vars, x_var) {
  df |>
    group_by({{ group_vars }}) |>
    summarize(
      n_miss = sum(is.na({{ x_var }})),
      .groups = "drop"
    )
}

flights |>
  count_missing(c(year, month, day), dep_time)
```

```
Error in `group_by()`:
i In argument: `c(year, month, day)`.
Caused by error:
! `c(year, month, day)` must be size 336776 or 1, not 1010328.
```

The problem is that `group_by()` uses data-masking rather than tidy-selection; it is selecting certain variables rather than evaluating values of those variables. These are the two most common subtypes of tidy evaluation:

- Data-masking is used in functions like arrange(), filter(), mutate(), and summarize() that compute with variables. Data masking is an R feature that blends programming variables that live inside environments (env-variables) with statistical variables stored in data frames (data-variables).

- Tidy-selection is used for functions like select(), relocate(), and rename() that select variables. Tidy selection provides a concise dialect of R for selecting variables based on their names or properties.

More detail can be found here.

The error above can be solved by using the `pick()` function, which uses tidy selection inside of data masking:

```r
count_missing <- function(df, group_vars, x_var) {
  df |>
    group_by(pick({{ group_vars }})) |>
    summarize(
      n_miss = sum(is.na({{ x_var }})),
      .groups = "drop"
  )
}

flights |>
  count_missing(c(year, month, day), dep_time)
```

```
# A tibble: 365 x 4
    year month   day n_miss
   <int> <int> <int>  <int>
 1  2013     1     1      4
 2  2013     1     2      8
 3  2013     1     3     10
 4  2013     1     4      6
 5  2013     1     5      3
 6  2013     1     6      1
 7  2013     1     7      3
 8  2013     1     8      4
 9  2013     1     9      5
10  2013     1    10      3
# i 355 more rows
```

[**Pause to Ponder:**] Here's another nice use of `pick()`. Predict what the function will do, then run the code to see if you are correct.

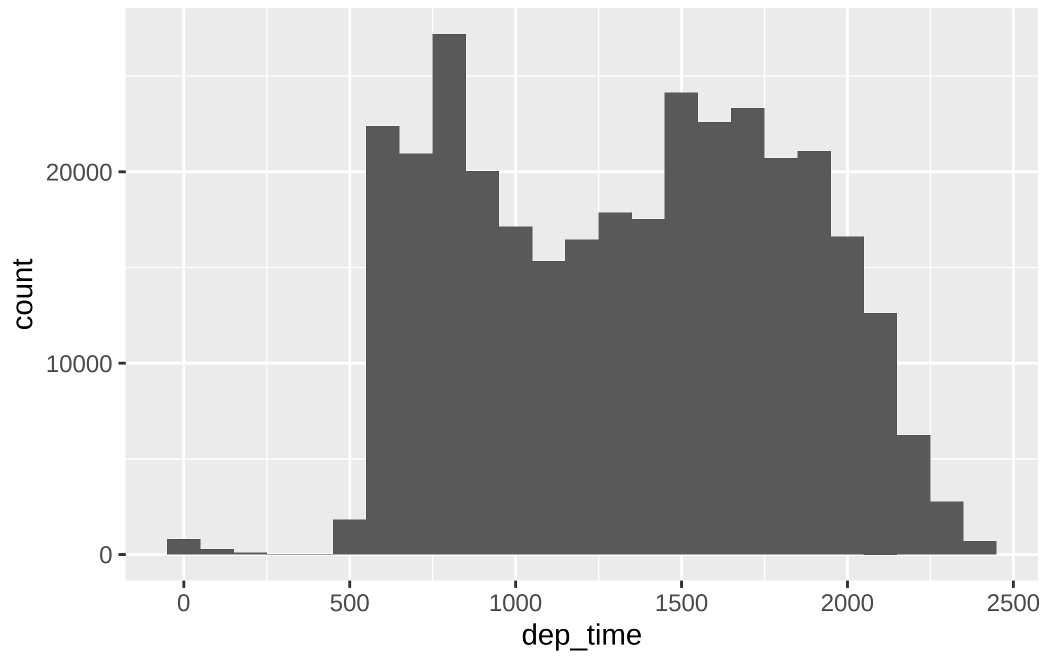
- This function pivots wider to create new columns based on the different values represented in the column `cols`, which in this case is the variable `cyl`. Then the values under these columns are determined by the number of each unique manufacture-mode-cyls combo.

```
# Source: https://twitter.com/pollicipes/status/1571606508944719876
new_function <- function(data, rows, cols) {
  data |>
    count(pick(c({{ rows }}, {{ cols }}))) |>
    pivot_wider(
      names_from = {{ cols }},
      values_from = n,
      names_sort = TRUE,
      values_fill = 0
    )
}

mpg |> new_function(c(manufacturer, model), cyl)
```

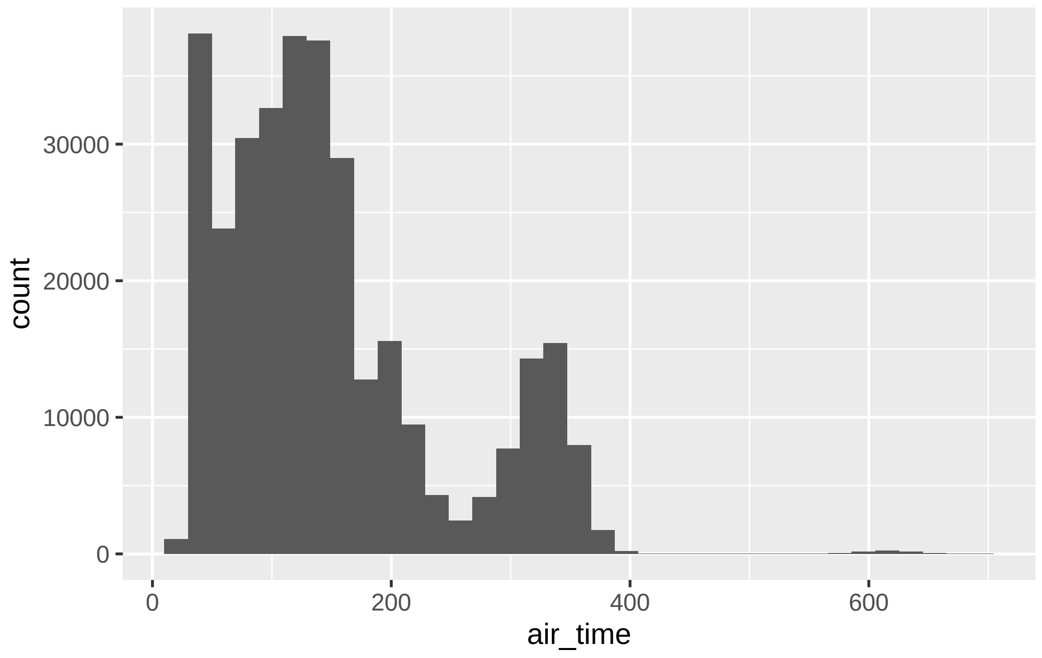
Plot functions

Let's say you find yourself making a lot of histograms:

```
flights |>
  ggplot(aes(x = dep_time)) +
  geom_histogram(bins = 25)
```

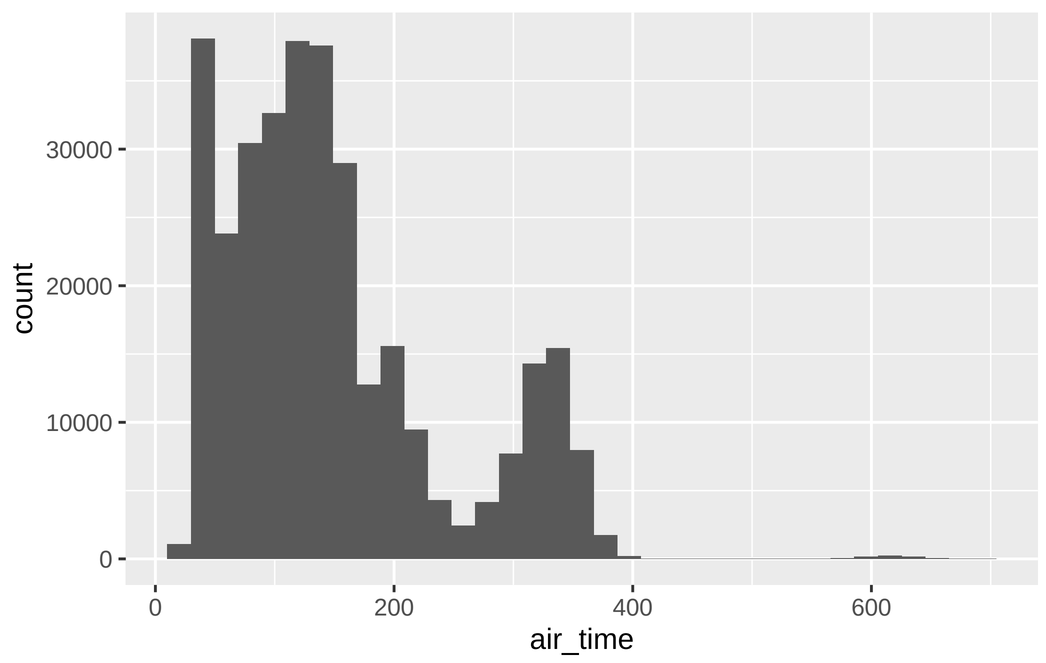


```
flights |>
  ggplot(aes(x = air_time)) +
  geom_histogram(bins = 35)
```



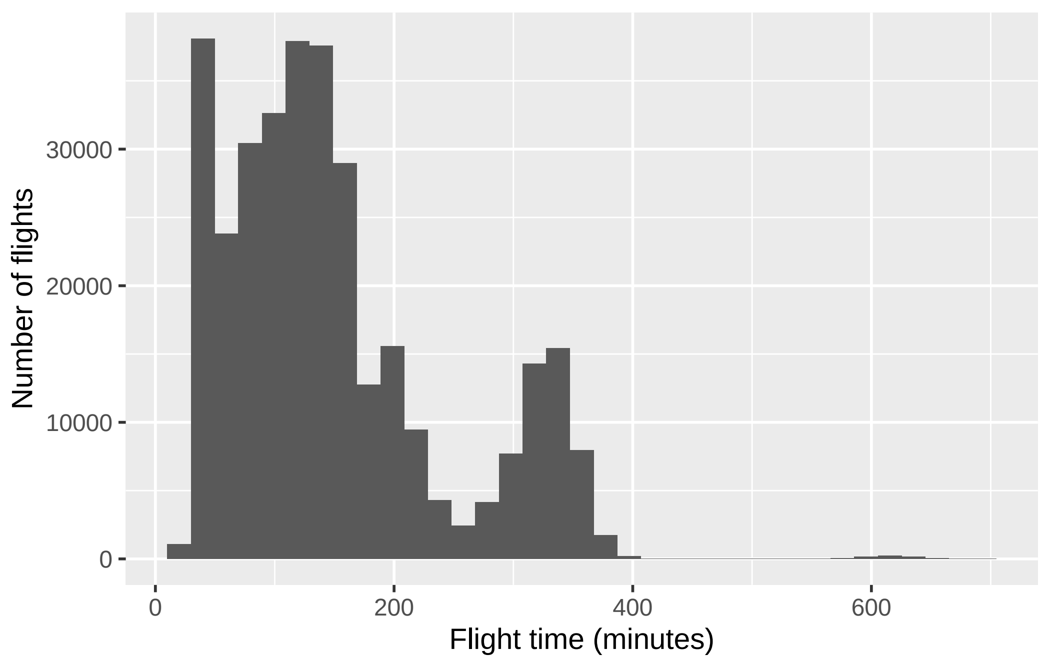
Just use embrace to create a histogram-making function

```
histogram <- function(df, var, bins = NULL) {
  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(bins = bins)
}

flights |> histogram(air_time, 35)
```



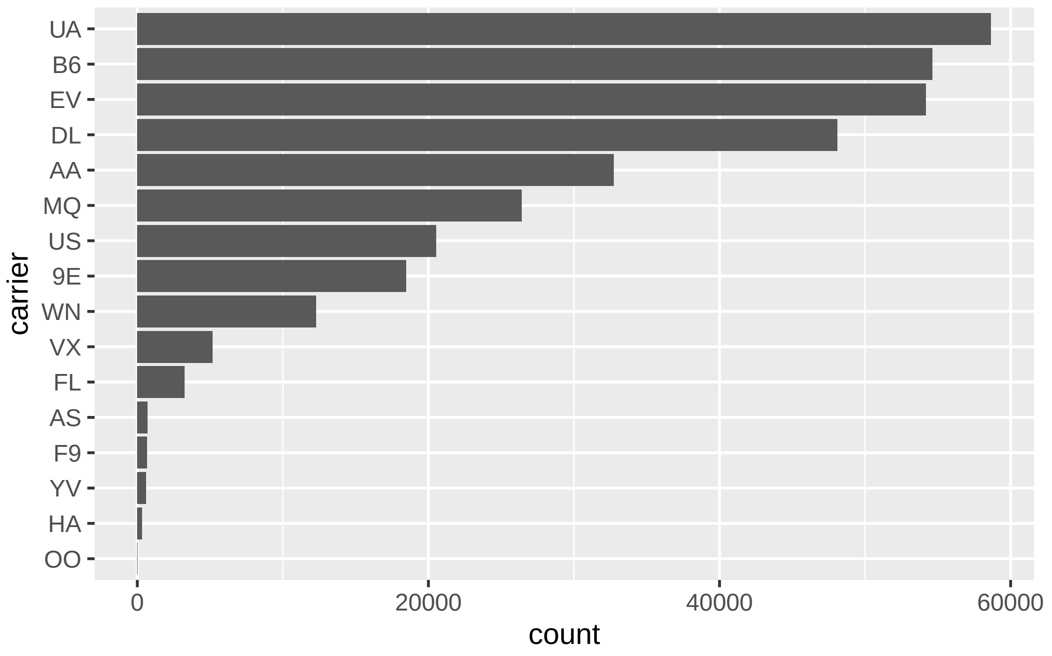
Since histogram() returns a ggplot, you can add any layers you want

```
flights |>
  histogram(air_time, 35) +
  labs(x = "Flight time (minutes)", y = "Number of flights")
```



You can also combine data wrangling with plotting. Note that we need the "walrus operator" (:=) since the variable name on the left is being generated with user-supplied data.

```
# sort counts with highest values at top and counts on x-axis
sorted_bars <- function(df, var) {
  df |>
    mutate({{ var }} := fct_rev(fct_infreq({{ var }})))  |>
    ggplot(aes(y = {{ var }})) +
    geom_bar()
}

flights |> sorted_bars(carrier)
```

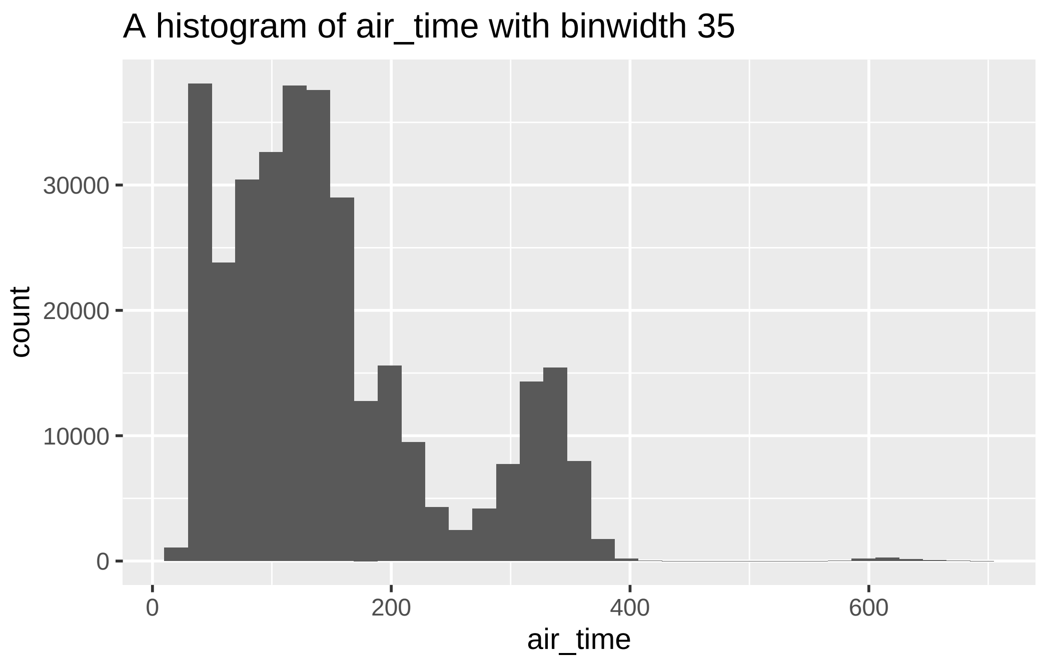


Finally, it would be really helpful to label plots based on user inputs. This is a bit more complicated, but still do-able!

For this, we'll need the `rlang` package. `rlang` is a low-level package that's used by just about every other package in the tidyverse because it implements tidy evaluation (as well as many other useful tools).

Check out the following update of our `histogram()` function which uses the `englue()` function from the `rlang` package:

```r
histogram <- function(df, var, bins) {
  label <- rlang::englue("A histogram of {{var}} with binwidth {bins}")

  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(bins = bins) +
    labs(title = label)
}

flights |> histogram(air_time, 35)
```



On Your Own

1. Rewrite this code snippet as a function: `x / sum(x, na.rm = TRUE)`. This code creates weights which sum to 1, where NA values are ignored. Test it for at least two different vectors. (Make sure at least one has NAs!)

```
weights_sum_1 <- function(x) {
  x / sum(x, na.rm = TRUE)
}

temp_vector_1 <- c(4, 6, 8, 9)
weights_sum_1(temp_vector_1)
```

```
[1] 0.1481481 0.2222222 0.2962963 0.3333333
```

```
temp_vector_2 <- c(4, 9, NA, 2)
weights_sum_1(temp_vector_2)
```

```
[1] 0.2666667 0.6000000        NA 0.1333333
```

2. Create a function to calculate the standard error of a variable, where SE = square root of the variance divided by the sample size. Hint: start with a vector like `x <- 0:5` or `x <- gss_cat$age` and write code to find the SE of x, then turn it into a function to handle any vector `x`. Note: `var` is the function to find variance in R and `sqrt` does square root. `length` may also be handy. Test your function on two vectors that do not include NAs (i.e. do **not** worry about removing NAs at this point).

```r
standard_error <- function(x) {
  sd(x) / sqrt(length(x))
}

test <- 0:5
standard_error(test)
```

```
[1] 0.7637626
```

```r
test_2 <- 0:10
standard_error(test_2)
```

```
[1] 1
```

3. Use your `se` function within summarize to get a table of the mean and s.e. of `hwy` and `cty` by `class` in the `mpg` dataset.

```r
mpg |>
  group_by(class) |>
  summarize(mean_cty = mean(hwy),
            se_cty = standard_error(hwy),
            mean_hwy = mean(cty),
            se_hwy = standard_error(cty))
```

```
# A tibble: 7 x 5
  class      mean_cty se_cty mean_hwy se_hwy
  <chr>         <dbl>  <dbl>    <dbl>  <dbl>
1 2seater        24.8  0.583     15.4  0.245
2 compact        28.3  0.552     20.1  0.494
3 midsize        27.3  0.334     18.8  0.304
4 minivan        22.4  0.622     15.8  0.553
5 pickup         16.9  0.396     13    0.356
6 subcompact     28.1  0.909     20.4  0.778
7 suv            18.1  0.378     13.5  0.307
```

4. Use your `se` function within summarize to get a table of the mean and s.e. of `arr_delay` and `dep_delay` by carrier in the `flights` dataset. Why does the output look like this?

```
flights |>
  group_by(carrier) |>
  summarise(mean_arr_delay = mean(arr_delay),
            se_arr_delay = standard_error(arr_delay),
            mean_dep_delay = mean(dep_delay),
            se_dep_delay = standard_error(dep_delay))
```

```
# A tibble: 16 x 5
   carrier mean_arr_delay se_arr_delay mean_dep_delay se_dep_delay
   <chr>            <dbl>        <dbl>          <dbl>        <dbl>
 1 9E               NA           NA             NA           NA
 2 AA               NA           NA             NA           NA
 3 AS               NA           NA             NA           NA
 4 B6               NA           NA             NA           NA
 5 DL               NA           NA             NA           NA
 6 EV               NA           NA             NA           NA
 7 F9               NA           NA             NA           NA
 8 FL               NA           NA             NA           NA
 9 HA               -6.92         4.06           4.90         4.01
10 MQ               NA           NA             NA           NA
11 OO               NA           NA             NA           NA
12 UA               NA           NA             NA           NA
13 US               NA           NA             NA           NA
14 VX               NA           NA             NA           NA
15 WN               NA           NA             NA           NA
16 YV               NA           NA             NA           NA
```

- The output looks like this because the NAs have not been removed.

5. Make your `se` function handle NAs with an na.rm option. Test your new function (you can call it `se` again) on a vector that doesn't include NA and on the same vector with an added NA. **Be sure to check that it gives the expected output with na.rm = TRUE and na.rm = FALSE.** Make na.rm = FALSE the default value. Repeat #4. (Hint: be sure when you divide by sample size you don't count any NAs)

```
standard_error <- function(x, na.rm = FALSE) {
   n <- length(x) - sum(is.na(x))
   sqrt(var(x, na.rm = na.rm) / n)
}
```

```r
test2 <- c(0:5)
standard_error(test2, na.rm = TRUE)
```

```
[1] 0.7637626
```

```r
test2 <- c(0:5, NA)
standard_error(test2, na.rm = TRUE)
```

```
[1] 0.7637626
```

```r
flights |>
  group_by(carrier) |>
  summarise(mean_arr_delay = mean(arr_delay, na.rm = TRUE),
            se_arr_delay = standard_error(arr_delay, na.rm = TRUE),
            mean_dep_delay = mean(dep_delay, na.rm = TRUE),
            se_dep_delay = standard_error(dep_delay, na.rm = TRUE))
```

```
# A tibble: 16 x 5
   carrier mean_arr_delay se_arr_delay mean_dep_delay se_dep_delay
   <chr>            <dbl>        <dbl>          <dbl>        <dbl>
 1 9E               7.38         0.381          16.7         0.348
 2 AA               0.364        0.238           8.59        0.209
 3 AS              -9.93         1.37            5.80        1.18
 4 B6               9.46         0.184          13.0         0.165
 5 DL               1.64         0.203           9.26        0.182
 6 EV              15.8          0.221          20.0         0.205
 7 F9              21.9          2.36           20.2         2.23
 8 FL              20.1          0.960          18.7         0.933
 9 HA              -6.92         4.06            4.90        4.01
10 MQ              10.8          0.273          10.6         0.247
11 OO              11.9          9.02           12.6         8.00
12 UA               3.56         0.170          12.1         0.148
13 US               2.13         0.235           3.78        0.199
14 VX               1.76         0.699          12.9         0.626
15 WN               9.65         0.427          17.7         0.394
16 YV              15.6          2.27           19.0         2.11
```

6. Create `both_na()`, a function that takes two vectors of the same length and returns how many positions have an NA in both vectors. Hint: create two vectors like `test_x <- c(1, 2, 3, NA, NA)` and `test_y <- c(NA, 1, 2, 3, NA)` and write code that works for `test_x` and `test_y`, then turn it into a function that can handle any `x` and `y`. (In this case, the answer would be 1, since both vectors have NA in the 5th position.) Test it for at least one more combination of `x` and `y`.

```r
both_na <- function(x, y) {
  sum(is.na(x) & is.na(y))
}

test_x <- c(1, 2, 3, NA, NA)
test_y <- c(NA, 1, 2, 3, NA)

both_na(test_x, test_y)
```

```
[1] 1
```

```r
test_a <- c(1, 2, NA, 3, NA)
test_b <- c(NA, 1, NA, 3, NA)

both_na(test_a, test_b)
```

```
[1] 2
```

7. Run your code from (6) with the following two vectors: `test_x <- c(1, 2, 3, NA, NA, NA)` and `test_y <- c(NA, 1, 2, 3, NA)`. Did you get the output you wanted or expected? Modify your function using `if`, `else`, and `stop` to print an error if x and y are not the same length. Then test again with `test_x`, `test_y` and the sets of vectors you used in (6).

```r
both_na <- function(x, y) {
  if (length(x) != length(y)) {
    stop("Vector lengths do not match.")
  }
  else
  sum(is.na(x) & is.na(y))
}

test_x_extra_NA <- c(1, 2, 3, NA, NA, NA)
```

```r
test_y <- c(NA, 1, 2, 3, NA)

both_na(test_x_extra_NA, test_y)
```

```
Error in both_na(test_x_extra_NA, test_y): Vector lengths do not match.
```

```r
test_x <- c(1, 2, 3, NA, NA)
test_y <- c(NA, 1, 2, 3, NA)

both_na(test_x, test_y)
```

```
[1] 1
```

```r
test_a <- c(1, 2, NA, 3, NA)
test_b <- c(NA, 1, NA, 3, NA)

both_na(test_a, test_b)
```

```
[1] 2
```

8. Here is a way to get `not_cancelled` flights in the flights dataset:

```r
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
```

Is it necessary to check is.na for both departure and arrival? Using summarize, find the number of flights missing departure delay, arrival delay, and both. (Use your new function!)

```r
flights |>
  summarize(dep_delay_NA = sum(is.na(dep_delay)),
            arr_delay_NA = sum(is.na(arr_delay)),
            both_delay_NA = both_na(dep_delay, arr_delay))
```

```
# A tibble: 1 x 3
  dep_delay_NA arr_delay_NA both_delay_NA
         <int>        <int>         <int>
1         8255         9430          8255
```

- It appears that it is not necessary to check is.na for both departure and arrival because if a flight was missing departure delay it was probably also missing arrival delay. The more confusing question is why there are so many flights missing arrival delays that are not accounted for within the departure delays.

9. Read the code for each of the following three functions, puzzle out what they do, and then brainstorm better names.

```r
f1 <- function(time1, time2) {
  hour1 <- time1 %/% 100
  min1 <- time1 %% 100
  hour2 <- time2 %/% 100
  min2 <- time2 %% 100

  (hour2 - hour1)*60 + (min2 - min1)
}


f2 <- function(lengthcm, widthcm) {
  (lengthcm / 2.54) * (widthcm / 2.54)
}


f3 <- function(x) {
  fct_collapse(x, "non answer" = c("No answer", "Refused",
                                   "Don't know", "Not applicable"))
}
```

- f1 finds the duration in minutes. I would name this `duration_minutes`.
- f2 is a function for calculating square inches. I would name this `inches_squared`.
- f3 is collapsing the answers listed into the category of `non answer`. I would name this `collapse_non_answer`.

10. Explain what the following function does and demonstrate by running `foo1(x)` with a few appropriately chosen vectors `x`. (Hint: set x and run the "guts" of the function piece by piece.)

```r
foo1 <- function(x) {
  diff <- x[-1] - x[1:(length(x) - 1)]
  sum(diff < 0)
}
```

```r
temp_a <- c(1, 2, 3, 4, 5)
foo1(temp_a)
```

```
[1] 0
```

```r
temp_b <- c(1, 2, 3, 2, 1)
foo1(temp_b)
```

```
[1] 2
```

```r
temp_c <- c(1, 2, 3, 4, 3, 2, 1)
foo1(temp_c)
```

```
[1] 3
```

```r
temp_d <- c(1, 2, 3, 2, 3, 2)
foo1(temp_d)
```

```
[1] 2
```

- The first part of this function creates a vector in which the first value in the vector X is removed and then it creates another vector with the last value in the vector X removed. The second vector created is then subtracted from the first. The sum function finds the amount of times the difference is less than zero.

11. The `foo1()` function doesn't perform well if a vector has missing values. Amend `foo1()` so that it produces a helpful error message and stops if there are any missing values in the input vector. Show that it works with appropriately chosen vectors `x`. Be sure you add `error = TRUE` to your R chunk, or else knitting will fail!

```r
foo1 <- function(x) {
  if (is.na(sum(x))) {
    stop("X cannot have NAs")
  } else {
  diff <- x[-1] - x[1:(length(x) - 1)]
  sum(diff < 0)
  }
}

temp_vector_x <- c(1, 2, 4, 1)

foo1(temp_vector_x)
```

```
[1] 1
```

```r
temp_vector_y <- c(1, 2, 3, NA)

foo1(temp_vector_y)
```

```
Error in foo1(temp_vector_y): X cannot have NAs
```

12. Write a function called `greet` using `if`, `else if`, and `else` to print out "good morning" if it's before 12 PM, "good afternoon" if it's between 12 PM and 5 PM, and "good evening" if it's after 5 PM. Your function should work if you input a time like: `greet(time = "2018-05-03 17:38:01 CDT")` or if you input the current time with `greet(time = Sys.time())`. [Hint: check out the `hour` function in the `lubridate` package]

```r
hour(Sys.time())
```

```
[1] 20
```

```r
hour("2018-05-03 17:38:01 CDT")
```

```
[1] 17
```

```r
greet <- function(time = Sys.time()) {
  if(hour(time) < 12) {
    print("good morning")
  } else if (hour(time) < 17) {
    print("good afternoon")
  } else {
  print("good evening")
  }
}
```

```r
greet()
```

```
[1] "good evening"
```

13. Modify the `summary6()` function from earlier to add an argument that gives the user an option to remove missing values, if any exist. Show that your function works for (a) the `hwy` variable in `mpg_tbl <- as_tibble(mpg)`, and (b) the `age` variable in `gss_cat`.

```r
summary6 <- function(data, var, na.rm = FALSE) {
  data |> summarize(
    min = min({{ var }}, na.rm = na.rm),
    mean = mean({{ var }}, na.rm = na.rm),
    median = median({{ var }}, na.rm = na.rm),
    max = max({{ var }}, na.rm = na.rm),
    n = n(),
    n_miss = sum(is.na({{ var }})),
    .groups = "drop"    # to leave the data in an ungrouped state
  )
}

mpg_tbl <- as_tibble(mpg)

mpg_tbl |> summary6(hwy)
```

```
# A tibble: 1 x 6
    min  mean median   max     n n_miss
  <int> <dbl>  <dbl> <int> <int>  <int>
1    12  23.4     24    44   234      0
```

```r
gss_cat |> summary6(age, na.rm = TRUE)
```

```
# A tibble: 1 x 6
    min  mean median   max     n n_miss
  <int> <dbl>  <int> <int> <int>  <int>
1    18  47.2     46    89 21483     76
```

14. Add an argument to (13) to produce summary statistics by group for a second variable (you should now have 4 possible inputs to your function). Show that your function works for (a) the `hwy` variable in `mpg_tbl <- as_tibble(mpg)` grouped by `drv`, and (b) the `age` variable in `gss_cat` grouped by `partyid`.

```r
summary6 <- function(data, var, group_var, na_option = FALSE) {
  data |>
    group_by(pick({{ group_var }})) |>
    summarize(
      mean = mean({{ var }}, na.rm = na_option),
      median = median({{ var }}, na.rm = na_option),
      sd = sd({{ var }}, na.rm = na_option),
      IQR = IQR({{ var }}, na.rm = na_option),
      n = n(),
      n_miss = sum(is.na({{ var }})),
      .groups = "drop"    # to leave the data in an ungrouped state
    )
}

mpg_tbl <- as_tibble(mpg)
summary6(data = mpg_tbl, var = hwy, group_var = drv)
```

```
# A tibble: 3 x 7
  drv    mean median    sd   IQR     n n_miss
  <chr> <dbl>  <dbl> <dbl> <dbl> <int>  <int>
1 4      19.2     18  4.08     5   103      0
2 f      28.2     28  4.21     3   106      0
3 r      21       21  3.66     7    25      0
```

```r
party_age <- summary6(
  data = gss_cat,
  var = age,
  group_var = partyid,
  na_option = TRUE
)
party_age
```

```
# A tibble: 10 x 7
   partyid             mean median    sd   IQR     n n_miss
   <fct>              <dbl>  <dbl> <dbl> <dbl> <int>  <int>
 1 No answer           50.8   48    18.7    28   154      9
 2 Don't know          34     34    NA       0     1      0
 3 Other party         45.2   44.5  15.8    23   393      3
 4 Strong republican   51.9   51    17.0    26  2314      8
 5 Not str republican  47.2   45    17.2    26  3032      8
 6 Ind,near rep         47.1   46     17.1   27     1791        2
 7 Independent          43.3   41     16.3   24     4119       18
 8 Ind,near dem         44.9   43     17.1   27     2499        2
 9 Not str democrat     46.5   44     17.3   26.5   3690       11
10 Strong democrat      51.2   50     17.4   27     3490       15
```

15. Create a function that has a vector as the input and returns the last value. (Note: Be sure to use a name that does not write over an existing function!)

```
last_value <- function(x) {
  tail({{x}}, 1)
}

temp_vector_a <- c(1, 2, 3, 4, 3)

last_value(temp_vector_a)
```

```
[1] 3
```

```
temp_vector_b <- c(1, 2, 5, 3, 1)

last_value(temp_vector_b)
```

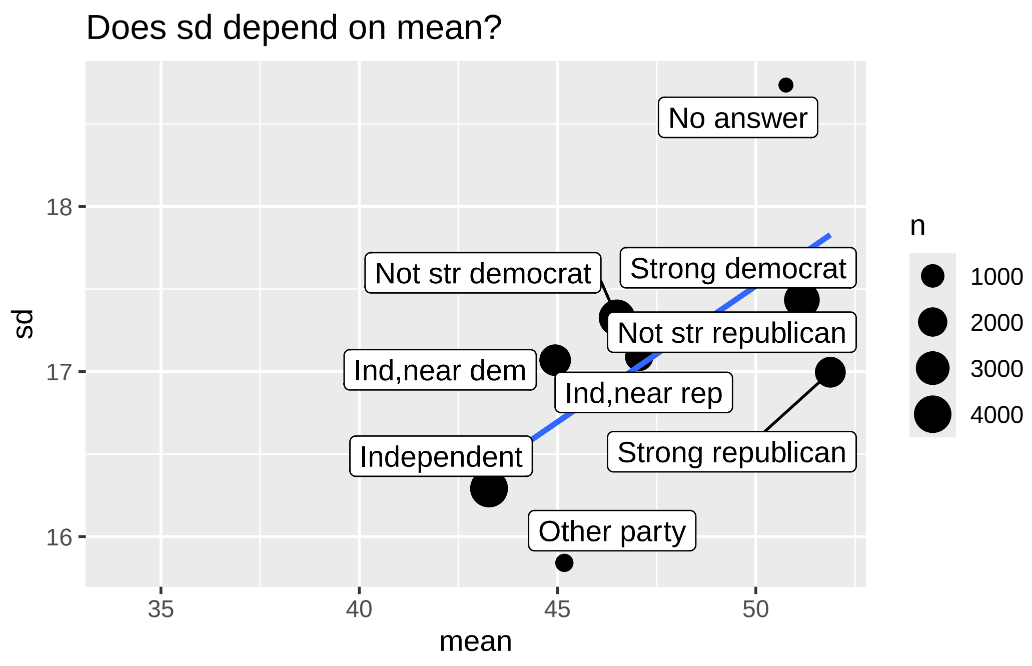
```
[1] 1
```

16. Save your final table from (14) and write a function to draw a scatterplot of a measure of center (mean or median - user can choose) vs. a measure of spread (sd or IQR - user can choose), with points sized by sample size, to see if there is constant variance. Each point should be labeled with partyid, and the plot title should reflect the variables chosen by the user.

```
library(ggrepel)
constant_var_check <- function(df, center, spread) {
  label <- rlang::englue("Does {{spread}} depend on {{center}}?")

df |>
  ggplot(aes(x = {{center}}, y = {{spread}})) +
  geom_point(aes(size = n)) +
  geom_smooth(method = lm, se = FALSE) +
  geom_label_repel(aes(label = partyid))+
  labs(title = label)
```

```r
}

constant_var_check(df = party_age, center = mean, spread = sd)
```

```
`geom_smooth()` using formula = 'y ~ x'

Warning: Removed 1 row containing non-finite outside the scale range
(`stat_smooth()`).

Warning: Removed 1 row containing missing values or values outside the scale range
(`geom_point()`).

Warning: Removed 1 row containing missing values or values outside the scale range
(`geom_label_repel()`).
```


Does sd depend on mean?
No answer
Not str democrat
Strong democrat
Not str republican
Ind,near dem
Ind,near rep
Independent
Strong republican
Other party
sd
mean
n