Week 10: Writing Functions

Ellen Bledsoe

2024-03-28

Writing Our Own Functions

Understandable and Reusable code

Writing our own functions can help us to write code in understandable chunks and also make more reusable code.

We often run into instances where we want to do something many times but slightly differently.

If you find yourself needing to run the same calculations or make the same plot repeatedly, writing a function to accomplish that task can save you from copying and pasting the same code over and over again.

Doing this can reduce errors in your code as well as condense the code into more understandable bits.

Function Structure

Below, I've written out the general structure of a function.

```
# function_name <- function(inputs) {
#   output <- do_something(inputs)
#   return(output)
# }
```

The braces indicate that the lines of code are a group that gets run together.

- A function will run all of the lines of code in the braces using the arguments provided.
- We then need to return the output using the `return()` function.

Let's write out our first function. We want to calculate shrub volumes.

```
calc_shrub_vol <- function(length, width, height) {
  area <- length * width
  volume <- area * height
  return(volume)
}
```

It is important to note that writing the function does not create the function. We have to remember to run the code to create the function and add it to our environment.

When we run this code chunk above, we can see that the function gets added to our environment in a new section called "functions."

Let's use our new function.

```
calc_shrub_vol(0.8, 1.6, 2.0)
```

```
## [1] 2.56
```

As with other functions, we can store the output as an object to use later.

```
shrub_vol <- calc_shrub_vol(0.8, 1.6, 2.0)
```

Let's Practice! Work on Question 1 in the Assignment.

Black Box

It is important to remember a few things about functions. In many ways, we need to treat functions as a black box.

What we mean by this is that the only thing that the function knows about are the inputs we give it.

Similarly, the only thing our environment know about the function is the output that we ask the function to return to us.

So, in the function above, we can't "access" variables or arguments that are created within the function. The function exists within it's own environment, so to speak.

```
width
volume
```

Let's Practice! Work on Question 2 in the Assignment.

Default arguments

As with many of the functions that we have already used in this class, we can set defaults for the arguments in the functions that we create.

For example, if many of our shrubs are the same height, we could specify that the default height should be 1.

```
calc_shrub_vol <- function(length, width, height = 1) {
  area <- length * width
  volume <- area * height
  return(volume)
}

calc_shrub_vol(0.8, 1.6)
```

```
## [1] 1.28
```

```
calc_shrub_vol(0.8, 1.6, 2.0)
```

```
## [1] 2.56
```

```
calc_shrub_vol(length = 0.8, width = 1.6, height = 2.0)
```

```
## [1] 2.56
```

Combining Functions

Again, as with other functions, we can combine functions of our own creation together.

```
est_shrub_mass <- function(volume){
  mass <- 2.65 * volume^0.9
}

shrub_volume <- calc_shrub_vol(0.8, 1.6, 2.0)
shrub_mass <- est_shrub_mass(shrub_volume)
```

We can also use pipes with our own functions. The output from the first function becomes the first argument for the second function.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
shrub_mass <- calc_shrub_vol(0.8, 1.6, 2.0) %>%
  est_shrub_mass()
```

Using the `tidyverse` in Functions

One of the really nice things about the `tidyverse` is that we usually don't need to put columns in quotes.

This is because they use "tidy evaluation," a special type of non-standard evaluation. Basically, they do fancy things under the surface to make them easier to work with.

This is useful for when *using* functions from the `tidyverse`, but it means that we need to add an extra step when we use `tidyverse` functions within our own functions that we are *writing*.

If we try to use `tidyverse` functions the way we normally would within our functions, they won't work.

First, let's load our packages and get some data from Palmer Penguins.

```
library(ggplot2)
library(palmerpenguins)

penguins <- penguins
```

Now, let's write some code for a ggplot within a function as we normally would.

```
make_plot <- function(df, column, label) {
  ggplot(data = df, mapping = aes(x = column)) +
    geom_histogram() +
    xlab(label)
}

make_plot(penguins, body_mass_g, "Body Mass (g)")
```

To fix this issue, we have to tell our code which inputs/arguments are this special type of data variable.

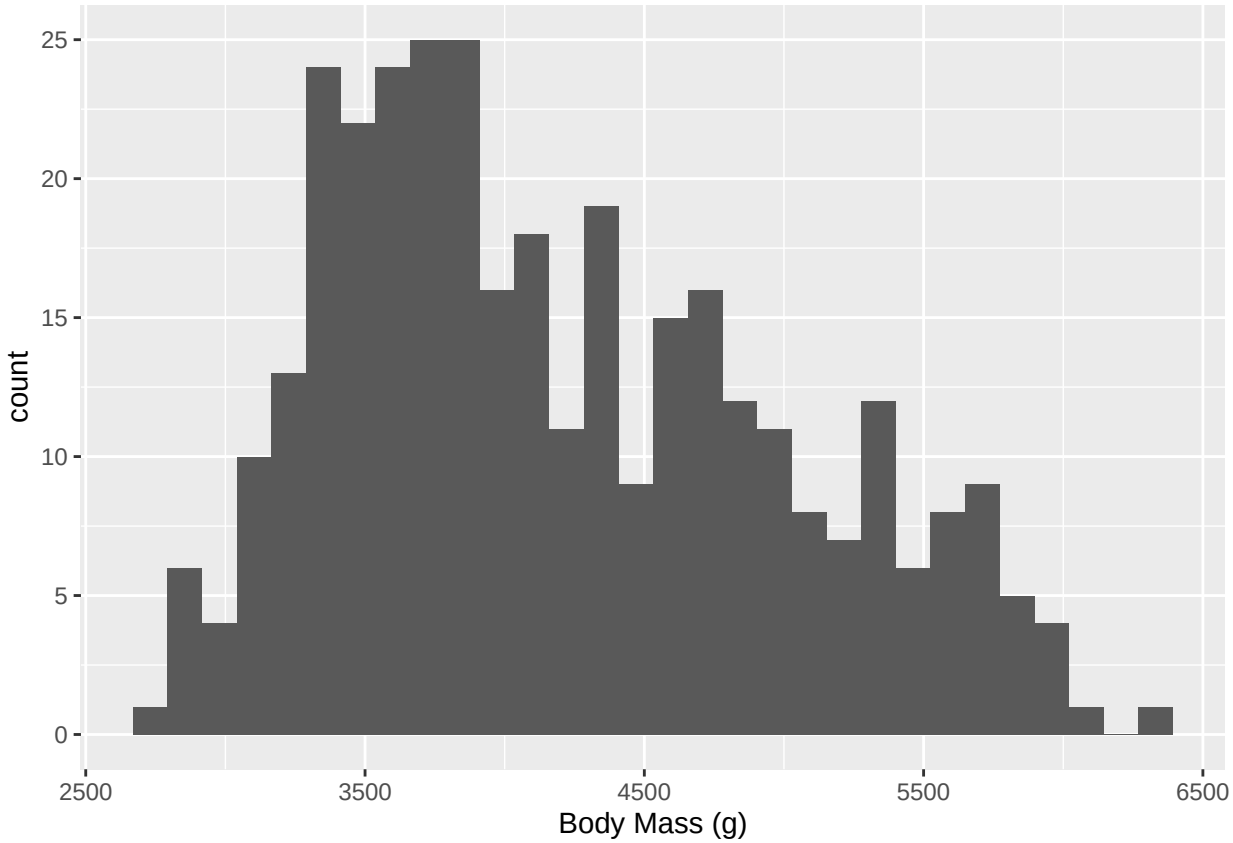
We can do this by "embracing" them in double braces. This tells the function to treat them in the "tidy evaluation" method.

```
make_plot <- function(df, column, label) {
    ggplot(data = df, mapping = aes(x = {{ column }})) +
    geom_histogram() +
    xlab(label)
}

make_plot(penguins, body_mass_g, "Body Mass (g)")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

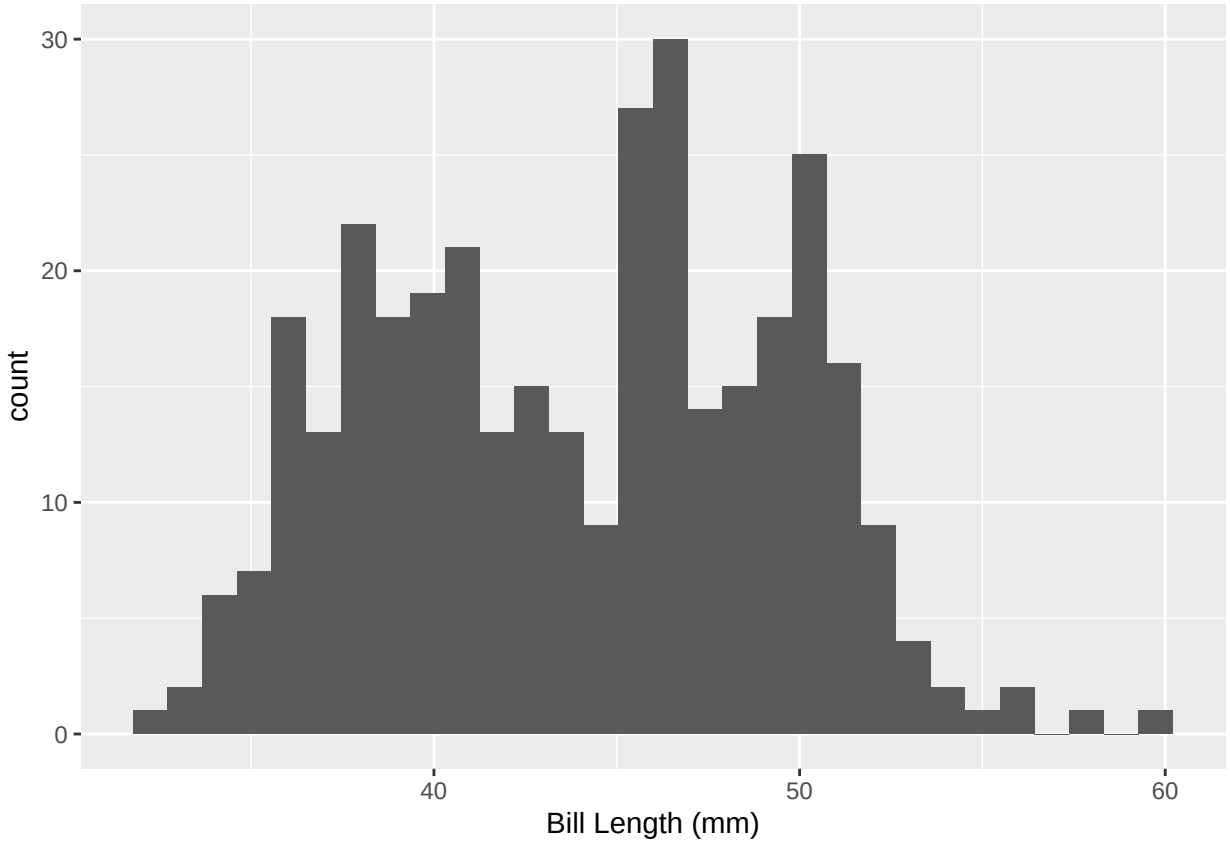
```
## Warning: Removed 2 rows containing non-finite values (`stat_bin()`).
```



```
make_plot(penguins, bill_length_mm, "Bill Length (mm)")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 2 rows containing non-finite values (`stat_bin()`).
```



Code Design with Functions

Functions let us break code up into logical chunks that can be understood in isolation.

When you write functions, place them at the top of your code then call them below.

The functions hold the details. You can heavily comment the code of your functions, as well.

The function calls will show you the outline of the code execution.

```
clean_data <- function(data){
  do_stuff(data)
}

process_data <- function(cleaned_data){
  do_dplyr_stuff(cleaned_data)
}

make_graph <- function(processed_data){
  do_ggplot_stuff(processed_data)
```

```
}

raw_data <- read.csv('mydata.csv')
cleaned_data <- clean_data(raw_data)
processed_data <- process_data(cleaned_data)
make_graph(processed_data)
```