

08: Coding Style and Wrangling II

Graphical Analysis of Biological Data

This web page is available as a [PDF file](#)

Reading and resources

I've spread the reading assignments and related resources throughout these notes.

Coding Style

Good code not only runs error-free but is easy to read, too. The [Tidyverse style guide](#) has good recommendations on writing functional and *readable* code. I expect you to follow the conventions used in the style guide.

Coding style: files

Chapter 1 of the style guide covers names and the structure of your file. Start organizing your code into sections. I tend to use the following sections for most of my R scripts.

- Load libraries
- Make global variables used through the script.
 - `file_path <- "data/"` is a good example.
- Custom functions. You will learn to write functions below.
- Data import
- Data wrangling
- Main analysis (if needed)
- Graphing

If you are writing a script that is not using markdown (files that are nothing but R code, and have a .R extension), the Code > Insert Section... from the menu inserts the nice section dividers shown in section 1.2 of the style guide. I am writing this in a notebook so that option is not available.

The section divider shown in the style guide begins with a #, which creates a header line in markdown format. If you used that divider, it would create a big bold line, which you wouldn't want in your notebook. You can mimic a section divider by using *** followed by a #### header, like that shown here. *** draws a rule across the page, followed by a small header.

```
***
#### Load libraries
```

A good reason to use section dividers is that you can use them for "code folding". You can show and hide sections separated by the dividers by clicking on small triangles between the line number and the # of the section divider¹. Hiding code you are not working on makes it much easier to navigate around your file. Here is an example.

¹You may have noticed these arrows in your previous notebooks

Load libraries

```
library(tidyverse)
library(smatr)
```

Global variables

```
file_path <- "data/"
```

Functions

```
sex_change <- function(x) {
  ifelse(x == "f", "female", "male")
}
```

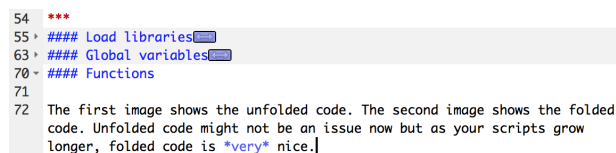
This image shows the unfolded code. Notice the small triangles next to the line numbers point downward, indicating unfolded code.



The screenshot shows an RStudio code editor with three code blocks. Each block is preceded by a line number and a downward-pointing triangle, indicating that the code is unfolded. The first block, starting at line 54, is titled 'Load libraries' and contains code to load 'tidyverse' and 'vegan'. The second block, starting at line 62, is titled 'Global variables' and contains code to set 'file_path'. The third block, starting at line 69, is titled 'Functions' and contains a function definition for 'sex_change'. The code is color-coded: comments are in red, function names in blue, and other code in black.

Figure 1: Unfolded code.

This image shows the first two sections folded. The triangle points to the right, indicating folded code. You click on the triangle or click on the two-headed arrow icon adjacent to your divider title to expand and view the code. You can also choose options from the Edit > Folding > menu.



The screenshot shows the same RStudio code editor as Figure 1, but the first two code blocks are now folded. The downward-pointing triangles next to line numbers 54 and 62 have changed to right-pointing triangles. The third block, starting at line 70, remains unfolded. The code is color-coded: comments are in red, function names in blue, and other code in black.

Figure 2: Unfolded code.

Unfolded code might not be an issue now but as your scripts grow longer, folded code is *very* nice. I encourage you to give it a try. When you write code outside of notebooks, use the section dividers provided by RStudio.

Coding style: syntax

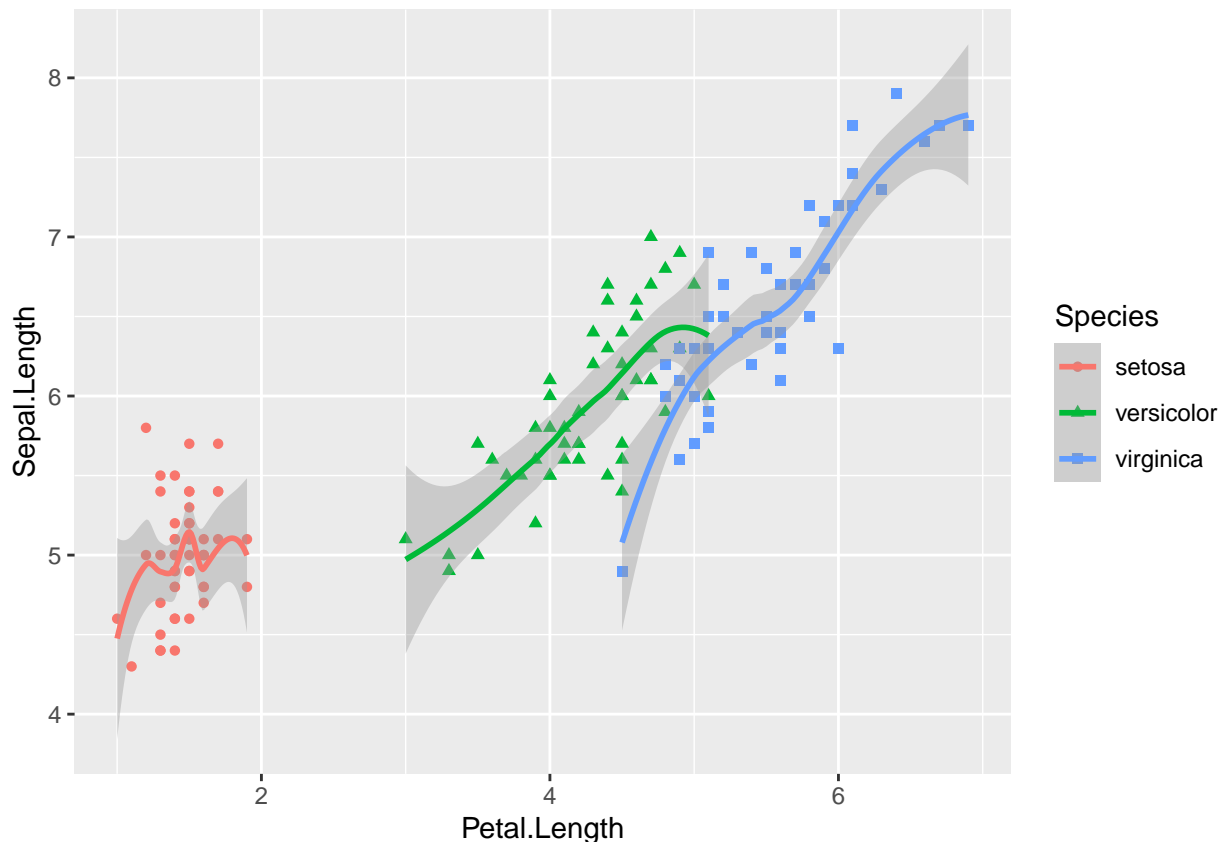
Chapter 2 covers the format of your code. Focus especially on object names, spacing, and the use of indents. The proper use of spaces and indents is one of the easiest ways of improving the readability of your code. Consider these two examples. The first is hard to read and hard to find mistakes. In fact, can you find the mistake?

```
data(iris)
ggplot(data=iris,aes(x=Petal.Length,y=Sepal.Length))+geom_point(aes(color=Species,shape=Species))+
geom_smooth(aes(color=Species))
```

Here is the same code, nicely formatted, and with the missing parenthesis mistake fixed.

```
ggplot(data = iris,
       aes(x = Petal.Length,
           y = Sepal.Length)) +
  geom_point(aes(color = Species,
                 shape = Species)) +
  geom_smooth(aes(color = Species))
```

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



The code and the *logic* of the code is much easier to read. Notice that I pressed the enter key after every comma, I have spaces around the equal signs, and that the + ends each line before I add another geom. Every time I pressed the Enter key, RStudio automatically indented the lines.

You can also highlight a code chunk, and then choose Code > Reformat Code or Code > Reindent Lines for help formatting your code. Neither of these options exactly match some of the recommendation made by the style guide. My example also did not correspond exactly to the guide. As long as you follow the guide reasonably close, you and I will be the happier for it.

Note: By spreading your code across more lines, the length of your file grows longer, increasing the value of code folding. *Fold your code!*

I expect you to follow these formatting guidelines in your code chunks and scripts. Writing code is just like writing text. Write legibly!

Coding style: pipes

Pipe style closely mirrors syntax style, like spacing and breaking up long lines.

Wrangling II

Read [R4ds Chapter 5: Data transformation](#). This chapter covers most of the other important functions for wrangling data. The three functions covered with the greatest depth are

- `mutate()`
- `summarize()`
- `group_by()`

These few functions harness a lot of power. Take your time working through the assignment, and really think about what you are doing. As usual, you will have to apply your newly learned skills in the next assignment.

You can use `mutate()` to change variables depending on the values of the starting variables, using `ifelse()` and `case_when()`. If you have only two values, use `ifelse()`. `ifelse()` has this structure: `ifelse(eval, TRUE, FALSE)`. The first argument is the evaluation, The middle argument is returned if the evaluation is TRUE. The last argument is returned if the evaluation is FALSE.

Assume you have a `sex` column that with values `f` and `m`. You want to change the column so that it reads `female` or `male`. To change this with `ifelse()`, use

```
data %>%  
  mutate(new_column = ifelse(sex == "f",  
                             "female",  
                             "male"))
```

If `sex` equals `f`, then the `new_column` is assigned `female`. If not `f`, then the `new_column` is assigned `male`.

If you have more than two choices, `case_when()` is a better option, but the structure is more complex. Let's use the `darther.csv` example from the previous assignment. Substrate (`majsub`) was defined as `s`, `fg`, `sg`, `lg`, and `c`, which stands for sand, fine gravel, small gravel, large gravel, and cobble. If you want to change the initials to the words, then use `case_when()`,

```
darther_data %>% mutate(new_sub = case_when(  
  majsub == "s" ~ "sand",  
  majsub == "sg" ~ "small_gravel",  
  TRUE ~ "other"))
```

This structure is different from typical R structures so some adjustment is required. `case_when` evaluates each option and assigns the value if it matches, and otherwise assigns the TRUE value. Walking through the example above, if `majsub` equals the value `s`, then the `new_sub` column is assigned the value of `sand`. If not equal, then `case_when()` goes to the next argument. If `majsub` equals the value of `sg`, then the `new_sub` column is assigned the value of `small_gravel`. If `majsub` is not equal to `s` or `sg`, then the `new_sub` column is assigned the value of `other`.

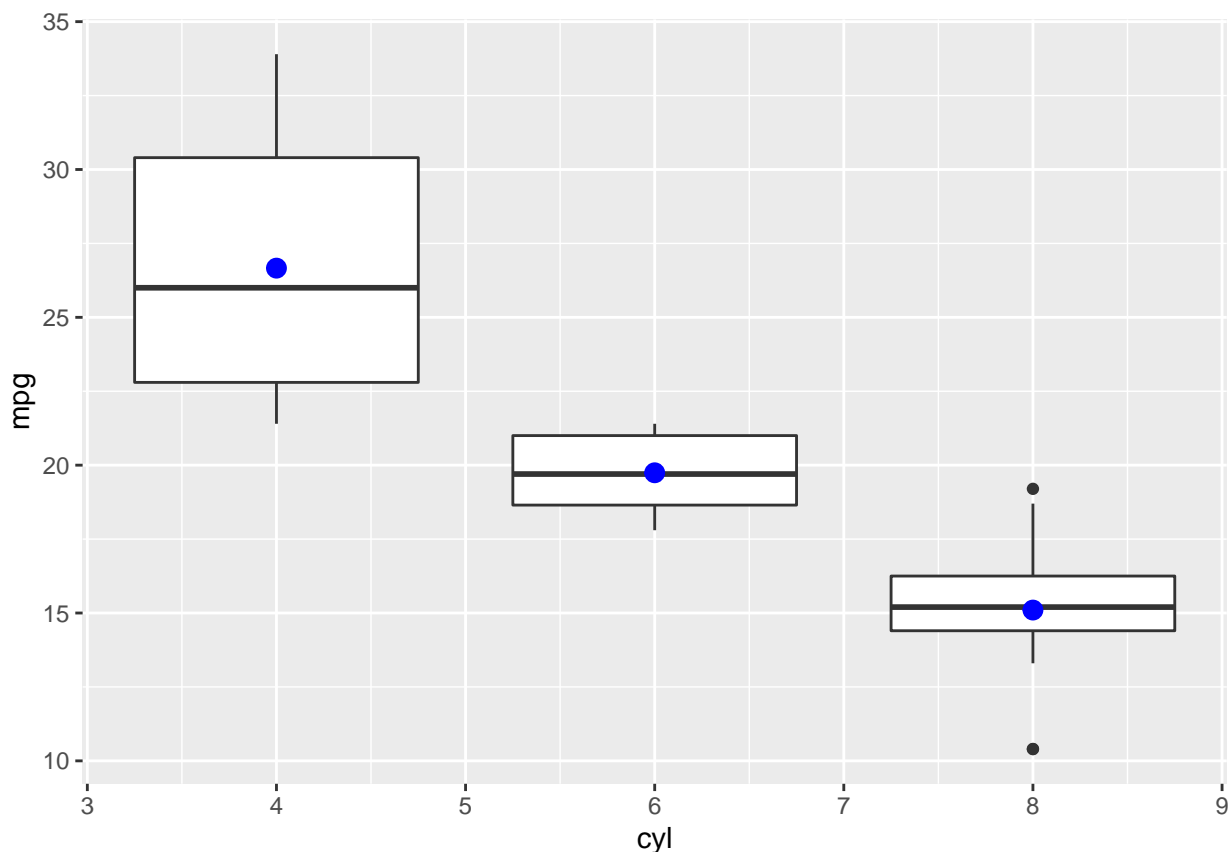
You will have to use only `ifelse()` for this assignment but be prepared for `case_when` situations in upcoming assignments.

A graphing tip

As you learned before, you can pipe your data to `ggplot()` for graphing. I use this technique regularly. Sometimes, you want to add a layer from another data set. You can do this, as shown below, because each geom can accept a separate `data =` argument.

```
# Create a new tibble summarizing data from the mtcars tibble.
avg_mpg <- mtcars %>%
  group_by(cyl) %>%
  summarize(mean_mpg = mean(mpg))

# Start with the mtcars tibble.
mtcars %>% ggplot() +
  geom_boxplot(aes(x = cyl,
                  y = mpg,
                  group = cyl)) +
  geom_point(data = avg_mpg, # Add data from avg_mpg tibble.
            aes(x = cyl,
                y = mean_mpg),
            color = "blue",
            size = 3)
```



Writing custom functions

Writing functions in R is a powerful skill to own. Writing robust functions takes many years to master but you can learn to write basic functions with an hour or two of practice. Start the clock!

Assume that you have three numbers, 5, 11, and 19. You want to add them together to get the total and you also want to find their average. You could accomplish those simple tasks in R with the following code.

```
total <- 5 + 11 + 19
average <- total/3

# Display the results
total
average
```

```
## [1] 35
## [1] 11.66667
```

That was not a lot of work. The amount of work would increase enormously if you had a vector with hundreds of numbers or hundreds of vectors with hundreds of numbers.

Fortunately, R provides **functions** called `sum()` and `mean()` that do the hard, repetitive, or tedious work for you.

```
# Create a vector of numbers
numbers <- c(5, 11, 19) # c() is a function!

total <- sum(numbers) # Use the sum function
average <- mean(numbers) # Use the mean function

# Display the results
total
```

```
## [1] 35
average
```

```
## [1] 11.66667
```

R and its many packages provide useful functions to accomplish many common (and uncommon) tasks. You are familiar with some of these, including `mean()`, `sum()`, `read_csv()`, `ggplot()`, and its associated geoms.

In the future, you will find a need for a custom function that is not built-in to R or available in an R package. The steps below show you how to write a function called `average` to calculate the arithmetic mean (“average”) of a vector of numbers. That is, you’ll write a function to mimic R’s built-in `mean()` function. You will compare the results of your function to the results of the `mean()` function so that you are confident your function (*ahem*) functions properly.

You will have to write two functions in the associated homework so be sure you get comfortable writing functions.

Format of a function

Functions have this form, described below.

```
your_function_name <- function(arguments) {
  code to do the work
  return(result)
}
```

`your_function_name` is the name you will use when you call your function, like `sum` or `mean`. `function()` is the built-in R function that tells R you are about to make a function. `arguments` accept any data that you pass to the function. In the example above, 5, 11, and 19 are the arguments passed to the `c()` function to create the `numbers` variable. The `numbers` variable was the argument passed to the `sum()` function. Finally, “code to do the work” is the R code that performs the actual work of the function. *Notice that the code to do the work is between curly braces {}* `return()` makes the result available to you after the work of the function is done.

Your “average” function.

The average function you are about to write will return the arithmetic mean from a vector of numbers. The arithmetic mean adds or sums together all of the numbers in a vector, and then divides the total by the number of elements in the vector. Your function must do these two steps to return the average.

We’ll begin with the basic form and then build on it.

```
average <- function(args) {  
  
}
```

Sometimes, especially for complex functions, writing *pseudocode* is helpful. Pseudocode is regular (non-coding) sentences that outlines each step of the function. The pseudocode below outlines the two steps needed to calculate the average, plus the final step to return the result.

```
average <- function(args) {  
  Add the numbers  
  divide by the number of elements in the vector  
  return the result  
}
```

Next, replace each line of pseudocode with real code. The addition of the numbers is handled by the `sum()` function. The sum is stored in the variable `total`. Notice I’m leaving the pseudocode in as comments to document the function.

```
average <- function(args) {  
  # Add the numbers, store in total  
  total <- sum(args)  
  divide by the number of elements in the vector  
  return the result  
}
```

Next, recall that `length()` returns the number of elements in a vector. Therefore, divide `total` by `length(args)` to calculate the average. Store the result in `avg`.

```
average <- function(args) {  
  # Add the numbers, store in total  
  total <- sum(args)  
  
  # Divide total number the number of elements in the vector  
  # Store in variable called avg  
  avg <- total/length(args)  
  return the result  
}
```

Finally, use the `return()` function to return the result of `average()` for use elsewhere in your code.

```
average <- function(args) {  
  # Add the numbers, store in total  
  total <- sum(args)  
  
  # Divide total number the number of elements in the vector  
  # Store in variable called avg  
  avg <- total/length(args)  
  
  # Return the result.  
  return(avg)  
}
```

Let's test the function, first with 5, 11, and 19, then with a vector of sepal lengths from the iris data set. We'll also use the built-in `mean()` function on the iris sepal lengths to compare with our results.

Notice that we use our `average()` function just as we do the `mean()` or any other function.

```
# Test with three numbers
average(c(5, 11, 19))

# Test with a vector of sepal lengths from the iris data frame.
average(iris$Sepal.Length)

# To be sure it works, use the built-in mean() function on iris sepal lengths.
mean(iris$Sepal.Length)

## [1] 11.66667
## [1] 5.843333
## [1] 5.843333
```

It works!

Enhancing your function.

The framework above is the minimum you need to write a function. If you're writing a custom function for a singular need, this may be all you need to do. But, let's go one step further to make a point.

Consider the vector `c(5, 11, NA, 19)`. It's the same vector as above but an NA has been inserted. Recall that NA is how R represents missing data. What happens if you pass this vector to your `average()` function?

```
numbers <- c(5, 11, NA, 19)
average(numbers)
```

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

The result is NA. You get the same result with `mean(numbers)`. R does not know how you might want to handle NA so it returns NA as a result. If you look at the help for `mean()` (`?mean`), you will see this usage.

```
Usage
mean(x, ...)
```

```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Notice the `na.rm = FALSE` argument. `na.rm` is a boolean value that when TRUE causes all instances of NA to be removed from a vector. The default setting is FALSE so NAs are not removed by default. You must specifically override the default with `na.rm = TRUE` to remove all NA elements from your vector. (Many R functions have the `na.rm = FALSE` argument.)

```
numbers <- c(5, 11, NA, 19)

# Default setting of na.rm = FALSE
mean(numbers) # Returns NA

# Override the default with na.rm = TRUE
mean(numbers, na.rm = TRUE) # Returns the mean
```

```
## [1] NA
## [1] 11.66667
```


Let's make two changes to our average function. First, you may have noticed from the help that the `mean()` function has several arguments. Our updated `average()` function will have two arguments, so `args` doesn't make sense as a name for *one* of the arguments. Change all instances of `args` to `values` to show that the first argument is a vector of values.

```
# Comments removed for brevity. Your code must have comments.
average <- function(values) {
  total <- sum(values)
  avg <- total/length(values)
  return(avg)
}
```

Easy enough. Use argument names that help you understand what the arguments are for. Next, let's add the `na.rm = FALSE` argument.

```
average <- function(values, na.rm = FALSE) {
  total <- sum(values)
  avg <- total/length(values)
  return(avg)
}
```

Next, we have to add code that tests whether `na.rm` is `FALSE` or `TRUE`. We'll use the `ifelse()` function described in the notes above. We'll also use another function called `na.omit()` that removes all instances of `NA` from a vector.

If `na.rm = TRUE`, then our code will pass the `values` argument to `na.omit()` to remove all instances of `NA`. If `na.rm = FALSE`, then the vector passes untouched. In pseudocode, the `ifelse()` function looks like this.

```
ifelse(na.rm is TRUE,                                # Test
      use na.omit to remove NAs from the vector,    # Yes
      leave the vector alone)                         # No
```

In real code, the line would be

```
ifelse(na.rm, # TRUE (yes) or FALSE (no)?
      values <- na.omit(values), # TRUE. Remove NAs.
      values)                    # FALSE. Leave vector alone.
```

Notice that you do not have to ask if `na.rm == TRUE` or `na.rm == FALSE`. The function will look at the value of `na.rm` and know whether it is `TRUE` or `FALSE`. If `na.rm` is `FALSE` (our default), then the `FALSE` argument will run. If `na.rm` is `TRUE`, then the `TRUE` argument will run.

Our final function looks like this.

```
average <- function(values, na.rm = FALSE) {
  ifelse (na.rm,                                # Test
        values <- na.omit(values), # Is TRUE
        values)                               # IS FALSE
  total <- sum(values)
  avg <- total/length(values)
  return(avg)
}
```

Now, let's make sure it works.

```
numbers <- c(5, 11, NA, 19)
average(numbers) # na.rm = FALSE is the default
```

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

```
average(numbers, na.rm = TRUE) # override the default
```

```
## [1] 11.66667
```

Your updated `average()` function is more robust than your first version because it does some basic *error checking* to ensure your data do not include NA among the elements. Other errors are still possible. What if you tried to pass data to your function that is *not* a vector of numbers?

```
average(c())
```

```
## Warning in rep(no, length.out = len): 'x' is NULL so the result will be NULL
```

```
## Error in ans[npos] <- rep(no, length.out = len)[npos]: replacement has length zero
```

```
average(NULL)
```

```
## Warning in rep(no, length.out = len): 'x' is NULL so the result will be NULL
```

```
## Error in ans[npos] <- rep(no, length.out = len)[npos]: replacement has length zero
```

```
average(c("a", "b", "c"))
```

```
## Error in sum(values): invalid 'type' (character) of argument
```

Well-written functions, even for your own use, include at least minimal error checking to ensure reliability of your results. For now though, the basic framework will serve our needs.

Review the notes above to see where you should place custom functions in your code.

Want to write another function?

You're going to have to write two functions in the homework assignment. One of them will be for the standard error of the mean, which has the following formula

$$\frac{s}{\sqrt{n}},$$

where s is the standard deviation of the sample and n is the sample size. Use `sd()` to calculate standard deviation of a vector, `length()` to calculate sample size, and `sqrt()` to calculate the square root of the sample size.

Note: You do not have to turn in this function as a separate assignment. You'll include the function as part of your homework assignment so I will see it then. But, writing it now lets you gauge your understanding now.

Another note: When you run the code that contains the function, you should not get any visible feedback. R loads your function into memory so that it is available when you call it later. If you get an error, then something is wrong with the structure of your code. You'll have to fix it until it loads without error.

- A function that loads correctly is not guaranteed to return the correct result. That's a separate issue.

Yet another note: Surprisingly, base R does not have a built-in function to calculate standard error of the mean.