Engle Lab useRguide

Jason S. Tsukahara

2023-05-20

Table of contents

Welcome

This is the useRguide for the Attention & Working Memory Lab at Georgia Tech. This guide provides training in the basics of how to use R, the tidyverse, and tools for how we process and analyze data in our lab. The workflow and data processing steps presented in this guide are perculiar to the type of data we tend to work with in our lab. We primarily collect behavioral data on a large set of cognitive tasks to test individual differences in cognitive ability. However, you will find that most of the principles presented here apply to working with other types of data. This is especially true of the sections on data manipulation and data visualization.

I have developed several R packages that were designed to work with the type of data and analyses we use in the lab. Most of these packages contain R script templates and Quarto documents to make it easier for you to create new R scripts and analysis documents.

Part I

Getting Started

1 R and RStudio

R is the actual programming software that executes the code. RStudio is an environment for writing R scripts and, Quarto, and R Markdown documents, managing packages and projects, and much more. When you open RStudio, it will open a window to the R console within the RStudio environment.

If you already have R and RStudio installed move on to Updating R. If you have an R version older than 4.0.0 than you need to update R.

1.1 Install R

irst you need to download the latest version of **R** from their website https://www.r-project.org

1. Select **CRAN** on the left, just under **Download**
2. Select the first option under 0-Cloud
3. Select the download option depending on your computer
4. Select the **base** installation (for Windows) or the **Latest Release** (for Mac)
5. Open and Run the installation file

1.2 Install R Studio

The easiest way to interact with R is through the R Studio environment. To do this you need to download R Studio

1. Select the **Free** version of **R Studio Desktop**
2. Select the download option depending on your computer

1.3 The RStudio Environemnt

Go ahead an open the RStudio application on your computer.

When you open a fresh session of RStudio there are 3 window panes open. The Console window, the Environment window, and the Files window. Go ahead and navigate to File -> New File -> R Script. You should now see something similar to the image below

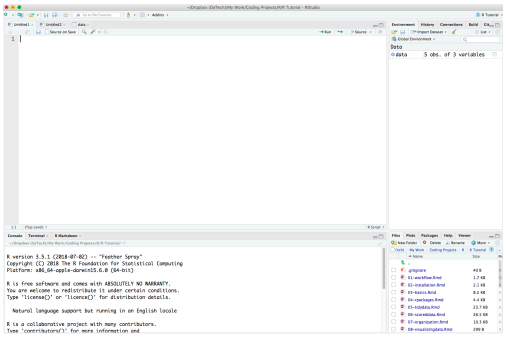


Figure 1.1: RStudio window panes

There are 4 window panes and each one has it's own set of tabs associated with it:

- The **Console** window (the bottom left window pane) is where code is executed and output is displayed.
- The **Source** window (the top left window pane) is where you will write your code to create a script file. When you open a new script file you will see a blank sheet where you can start writing the script. When you execute lines of code from here you will see it being executed in the Console window.

  The Source window is also where you can view data frames you have just imported or created. In the image above, notice the different tabs in the Source window. There are two "Untitled" script files open and one data frame called 'data'.
- The **Environment** window (top right window pane) is where you can see any data frames, variables, or functions you have created. Go ahead and type the following in your Console window and hit enter.

```
welcome_message <- "hello"
```

You should now see the object `welcome_message` in the Environment window pane

- The **Files** window (the bottom right window pane) is where you can see your computer's directories, plots you create, manage packages, and see help documentation.

1.4 RStudio Settings

I highly suggest changing the default RStudio General settings by going to Tools -> Global Options

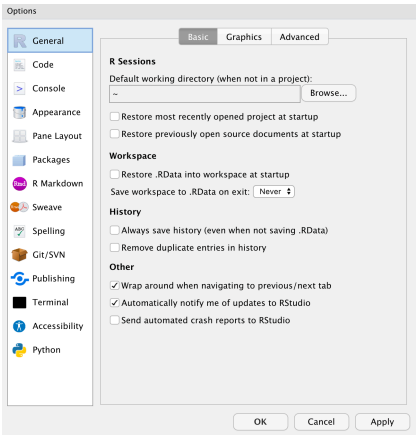


Figure 1.2: RStudio settings window

You can also change the theme, font type, and font size, if you navigate to the Appearance tab in Settings. You may also like to change the organization of the RStudio window panes in Pane Layout.

1.5 Update R

If you already have R installed, but want to update it to the most current version follow these steps.

Warning: When updating R (not RStudio), it may remove all packages you have installed

First check what version of R you have installed.

1. Open RStudio
2. In the console window you will see the R version you are running (e.g., R version 4.1.0)
3. If you have an R version older than 4.0.0 than you should update R.
4. Run the following lines of code in your console window. This is an easy way to re-install all your currently installed packages. This step will save a list of packages to re-install later.

```
# Save current packages and their versions to object called ip

ip <- installed.packages()
ip

# Save the object as an .rds file

saveRDS(ip, "CurrentPackages.rds")
```

5. Exit out of all R or RStudio windows
6. Download and install the latest version of R (see the section on installing R above)
7. Open RStudio
8. Check if your previously installed packages are installed using the **Packages** tab in the bottom right window
9. If you need to re-install your previous packages, then run the following lines of code

```
# After updating R, load the file and reinstall packages

ip <- readRDS("CurrentPackages.rds")

install.packages(ip[,1])
```

1.6 Update R Studio

Go to Help -> Check for Updates

2 Useful Tips

2.1 Helper Function

When working with functions, it can be difficult to remember the argument names and values you need to specify. However, there is a helper function that can make this process much easier: ?. By typing ?function_name() in the console, you can access the function's documentation and quickly figure out what arguments you need to provide. This can save you a lot of time and frustration, especially when working with complex functions.

```
?seq()
```

2.2 Generative AI

Generative AI can be a useful assistant to both learning and writing R code. It will make mistakes but that is what actually makes it a useful learning tool, it can also help you discover ways of doing things you wouldn't have thought of before.

Start out small, if you are not immediately sure how to proceed with writing R code for something you want to do then prompt an AI model to write some code and provide an explanation for you. Continuing prompting it and/or edit the code to suit your specific need.

You should also start using AI models to assist you in other areas of your work as well. Again, just start out small. Get in the habit and setup a workflow where an AI model is right at your fingertips, just a few clicks of the mouse or keyboard away.

2.3 Store Frequently Used Code For Reuse

If you find yourself using the same or similar sequences of code repeatedly, it can be incredibly helpful to have a central location where you can store your frequently used code and easily retrieve it at any time. While GitHub is a popular option for this, it requires learning a new system. Notion is a program I personally use and recommend, but a simple folder on your desktop with R scripts is also a viable option.

Without a central location for frequently used code, you may find yourself spending a significant amount of time and effort searching through previous projects to locate the code you need. This can be a daunting task, requiring a good memory and a lot of time. Having a singular place to go to for all your frequently used code can make this process much easier and save you time and energy in the long run.

2.4 Use Templates!

Creating your own templates and/or templates for your lab is highly recommended. This will save you a significant amount of time and effort, enabling you to start working with your data more quickly and set up new data analysis projects with ease. Additionally, consider creating an R package that include Quarto documents for analyses and reports. This will help streamline your workflow even further.

I have developed several R packages for the lab that contain useful templates and documents. Please make use of them.

2.5 Limit Number of Packages

When using R, it's recommended to limit the number of packages you use. You may be surprised at how much you can accomplish with just a few packages. Limiting your package usage makes it easier to manage your installed packages, and also helps with the learning curve, as you don't have to memorize functions from a large number of packages.

2.6 Ask a Friend

While there are many functions available for most tasks, finding the right one can be a challenge. Instead of spending time on long and convoluted solutions, consider asking friends or colleagues if they know of a package or function that could help you accomplish what you need. Collaborating with others is a great way to discover new functions and tools that you may not have known existed.

2.7 Google Search

This section may be less relevant now, given the rise of generative AI models. However, Google can still be a valuable tool for finding R solutions quickly. To get more targeted results, try including the name of the package or function you think might help you in your search phrase.

If you use `dplyr` frequently, for example, use `dplyr` in your search phrase to find solutions that are consistent with your preferred way of working.

If you're unsure of which function to use, try including the function name in your search phrase. You can also use Google to find more detailed documentation for specific packages or functions. However, it's best to avoid links that start with `https://cran.r-project.org` or `https://www.rdocumentation.org`, as these are usually just copies of the `?` help documentation.

Instead, look for links that include `https://github.com`. GitHub repos often include links to more extensive documentation, such as the GitHub repo for the popular `dplyr` package (https://github.com/tidyverse/dplyr), which has a link on the right side of the repo page to detailed documentation on all the functions in the package.

2.8 Explore Functions in a Package

Additionally, there may be functions in the packages you already use that you have yet to discover. Take some time to explore all the different functions within a package, particularly those that you use frequently. The GitHub repository for a package is a great resource for exploring all of its functions. To access it, simply type the name of the package followed by the term "GitHub" into a search engine.

3 R Packages and Functions

The community of R users have developed a vast number of functions that expand on the base R functions. Many of the functions developed by R users allow you to do more complicated things with your data without having to be an advanced R programmer. And the great thing is that as more psychology researchers use R, the more functions there are specifically for psychological research.

Functions that R users develop are collected in what are called **packages**. Most R packages are hosted on The Comprehensive R Archive Network - **CRAN**. Some other packages, ones that are in more developmental stages and may not be as stable, are hosted on GitHub.

3.1 Install and Load Packages

To install packages from **CRAN** is easy. Simply type into the console window: `install.packages("packagename")`

For example:

```
install.packages("dplyr")
```

Once you have a package installed, you can load that package into your current environment. Your **R Environment** consists of things such as objects (variables) you have created, data you have imported, and functions you have loaded. Your R Environment are like the tools and objects you have available to you and are working with.

When you load a package you are bringing the functions from that package into your **environment** so you can start using them. To load a package is easy: `library(package_name)`

For example:

```
library(dplyr)
```

3.2 Using Functions

Basically anything you do in R is by using **functions**. In fact, learning R is just learning what functions are available and how to use them.

Functions start with the name of the function followed by parentheses `function_name()`. Inside the `()` is where you specify certain arguments separated by commas `,` . Some arguments are optional and some are required for the function to work.

For example, there is a function to create a sequence of numbers, `seq()`.

```
seq(1, 100, by = 10)
```

```
 [1]  1 11 21 31 41 51 61 71 81 91
```

In the `seq()` function above we specified three arguments, separated by commas. The first two arguments were set without specifying the argument name, however the third argument we used the argument name `by` to define `seq(by = 10)`. If you don't explicitly use the argument name it will implicitly assume an argument **based on the order it is entered**, depending on how the author created the function.

The Helper Function

A handy tip is to frequently make use of the helper function, `?`. Type `?seq` into the R console. Helper documentation will be provided for that function and as you can see, the first argument defined is `from` and the second argument is `to`.

Order only matters if you do not specify argument names

Specifying the actual argument names, the above code is identical to the three following examples:

```
seq(from = 1, to = 100, by = 10)
seq(to = 100, by = 10, from = 1)
seq(1, 100, 10)
```

There are also default values that arguments take, which means if you don't define an argument it will take on the default value. The helper documentation shows that the `from` argument

has a default of `from = 1`, therefore we could even leave out the `from =` argument because we were using the default value:

```
seq(to = 100, by = 10)
```

What this means is that it can be important to know what the default values are for functions you are using and you can figure that out with the helper function `?`

4 R Basics

This chapter will cover some of the basics of using R.

As you go through any of the chapters in this guide, I encourage you to experiment. If you are curious what happens if you write the code slightly differently, if you do this or that, go ahead and try it out. See what happens. If it is not what you expected spend some time figuring out why.

If you have not done so already, open a new R script file. To create a new R script go to

File -> New File -> R Script

This should have opened a blank **Script** window called **Untitled**.

The **Script** window is a file where you are saving your code. This is where you will write, edit, delete, and re-write your code.

To follow along with the tutorial, you should type (for now, resist just copying and pasting) the lines of code I display in the tutorial into your script.

> **Note**
>
> Save your empty script somewhere on your computer

4.1 Creating R objects

In R, everything that exists is an object and everything you do to objects are functions. You can define an object using the assignment operator `<-`.

Everything on the left hand side of the `<-` assignment operator is an object. Everything on the right hand side of `<-` are functions or values. Go ahead and type the following two lines of code in your script

```
string <- "hello"
string
## [1] "hello"
```

You can execute/run a line of code by placing the cursor anywhere on the line and press `Ctrl + Enter`. Go ahead an run the two lines of code.

In this example, the first line creates a new object called `string` with a value of "hello". The second line simply prints the output of `string` to the **Console window**. In the second line there is no assignment operator. When there is no `<-` this means you are essentially just printing to the console. You can't do anything with stuff that is just printed to the console, it is just for viewing purposes.

For instance, if I wanted to calculate `1 + 2` I could do this by printing it to the console

```
1 + 2
## [1] 3
```

However, if I wanted to do something else with the result of that calculation then I would not be able to unless I assigned the result to an object using `<-`

```
result <- 1 + 2
result <- result * 5

result
## [1] 15
```

The point is, you are almost always going to assign the result of some function or value to an object. Printing to the console is not very useful. Almost every line of code, then, will have an object name on the left hand side of `<-` and a function or value on the right hand side of `<-`

In the first example above, notice how I included `" "` around hello. This tells R that hello is a string, not an object. If I were to not include `" "`, then R would think I am calling an object. And since there is no object with the name hello it will print an error

```
string <- hello
## Error in eval(expr, envir, enclos): object 'hello' not found
```

Do not use `" "` for Numerical values

```
a <- "5" + "1"
## Error in "5" + "1": non-numeric argument to binary operator
```

You can execute lines of code by:

1. Typing them directly into the Console window

2. Typing them into the Script window and then on that line of code pressing **Ctrl+Enter**. With **Ctrl+Enter** you can execute one line of your code at a time.
3. Clicking on **Source** at the top right of the Script window. This will run ALL the lines of code contained in the script file.

It is important to know that EVERYTHING in R is **case sensitive**.

```
a <- 5

a + 5
## [1] 10
A + 5
## Error in eval(expr, envir, enclos): object 'A' not found
```

4.2 Classes

Classes are types of values that exist in R:

- character `"hello"`, `"19"`
- numeric (or double) `2`, `32.55`
- integer `5`, `99`
- logical `TRUE`, `FALSE`

To evaluate the class of an object you can use the `typeof()`

```
typeof(a)
## [1] "double"
```

To change the class of values in an object you can use the function `as.character()` , `as.numeric()` , `as.double()` , `as.integer()` , `as.logical()` functions.

```
as.integer(a)
## [1] 5
```

```
as.character(a)
## [1] "5"
```

```
as.numeric("hello")
## Warning: NAs introduced by coercion
## [1] NA
```

4.3 Vectors

Okay so now I want to talk about creating more interesting objects than just `a <- 5`. If you are going to do anything in R it is important that you understand the different data types and data structures you can use in R. I will not cover all of them in this tutorial. For more information on data types and structures see this nice Introduction to R

Vectors contain elements of data. The length of a vector is the number of elements in the vector. For instance, the variable `a` we created earlier is actually a vector of length 1. It contains one element with a value of 5. Now let's create a vector with more than one element.

```
b <- c(1,3,5)
```

`c()` is a function. Functions contain arguments that are inputs for the function. Arguments are separated by commas. In this example the `c()` function concatenates the arguments (`1, 3, 5`) into a vector. We are passing the result of this function to the object b. What do you think the output of b will look like?

```
b
## [1] 1 3 5
```

You can see that we now have a vector that contains 3 elements; `1, 3, 5`. If you want to reference the value of specific elements of a vector you use brackets `[ ]`. For instance,

```
b[2]
## [1] 3
```

The value of the second element in vector `b` is 3. Let's say we want to grab only the 2nd and 3rd elements. We can do this at least two different ways.

```
b[2:3]
## [1] 3 5
b[-1]
## [1] 3 5
```

Now, it is important to note that we have not been changing vector `b`. If we display the output of `b`, we can see that it still contains the 3 elements.

```
b
## [1] 1 3 5
```

To change vector `b` we need to define `b` as vector `b` with the first element removed

```
b <- b[-1]
b
## [1] 3 5
```

Vector `b` no longer contains 3 elements. Now, let's say we want to add an element to vector `b`.

```
c(5,b)
## [1] 5 3 5
```

Here the `c()` function created a vector with the value 5 as the first element followed by the values in vector `b`

Or we can use the variable `a` that has a value of 5. Let's add this to vector `b`

```
b <- c(a,b)
b
## [1] 5 3 5
```

What if you want to create a long vector with many elements? If there is a pattern to the sequence of elements in the vector then you can create the vector using `seq()`

```
seq(0, 1000, by = 4)
##   [1]   0   4   8  12  16  20  24  28  32  36  40  44  48  52
##  [15]  56  60  64  68  72  76  80  84  88  92  96 100 104 108
##  [29] 112 116 120 124 128 132 136 140 144 148 152 156 160 164
##  [43] 168 172 176 180 184 188 192 196 200 204 208 212 216 220
##  [57] 224 228 232 236 240 244 248 252 256 260 264 268 272 276
##  [71] 280 284 288 292 296 300 304 308 312 316 320 324 328 332
##  [85] 336 340 344 348 352 356 360 364 368 372 376 380 384 388
##  [99] 392 396 400 404 408 412 416 420 424 428 432 436 440 444
## [113] 448 452 456 460 464 468 472 476 480 484 488 492 496 500
## [127] 504 508 512 516 520 524 528 532 536 540 544 548 552 556
## [141] 560 564 568 572 576 580 584 588 592 596 600 604 608 612
```

```
## [155]  616  620  624  628  632  636  640  644  648  652  656  660  664  668
## [169]  672  676  680  684  688  692  696  700  704  708  712  716  720  724
## [183]  728  732  736  740  744  748  752  756  760  764  768  772  776  780
## [197]  784  788  792  796  800  804  808  812  816  820  824  828  832  836
## [211]  840  844  848  852  856  860  864  868  872  876  880  884  888  892
## [225]  896  900  904  908  912  916  920  924  928  932  936  940  944  948
## [239]  952  956  960  964  968  972  976  980  984  988  992  996 1000
```

Vectors can only contain elements of the same "class".

```
d <- c(1, "2", 5, 9)
d
## [1] "1" "2" "5" "9"
```

```
as.numeric(d)
## [1] 1 2 5 9
```

4.4 Factors

Factors are special types of vectors that can represent categorical data. You can change a vector into a factor object using `factor()`

```
factor(c("male", "female", "male", "male", "female"))
## [1] male   female male   male   female
## Levels: female male
```

```
factor(c("high", "low", "medium", "high", "low"))
## [1] high   low    medium high   low
## Levels: high low medium
```

```
f <- factor(c("high", "low", "medium", "high", "low"),
            levels = c("high", "medium", "low"))
f
## [1] high   low    medium high   low
## Levels: high medium low
```

4.5 Lists

Lists are containers of objects. Unlike Vectors, Lists can hold different classes of objects.

```
list(1, "2", 2, 4, 9, "hello")
## [[1]]
## [1] 1
##
## [[2]]
## [1] "2"
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 9
##
## [[6]]
## [1] "hello"
```

You might have noticed that there are not only single brackets, but double brackets `[[ ]]`

This is because Lists can hold not only single elements but can hold vectors, factors, lists, data frames, and pretty much any kind of object.

```
l <- list(c(1,2,3,4), "2", "hello", c("a", "b", "c"))
l
## [[1]]
## [1] 1 2 3 4
##
## [[2]]
## [1] "2"
##
## [[3]]
## [1] "hello"
##
## [[4]]
## [1] "a" "b" "c"
```

You can see that the length of each element in a list does not have to be the same. To reference the elements in a list you need to use the double brackets `[[ ]]`.

```
l[[1]]
## [1] 1 2 3 4
```

To reference elements within list elements you use double brackets followed by a single bracket

```
l[[4]][2]
## [1] "b"
```

You can even give names to the list elements

```
person <- list(name = "Jason",
               phone = "123-456-7890",
               age = 23,
               favorite_colors = c("blue", "red", "brown"))
person
## $name
## [1] "Jason"
##
## $phone
## [1] "123-456-7890"
##
## $age
## [1] 23
##
## $favorite_colors
## [1] "blue"  "red"   "brown"
```

And you can use the names to reference elements in a list

```
person[["name"]]
## [1] "Jason"
person[["favorite_colors"]][3]
## [1] "brown"
```

4.6 Data Frames

You are probably already familiar with data frames. SPSS and Excel uses this type of structure. It is just rows and columns of data. A data table! This is the format that is used to perform statistical analyses on.

So let's create a data frame so you can see what one looks like in RStudio

```
data <- data.frame(id = 1:10,
                   x = c("a", "b"),
                   y = seq(10,100, by = 10))
data
##    id x   y
## 1   1 a  10
## 2   2 b  20
## 3   3 a  30
## 4   4 b  40
## 5   5 a  50
## 6   6 b  60
## 7   7 a  70
## 8   8 b  80
## 9   9 a  90
## 10 10 b 100
```

You can view the Data Frame by clicking on the object in the **Environment** window or by executing the command `View(data)`

Notice that it created three columns labeled `id`, `x`, and `y`. Also notice that since we only specified a vector of length 2 for `x` this column is coerced into 10 rows of repeating "a" and "b". All columns in a data frame must have the same number of rows.

You can use the `$` notation to reference just one of the columns in the data frame

```
data$y
##  [1]  10  20  30  40  50  60  70  80  90 100
```

Alternatively you can use

```
data["y"]
##      y
## 1   10
## 2   20
## 3   30
```

```
## 4   40
## 5   50
## 6   60
## 7   70
## 8   80
## 9   90
## 10 100
```

To reference only certain rows within a column

```
data$y[1:5]
## [1] 10 20 30 40 50
data[1:5,"y"]
## [1] 10 20 30 40 50
```

4.7 If...then Statements

If...then statements are useful for when you need to execute code only if a certain statement is TRUE. For instance,...

First we need to know how to perform logical operations in R

Okay, we have this variable `a`

```
a <- 5
```

Now let's say we want to determine if the value of `a` is greater than 3

```
a > 3
## [1] TRUE
```

You can see that the output of this statement `a > 3` is `TRUE`

Now let's write an if...then statement. If `a` is greater than 3, then multiply `a` by 2.

```
if (a > 3) {
  a <- a*2
}
a
## [1] 10
```

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

Figure 4.1: List of logical operators in R

The expression that is being tested is contained in parentheses, right after the `if` statement. If this expression is evaluated as `TRUE` then it will perform the next line(s) of code.

The `{` is just a way of encasing multiple lines of code within one if statement. The lines of code then need to be closed of with `}`. In this case, since we only had one line of code `b <- a*2` we could have just written it as.

```
a <- 5
if (a > 3) a <- a*2
a
## [1] 10
```

What if we want to do something to `a` if `a` is NOT greater than 3? In other words… if `a` is greater than 3, then multiple `a` by 2 else set `a` to `missing`

```
a <- 5
if (a > 3) {
  a <- a*2
} else {
  a <- NA
}
a
## [1] 10
```

You can keep on chaining if…then… else… if… then statements together.

```
a <- 5
if (is.na(a)) {
  print("Missing Value")
} else if (a < 0) {
  print("A is less than 0")
} else if (a > 3) {
  print("A is greater than 3")
}
## [1] "A is greater than 3"
```

4.8 R Packages

R comes with a basic set of functions. All the functions we have used so far are part of the R basic functions. But when you want to start doing more complex operations it would be nice to have more complex functions. This is where R Packages come in…

An R Package is simply a collection of functions - that usually have some common theme to them. **Now the most wonderful thing about R is that other R users have developed tons of packages with functions they created themselves**. For instance, a group of users have developed an R package called `lavaan` that makes it extremely easy to conduct SEM in R.

4.8.1 Installing and Loading R Packages

R packages are easy to install and load. You just need to know the name of the package.

```
install.packages("name_of_package")
```

or for multiple packages at once

```
install.packages(c("package1", "package2", "package3"))
```

Installing the package does not mean you can start using the functions. To be able to use the function you need to then load the package library of functions as such

```
library(name_of_package)
```

When loading packages you do not have to in-case the package name in `" "`

4.9 More Resources

For additional tips in the basics of coding R see:

https://r4ds.hadley.nz

https://ramnathv.github.io/pycon2014-r/visualize/README.html

https://www.datacamp.com/courses/free-introduction-to-r/?tap_a=5644-dce66f&tap_s=10907-287229

http://compcogscisydney.org/psyr/

5 R Intermediate

This chapter will cover more intermediate R programming, such as for loops, and functions.

> **Note**
>
> Save your empty script somewhere on your computer

5.1 For Loops

For loops allow you iterate the same line of code over multiple instances.

Let's say we have a vector of numerical values

```
c <- c(1,6,3,8,2,9)
c
## [1] 1 6 3 8 2 9
```

and want perform an if...then operation on each of the elements. Let's use the same if...then statement we used above. If the element is greater than 3, then multiply it by 2 - else set it to missing. Let's put the results of this if...then statement into a new vector `d`

What we need to do is **loop** this if...then statement **for** each element in `c`

We can start out by writing the for loop statement

```
for (i in seq_along(c)) {

}
```

This is how it works. The statement inside of parentheses after `for` contains two statements separated by `in`. The first statement is the variable that is going to change it's value over each iteration of the `loop`. **You can name this whatever you want**. In this case I chose the label `i`. The second statement defines all the values that will be used at each iteration. The second statement will always be a vector. In this case the vector is `seq_along(c)`.

`seq_along()` is a function that creates a vector that contains a sequence of numbers from 1 to the length of the object. In this case the object is the vector `c`, which has a length of `6` elements. Therefore `seq_along(c)`, creates a vector containing 1, 2, 3, 4, 5, 6.

The `for loop` will start with `i` defined as `1`, then on the next iteration the value of `i` will be `2` … and so until the last element of `seq_along(c)`, which is `6`. We can see how this is working by printing 'i' on each iteration.

```
for (i in seq_along(c)) {
 print(i)
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

You can see how on each iteration it prints the values of `seq_along(c)` from the first element to the last element.

What we will want to do is, on each iteration of the for loop, access the `i`th element of the vector `c`.

Recall, you can access the element in a vector with `[ ]`, for instance `c[1]`. Let's print each `i`th element of `c`.

```
for (i in seq_along(c)) {
 print(c[i])
}
## [1] 1
## [1] 6
## [1] 3
## [1] 8
## [1] 2
## [1] 9
```

Now instead of printing `i` the for loop is printing each element of vector `c`.

Let's use the same if…then statement as above

```
a <- 5
if (a > 3) {
  a <- a*2
```

```
} else {
  a <- NA
}
a
## [1] 10
```

But instead we need to replace a with `c[i]`

For now let's just `print()` the output of the if... then statement.

```
for (i in seq_along(c)) {
  if (c[i] > 3) {
    print(c[i]*2)
  } else {
    print(NA)
  }
}
## [1] NA
## [1] 12
## [1] NA
## [1] 16
## [1] NA
## [1] 18
```

Now for each element in `c`, if it is is greater than 3, then multiply it by 2 - else set as missing value. You can see that on each iteration the output is either the `i`th element of c multiplied by 2 or `NA`.

But just printing things to the console is useless! Let's overwright the old values in `c` with the new values.

```
for (i in seq_along(c)) {
  if (c[i] > 3) {
    c[i] <- c[i]*2
  } else {
    c[i] <- NA
  }
}
```

But what if we want to preserve the original vector `c`? Well we need to put it into a new vector, let's call it vector `d`. This get's a little more complicated but is something you might find yourself doing fairly often so it is good to understand how this works.

But if you are goind to do this to a "new" vector that is not yet created you will run into an error.

```
c <- c(1,6,3,8,2,9)
for (i in seq_along(c)) {
  if (c[i] > 3) {
    d[i] <- c[i]*2
  } else {
    d[i] <- NA
  }
}
## Error in eval(expr, envir, enclos): object 'd' not found
```

You first need to create vector `d` - in this case we can create an empty vector.

```
d <- c()
```

So the logic of our for loop, if...then statement is such that; on the `i`th iteration - if `c[i]` is greater than 3, then set `d[i]` to `c[i]*2` - else set `d[i]` to `NA`.

```
c <- c(1,6,3,8,2,9)
d <- c()
for (i in seq_along(c)) {
  if (c[i] > 3) {
    d[i] <- c[i]*2
  } else {
    d[i] <- NA
  }
}

c
## [1] 1 6 3 8 2 9
d
## [1] NA 12 NA 16 NA 18
```

Yay! Good job.

5.2 Functions

Basically anything you do in R is by using **functions**. In fact, learning R is just learning what functions are available and how to use them. Not much more to it than that.

You have only seen a couple of functions at this point. In this chapter, a common function used was `c()`. This function simply concatenates a series of numerical or string values into a vector. `c(1,6,3,7)`.

Functions start with the name of the function followed by parentheses `function_name()`. Inside the `()` is where you specify certain arguments separated by commas `,` . Some arguments are optional and some are required for the function to work.

For example, another function you saw last chapter was `data.frame()`. This function creates a data frame with the columns specified by arguments.

```
data.frame(id = 1:10, x = c("a", "b"), y = seq(10,100, by = 10))
##    id x   y
## 1   1 a  10
## 2   2 b  20
## 3   3 a  30
## 4   4 b  40
## 5   5 a  50
## 6   6 b  60
## 7   7 a  70
## 8   8 b  80
## 9   9 a  90
## 10 10 b 100
```

The arguments `id`, `x`, and `y` form the columns in the data frame. These arguments themselves used functions. For instance `y` used the function `seq()`. This function creates a sequence of numbers in a certain range at a given interval. Sometimes arguments are not defined by an `=`. The first two arguments in in `seq()` specify the range of 10 to 100. The third argument `by` specified the interval to be 10. So `seq(10, 100, by = 10)` creates a sequence of numbers ranging from 10 to 100 in intervals of 10.

```
seq(10, 100, by = 10)
##  [1]  10  20  30  40  50  60  70  80  90 100
```

In the `seq()` function the `by` argument is not required. This is because there is a default `by` value of 1.

```
seq(10, 100)
##  [1]  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
## [19]  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45
## [37]  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
## [55]  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81
## [73]  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
```

```
## [91] 100
```

Obviously if you want to specify a different interval, then you will need to specify `by =`.

5.3 Creating Your Own Functions

This section is optional. It will go over how to create your own functions.

Even if you do not want to get too proficient in R, it can be a good idea to know how to create your own function. It also helps you better understand how functions actually work.

We are going to create a function that calculates an average of values.

To define a function you use the `function()` and assign the output of `function()` to an object, which becomes the name of the function. For instance,

```
function_name <- function() {

}
```

This is a blank function so it is useless. Before we put stuff inside of a function let's work out the steps to calculate an average.

Let's say we have an array `a` that has 10 elements

```
a <- c(1,7,4,3,8,8,7,9,2,4)
a
##  [1] 1 7 4 3 8 8 7 9 2 4
```

To calculate an average we want to take the sum of all the values in `a` and divide it by the number of elements in `a`. To do this we can use the `sum()` and `length()` functions.

```
sum(a)
## [1] 53
length(a)
## [1] 10

sum(a)/length(a)
## [1] 5.3
```

Easy! So now we can just put this into a function.

```r
average <- function(x) {
  avg <- sum(x)/length(x)
  return(avg)
}
```

When creating a function, you need to specify what input arguments the function is able to take. Here were are specifying the argument `x`. You can use whatever letter or string of letters you want, but a common notation is to use `x` for the object that is going to be evaluated by the function. Then, inside the function we use the same letter `x` to calculate the `sum()` and `length()` of `x`. What this means is that **Arguments specified in a function become objects (or variables) passed inside the function**

You can create new objects inside a function. For instance we are creating an object, `avg`. However, these objects are created only inside the environment of the function. You cannot use those objects outside the function and they will not appear in your Environment window. To pass the value of an object outside of the function, you need to specify what you want to `return()` or what is the outpute of the function. In this case it is the object `avg` that we created inside the function.

Let's see the function in action

```r
average(a)
## [1] 5.3
```

Cool! You created your first function. Becuase the function only takes one argument `x` it knows that whatever object we specify in `average()` is the object we want to evaluate.

But what if our vector contains missing values?

```r
b <- c(1,NA,4,2,7,NA,8,4,9,3)
average(b)
## [1] NA
```

Uh oh. Here the vector `b` contains two missing values and the function `average(b)` returns `NA`. This is because in our function we use the function `sum()` without specifying to ignore missing values. If you type in the console `?sum` you will see that there is an argument to specify whether missing values should be removed or not. The default value of this argument is `FALSE` so if we want to remove the missing values we need to specify `na.rm = TRUE`.

It is a good idea to make your functions as flexible as possible. Allow the user to decide what they want to happen. For instance, it might be the case that the user wants a value of `NA` returned when a vector contains missing values. So we can add an argument to our `average()` function that allows the user to decide what they want to happen; ignore missing values or return `NA` if missing values are present.

Let's label this argument `na.ignore`. We could label it `na.rm` like the `sum()` function but for the sake of this Tutorial I want you to learn that you can label these arguments however you want, it is arbitrary. The label should make sense however.

Before we write the function let's think about what we need to change inside the function. Basically we want our new argument `na.ignore` to change the value of `na.rm` in the `sum()` function. If `na.ignore` is `TRUE` then we want `na.rm = TRUE`. Remember that arguments become objects inside of a function. So we will want to change:

```
avg <- sum(x)/length(x)
```

to

```
avg <- sum(x, na.rm = na.ignore)/length(x)
```

Let's try this out on our vector `b`

```
na.ignore <- TRUE
sum(b, na.rm = na.ignore)/length(b)
## [1] 3.8
```

We can test if our average function is calculating this correctly by using the actual base R function `mean()`.

```
mean(b, na.rm = TRUE)
## [1] 4.75
```

Uh oh. We are getting different values. This is because `length()` is also not ignoring missing values. The length of `b`, is 10. The length of `b` ignoring missing values is 8.

Unfortunately, `length()` does not have an argument to specify we want to ignore missing values. How we can tell `length()` to ignore missing values is by

```
length(b[!is.na(b)])
## [1] 8
```

This is saying, evaluate the length of elements in `b` that are not missing.

Now we can modify our function with

```
na.ignore <- TRUE
sum(b, na.rm = na.ignore)/length(b[!is.na(b)])
## [1] 4.75
```

to get

```
average <- function(x, na.ignore = FALSE) {
  avg <- sum(x, na.rm = na.ignore)/length(x[!is.na(x)])
  return(avg)
}
```

```
average(b, na.ignore = TRUE)
## [1] 4.75
```

```
mean(b, na.rm = TRUE)
## [1] 4.75
```

Walla! You did it. You created a function. Notice that we set the default value of `na.ignore` to `FALSE`. If we had set it as `TRUE` then we would not need to specify `average(na.ignore = TRUE)` since `TRUE` would have been the default.

When using functions it is important to know what the default values are

Both for loops and functions allow you to write more concise and readable code. If you are copying and pasting the same lines of code with only small modification, you can probably write those lines of code in a for loop or a function.

6 Quarto Basics

Quarto documents are a versitle way to create documents of different kinds and output formats. It can be used to generate statistical reports, presentations, websites, books, and interactive documents. This guide was created using Quarto Books.

6.1 R Scripts vs. Quarto

Using R scripts makes sense when the purpose is to output a data file. Once you have your script all setup, you don't necessarily care about what objects are created in the R environment or what is being printed to the console, etc. What you care about is the outputted data file. The R script was just a means to get there.

With data visualization and statistical analysis we care about generating figures, tables, and statistical reports - not data files. The process of data visualization and statistical analysis is also typically more explorative, iterative, and cyclic. Ultimately, we will want to generate a formatted, easy to read, final report of our analyses. All these things are not possible with R scripts.

An alternative to R scripts (.R) are Quarto (.qmd) and R Markdown (.Rmd) documents. The primary use of Quarto documents in the lab is to generate statistical reports in .html format.

> **Note**
>
> Quarto documents are the next generation of R Markdown documents, therefore, this guide will simply refer to Quarto documents from here on out.
> The basic structure of R Markdown and Quarto documents are the same. And most, though not all, features of R Markdown documents have been ported over to Quarto documents. And for the most part you can convert R Markdown documents to Quarto without any problems.

Follow this link for a brief Intro to Quarto Documents.

To open an Quartodocument go to

File -> New File -> Quarto Document...

Select HTML and click Create

An example Quarto document will open. We can use this to get familiar with the structure of Quarto documents.

By default, Quarto documents will open using the Visual Markdown Editor. This editor mode creates an easy to read formatted text that feels more similar to writing in a Microsoft Word document. This type of visual markdown text is called WYSIWG. It hides the markdown formatting and displays the text as it would be formatted in a document.

> **Note**
>
> You can switch to Source Editor mode if you want to see the actual raw non-formatted content of the document. Go ahead and try switching modes to see the difference.
> You will likely find some advantages to editing Quarto documents in source mode rather than Visual Editor mode.

There are three types of content that form the structure of a Quarto document.

- A YAML header
- R code chunks
- Markdown formatted text

6.2 YAML header

The YAML header contains metadata about how the document should be rendered and the output format. It is located at the very top of the document and is surrounded by lines of three dashes.

```
---
title: "Title of document"
output: html_document
---
```

There are various metadata options you can specify, such as if you want to include a table of contents. To learn about a few of them see https://quarto.org/docs/output-formats/html-basics.html

6.3 R code chunks

Unlike a typical R script file (.R), a Quarto document (.qmd) is a mixture of markdown text and R code chunks. Not everything in an R Markdown document is executed in the R console, only the R code chunks.

6.3.1 Create an R code chunk

R code chunks are enclosed with

#> “‘{r}

#> “‘

You can create R code chunks with the shortcut:

Mac: i (command + alt/opt + i)

Windows: i (ctrl + alt + i)

Or by going to Insert -> Code Cell -> R in the toolboar of the Quarto document (same section as Source / Visual).

Insert the followin code in the R code chunk

```
summary(cars)
```

Create another R code chunk with the following code

```
plot(pressure)
```

6.3.2 Execute R code chunk

To run chunks of R code you can click on the green “play” button on the top right of the R code chunk. Go ahead and do so.

You can see that the results of the R code are now displayed in the document.

6.4 Markdown text

The markdown text sections are **not the same as adding comments to lines of code**. You can write up descriptive reports, create bulletted or numbered lists, embed images or web links, create tables, and more.

The text is formatted using a language known as **Markdown**. Markdown is a convenient and flexible way to format text. When a Markdown document is rendered into some output (such as html or PDF), the text will be formatted as specified by Markdown syntax.

There are a lot of guides on how to use Markdown syntax. I will not cover this so you should check them out on your own. Here is one I reference often: **Markdown Cheatsheet**

6.5 Rendering a Quarto document

When you have finalized the content of a Quarto document you will then want to generate the document into the specified output format (.html in this case).

To render a Quarto document click on **Render** at the top. This will

1. Generate a live preview of the document in the Viewer pane
2. Create a .html file

You can now use the .html file to view your visualizations and statistical analyses and share or present them with the lab.

6.6 Visual Editor

See Visual Editing in RStudio for more details on using the visual editor including shortcuts and tips.

There are two convenient ways to insert formatted content in the visual editor:

1. Use the Insert tool in the toolbar at the top
2. Simply type `/` to quickly display the Insert Anything shortcut options. You can even start typing/searching what you want to insert.

Part II

Tidyverse

7 This Is The Way

The tidyverse is a set of packages that share an underlying design philosophy, grammar, and data structures. The most common packages in the tidyverse are; `readr` for importing and writing data files, `dplyr` and `tidyr` for manipulating and reformatting data, and `ggplot2` for data visualization.

Figure 7.1: graphic of some tidyverse packages

Although you will be learning R in this tutorial, it might be more appropriate to say that you are learning the tidyverse.

The tidyverse consists of packages that are simple and intuitive to use and will take you from importing data (with **readr**), to transforming and manipulating data structures (with **dplyr** and **tidyr**), and to data visualization (with **ggplot2**).

There is also a tidyverse style guide for writing R code.

There is some controversy in the R community about the use of the **tidyverse** way over the standard **base R** way of doing things. However, the tidyverse offers a more intuitive

and common language way of working with data in R; whereas, using base R feels more like programming and is harder to read, like a language you are not fully fluent in.

David provides a simple example of this (and there are a lot of other examples out there) on his blog post:

The pipe operator allows you to chain together a series of functions to conduct a series of manipulations on it. Conceptually, here's what code written using the tidyverse looks like

```
data %>%
  step1 %>%
  step2 %>%
  step3
```

We start with our data. Then we do step1. Then we do step2. Then we do step3. The pipe ties it all together, enabling us to do multiple things to our data, all in one execution of code.

Actual tidyverse code might look something like this:

```
data %>%
  filter(year == 2019) %>%
  summarise(mean_satisfaction_score =
              mean(satisfaction_rating)) %>%
  arrange(mean_satisfaction_score)
```

You can almost read this code in English.

1. Start with your data.
2. Filter to only include observations from 2019.
3. Calculate a mean satisfaction score from satisfaction ratings
4. Arrange our results by the mean satisfaction score.

Being able to read the code is one of the unique features of the Tidyverse. The Tidyverse manifesto prioritizes software that is designed "…primarily so that it is easy to use by humans. Computer efficiency is a secondary concern because the bottleneck in most data analysis is thinking time, not computing time."

The emphasis on human-readable function names (filter, group_by, summarize, arrange, etc.) is one part of this. The use of the pipe is another. Think of breaking code into multiple lines using the pipe as the equivalent of using spaces to make words easier to use. Youcanwriteasentencewithoutspacesbetweenwordsbutitsureisn'teasytoreadamirite?

Did you enjoy that? Then you'll especially enjoy reading base R code, which often looks like this:

```
mean(data$satisfaction_rating[data[year == 2019]])
```

And note, this does the same thing as the piped version above, minus the arrange step. Even with one fewer step, it's already much less readable.

7.1 The pipe operator

There are now two pipe operators. The original `dplyr` pipe operator is `%>%` and the newer base R pipe operator is `|>` . You can use either one but as a lab we will try to switch over to using the base R pipe.

The pipe operator is extremely useful and we use it extensively but needs some explanation.

There are different methods for writing code that performs multiple functions on the same object. For instance, all these examples use the tidyverse but only one of them (the last one) uses the pipe operator.

```
library(dplyr)
# create a sample dataframe
data <- data.frame(x = c(1, 2, 3), y = c("a", "b", "c"))

# three functions, filter, calculate a mean,
# then select only some columns to keep
data <- filter(data, y != "c")
data <- mutate(data, x_mean = mean(x))
data <- select(data, y, x_mean)
```

Another way of writing this code:

```
library(dplyr)
# create a sample dataframe
data <- data.frame(x = c(1, 2, 3), y = c("a", "b", "c"))

# three functions, filter, calculate a mean,
# then select only some columns to keep
data <- select(mutate(filter(data, y != "c"),
                      x_mean = mean(x)), y, x_mean)
```

Of these two methods, the first is preferable as it is easier to read. Another alternative is to use the pipe operator `|>`.

```
library(dplyr)
# create a sample dataframe
data <- data.frame(x = c(1, 2, 3), y = c("a", "b", "c"))

# three functions, filter, calculate a mean,
# then select only some columns to keep
data <- data |>
  filter(y != "c") |>
  mutate(x_mean = mean(x)) |>
  select(y, x_mean)
```

With the pipe operator, the result of the previous line gets passed (or piped) onto the next function. The first line in this example is simply specifying the data frame that is being passed from one line to the next. Notice how I did not have to specify `data` inside the `filter()`, `mutate()`, and `select()` functions. This makes the code more concise and easier to read. The end result of the last function, then gets assigned to the `data <-`.

If an error occurs somewhere in the pipe, an easy way to troubleshoot it is to remove the pipe operator at one of the lines one at a time to figure out where the error is located.

8 Import Data

Every R script that you write will require you to import (read in) a data file and output (write) a new data file.

In this Chapter you will learn various functions to import and output comma-separate value (csv), tab-delimited, SPSS, E-Prime -Export data files, and a batch of multiple data files.

For most of these data types we can use the **readr** package

Figure 8.1: readr logo

The `readr` package contains useful functions for importing and outputting data files.

Go ahead and install the `readr` package. In the console type:

```
install.packages("readr")
```

We will also use the `foreign` and `haven` packages for SPSS data files

```
install.packages("foreign")
install.packages("haven")
```

We will use some example data files for this chapter. Go ahead and download these files. You will have to unzip the file. For now just unzip it in your **downloads folder**. Inside the unzipped folder you will see a number of data files in different file formats.

> Tip
>
> Download example data files

8.1 CSV Files

`csv` files are by far the easiest files to import into R and most software programs. For this reason, I suggest any time you want to save/output a data file to your computer, do it in `csv` format.

8.1.1 Import .csv

We can import `csv` files using `read_csv()` from the `readr` package.

```
library(readr)
read_csv("filepath/datafile.csv")
```

You can see this is very simple. We just need to specify a file path to the data.

> Important
>
> DO NOT USE ABSOLUTE FILE PATHS!

I will talk more about file paths later but for now we will use `absolute` file paths, although it is highly suggested not to use them. This chapter is more about the different functions to import various types of data files.

First, figure out the `absolute` file path to your downloads folder (or wherever the unzipped data folder is located). On Windows the `absolute` file path will usually start from the `C:/` drive. On Macs, it starts from `~/`

Import the `Flanker_Scores.csv` file. You might have something that looks like

```
read_csv("~/Downloads/Flanker_Scores.csv")
```

However, this just printed the output of `read_csv()` to the console. To actually import this file into R, we need to assign it to an object in our `Environment`.

```
import_csv <- read_csv("~/Downloads/Flanker_Scores.csv")
```

You can name the object whatever you like. I named it `import_csv`.

To view the data frame

```
View(import_csv)
```

8.1.2 Output .csv

We can output a `csv` file using `write_csv()` from the `readr` package.

```
write_csv(object, "filepath/filename.csv")
```

Let's output the object `import_csv` to a `csv` file named: `new_Flanker_Scores.csv` to the downloads folder

```
write_csv(import_csv, "~/Downloads/new_Flanker_Scores.csv")
```

Note that whenever writing (outputting) a file to our computer there is no need to assign the output to an object.

8.2 Tab-Delimited

`tab-delimited` files are a little more tedious to import just because they require specifying more arguments. Which means you have to memorize more to import `tab-delimited` files.

8.2.1 Import .txt

To import a `tab-delimited` file we can use `read_delim()` from the `readr` package.

```
read_delim("filepath/filename.txt", delim = "\t",
           escape_double = FALSE, trim_ws = TRUE)
```

There are three additional arguments we have to specify: `delim`, `escape_double`, and `trim_ws`. The notation for `tab-delimted` files is `"\t"`.

Let's import the `Flanker_raw.txt` file

```
import_tab <- read_delim("~/Downloads/Flanker_raw.txt", "\t",
                         escape_double = FALSE, trim_ws = TRUE)
```

View the `import_tab` object

8.2.2 Output .txt

We can output a `tab-delimited` file using `write_delim()` from the `readr` package.

```
write_delim(object,
            path = "filepath/filename.txt", delim = "\t")
```

Output the `import_tab` object to a file named: `new_Flanker_raw.txt`

```
write_delim(import_tab,
            path = "~/Downloads/Flanker_raw.txt", delim = "\t")
```

8.3 SPSS

As horrible as it might sound, there might be occasions where we need to import an SPSS data file. And worse, we might need to output an SPSS data file!

I will suggest to use different packages for importing and outputing spss files.

8.3.1 Import .sav

To import an SPSS data file we can use `read.spss()` from the `foreign` package.

```
library(foreign)
read.spss("filepath/filename.sav",
          to.data.frame = TRUE, use.value.labels = TRUE)
```

The `use.value.labels` argument allows us to import the value labels from an SPSS file.

Import and View the `sav` file `CH9 Salary Ex04.sav`

```
import_sav <- read.spss("~/Downloads/CH9 Salary Ex04.sav")
```

8.3.2 Output .sav

To output an SPSS data file we can use `write_sav()` from the `haven` packge.

```
library(haven)
write_sav(object, "filepath/filename.sav")
```

Go ahead and output the `import_sav` object to a file: `new_CH9 Salary Ex04.sav`

```
write_sav(import_sav, "~/Downloads/new_CH9 Salary Ex04.sav")
```

8.4 RStudio Import GUI

The nice thing about R Studio is that there is also a GUI for importing data files.

When you are having difficulty importing a file correctly or unsure of the file format you can use the RStudio Import GUI.

In the **Environment** window click on **"Import Dataset"**. You will see several options available, these options all rely on different packages. Select whatever data type you want to import

You will see a data import window open up that looks like this

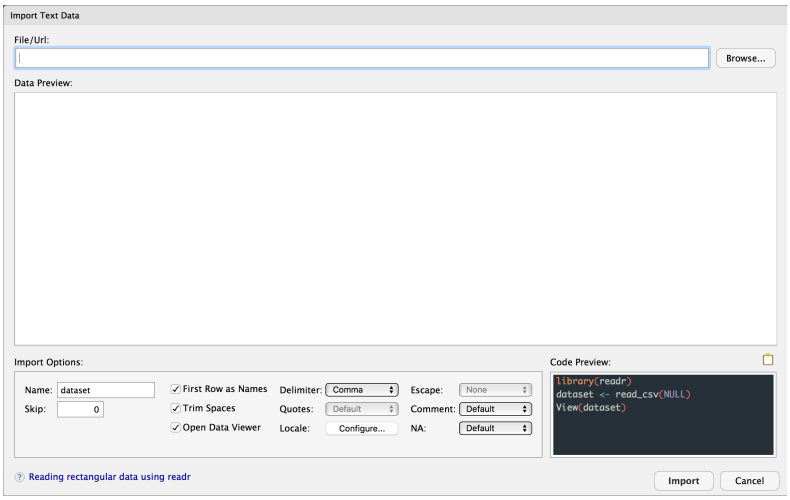


Figure 8.2: RStudio Import GUI

Select "Browse" on the top right and select the data file you want to import.

The "Data Preview" window will let you see if it is importing it in the right format. You can change the import options below this.

You might want to change the "Name" but you can always do this later in the R Script.

Make sure all the settings are correct by assessing the "Data Preview" window. Does the data frame look as you would expect it to?

Finally, copy and paste the code you need in the "Code Preview" box at the bottom right. You might not always need the `library(readr)` or `View(data)` lines.

Rather than selecting "Import" I suggest just closing out of the window and pasting the code into your R script.

`csv` files have a nice feature in that RStudio knows that these are file types we might want to import. So instead of navigating through the `Import Dataset` GUI we can just click on the file in the `Files` window pane.

8.5 E-Prime -Export

The E-Prime program we use to administer tasks has the option to output a .txt file. These file types are appended with **-Export** in the filename. These files are tab delimited **but also have a unique encoding** which makes them difficult to figure out how to import into R. After some trial and error I figured out they are encoded as **UCS-2LE**, thus they can be imported with the following settings in `read_delim()`

```
read_delim("data/folder", delim = "\t",
           escape_double = FALSE, trim_ws = TRUE, na = "NULL",
           locale = locale(encoding = "UCS-2LE"))
```

8.6 Multiple Files

You might find yourself in a situation where you need to import multiple data files and merge them into a single dataframe. For instance, with a batch of E-Prime -Export data files.

8.6.1 Bind

In R, a "bind" is combining data frames together by staking either the rows or columns. It is unlikely that we you will need to do a column bind so we can skip that. A row "bind" takes data frames that have the same columns but different rows. This will happen if you have separate data files for each subject from the same task. Each subject data file will have their unique rows (subject by trial level data) but they will all have the same columns.

To bind multiple files together requires only a couple of steps

1. Get a list of all the files

2. Import and bind the list of files using `purrr::map_df()`

   `map_df()` is a function from the `purrr` R package that allows you to apply a function to each element of a list or vector and then combine the results into a data frame. It is especially useful when you have a list of data frames that you want to combine into a single data frame. The function applies a function to each data frame in the list, and then combines the results into a single data frame. *This paragraph was written by Notion AI*

```
library(readr)
library(dplyr)
library(purrr)

# 1. Get a list of all files
files <- list.files("data/folder", pattern = "-Export",
                    full.names = TRUE)

# 2. Import and bind the list of files using purrr::map_df()
data_import <- files %>%
  map_df(read_delim, delim = "\t",
         escape_double = FALSE, trim_ws = TRUE, na = "NULL",
         locale = locale(encoding = "UCS-2LE"))
```

This example shows how to import multiple E-Prime -Export files into a single dataframe. But the same procedure, with different arguments inside of `purrr::map_df()`, can be used for any type of data file.

8.6.2 Join

In R, a "join" is merging data frames together that have at least some rows in common (e.g. Same Subject IDs) and have at least one column that is different. The rows that are common serve as the reference for how to "join" the data frames together.

To join multiple files together requires a few steps:

1. Get a list of all the files
2. Import the list of files using `purrr::map_df()`

   `map_df()` is a function from the `purrr` R package that allows you to apply a function to each element of a list or vector and then combine the results into a data frame. It is especially useful when you have a list of data frames that you want to combine into a single data frame. The function applies a function to each data frame in the list, and

then combines the results into a single data frame. *This paragraph was written by Notion AI*

3. Join each imported data file into a single data frame by a unique id using `purrr::reduce()` and `dplyr::full_join()`.

   The function reduce() from the purrr R package is used to iteratively apply a function to a list or vector and accumulate the results. The function takes two arguments: the list or vector to be processed, and the function to be applied to each element. The function is applied to the first element and the result is stored. The function is then applied to the stored result and the second element, and the result is stored. This process is repeated until all elements have been processed, and the final result is returned. This can be useful for operations like calculating the sum or product of a list of numbers, or concatenating a list of strings. *This paragraph was written by Notion AI*

```r
library(readr)
library(dplyr)
library(purrr)

# 1. Get a list of all files
files <- list.files("data/folder", pattern = "_Scores",
                    full.names = TRUE)

# 2 - 3. Import with lapply() and then join with plyr::join_all()
data_import <- files %>%
  map(read_csv) %>%
  reduce(full_join, by = "Subject")
```

This example shows how to import multiple scored data files (csv file type) and join all of them by ids in the "Subject" column using `purrr::reduce()` and `dplyr::full_join()`.

9 Working with Data

For this Chapter we will use an example data set from the *Flanker* task. This data set is a *tidy* raw data file for over 100 subjects on the *Flanker* task. There is one row per `Trial` per `Subject` and there is `RT` and `Accuracy` data on each `Trial`. Each `Trial` is either `congruent` or `incongruent`.

What we will want to do is calculate a `FlankerEffect` for each `Subject` so that we end up with one score for each `Subject`.

> Tip
>
> Download Example Date Set

Install the `dplyr` and `tidyr` packages

```
install.packages("dplyr")
install.packages("tidyr")
```

9.1 dplyr

The language of `dplyr` will be the underlying framework for how you will think about manipulating a `dataframe`.

Figure 9.1: dplyr logo

`dplyr` uses intuitive language that you are already familiar with. As with any R function, you can think of functions in the dplyr package as verbs that refer to performing a particular action on a data frame. The core dplyr functions are:

- `rename()` renames columns
- `filter()` filters rows based on their values in specified columns
- `select()` selects (or removes) columns
- `mutate()` creates new columns based on transformation from other columns, or edits values within existing columns
- `group_by()` splits data frame into separate groups based on specified columns
- `summarise()` aggregates across rows to create a summary statistic (means, standard deviations, etc.)

For more information on these functions Visit the dplyr webpage

For more detailed instructions on how to use the `dplyr` functions see the Data Transformation chapter in the popular R for Data Science book.

9.2 Stay within the Data Frame

Not only is the language of `dplyr` intuitive but it allows you to perform data manipulations all within the `dataframe` itself, without having to create external variables, lists, for loops, etc.

It can be tempting to hold information outside of a data frame but in general I suggest avoiding this strategy. Instead, hold the information in a new column within the `data frame` itself.

For example: A common strategy I see any many R scripts is to hold the mean or count of a column of values outside the `dataframe` and in a new variable in the Environment.

```
data <- data.frame(x = c(1,6,4,3,7,5,8,4), y = c(2,3,2,1,4,6,4,3))

y_mean <- mean(data$y)
```

This variable then could be used to subtract out the mean from the values in column `y`

```
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

data <- mutate(data,
               y_new = y - y_mean)

head(data)
##   x y  y_new
## 1 1 2 -1.125
## 2 6 3 -0.125
## 3 4 2 -1.125
## 4 3 1 -2.125
## 5 7 4  0.875
## 6 5 6  2.875
```

Although there is nothing wrong with this approach, in general, I would advise against this strategy. A better strategy is to do all this without leaving the data frame `data`.

```
library(dplyr)

data <- data.frame(x = c(1,6,4,3,7,5,8,4), y = c(2,3,2,1,4,6,4,3))
data <- mutate(data,
               y_mean = mean(y),
               y_new = y - y_mean)

head(data)
##   x y y_mean  y_new
## 1 1 2  3.125 -1.125
## 2 6 3  3.125 -0.125
## 3 4 2  3.125 -1.125
## 4 3 1  3.125 -2.125
## 5 7 4  3.125  0.875
## 6 5 6  3.125  2.875
```

It can be tempting to also think about writing for loops in your R script, but honestly for the most part for loops are avoidable thanks to a dplyr function called `group_by()`.

The only time I end up needing a for loop is when creating code to put into a function.

9.3 Setup R Script

9.3.1 Setup

At the top of your script load the three packages you will need for this Chapter

```
# Setup
library(readr)
library(dplyr)
library(tidyr)
```

Notice how I added a *commented* line at the top. Adding *comments* to your scripts is highly advisable, as it will help you understand your scripts when you come back to them after not working on them for a while.

9.3.2 Import

Import the data file you downloaded. Refer to Chapter 8 for how to do this.

```
import <- read_csv("data/tidyverse_example.csv")
```

It is always a good idea to get to know your `dataframe` before you start messing with it. What are the column names? What kind of values are stored in each column? How many observations are there? How many `Subjects`? How many `Trials`? etc.

What are the column names? use `colnames()` for a quick glance at the column names

```
colnames(import)
##  [1] "Subject"              "TrialProc"            "Trial"
##  [4] "Condition"            "RT"                   "ACC"
##  [7] "Response"             "TargetArrowDirection" "SessionDate"
## [10] "SessionTime"
```

To take a quick look at the first few rows of a `dataframe` use `head()`.

```
head(import)
## # A tibble: 6 x 10
##   Subject TrialProc Trial Condition      RT   ACC Response
##     <dbl> <chr>     <dbl> <chr>       <dbl> <dbl> <chr>
## 1   14000 practice      1 incongruent  1086     1 left
## 2   14000 practice      2 incongruent   863     1 left
```

```
## 3   14000 practice      3 congruent     488     1 right
## 4   14000 practice      4 incongruent   588     1 right
## 5   14000 practice      5 congruent     581     1 right
## 6   14000 practice      6 incongruent   544     1 right
## # i 3 more variables: TargetArrowDirection <chr>, SessionDate <chr>,
## #   SessionTime <time>
```

This gives you a good idea of what column names you will be working with and what kind of values they contain.

To evaluate what are all the unique values in a column you can use `unique()`. You can also use this in combination with `length()` to evaluate how many unique values are in a column.

```
unique(import$Condition)
```

```
[1] "incongruent" "congruent"
```

```
unique(import$Trial)
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
[163] 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
[181] 181 182 183 184 185 186 187 188 189 190 191 192
```

```
max(import$Trial)
```

```
[1] 192
```

```
length(unique(import$Subject))
```

```
[1] 410
```

```
unique(import$TrialProc)
```

```
[1] "practice" "real"
```

```
unique(import$ACC)
```

```
[1] 1 0
```

All these functions we just used from `colnames()` to `unique()` were to temporarily evaluate our data. They are not required to perform the actual data analysis. Therefore, I usually just type these in the console. A general rule of thumb is that if it is not required to be saved in your Script file then just type it in the console.

Okay let's take a look at how to use the `dplyr` functions to score this data.

9.4 rename()

We do not really need to, but let's go ahead and `rename()` a column. How about instead of `ACC` let's label it as `Accuracy`. Pretty simple

```
data <- rename(import, Accuracy = ACC)
```

`rename()` is really only useful if you are not also using `select()` or `mutate()`. In `select()` you can also rename columns as you select them to keep. This will be illustrated this later

Notice that I passed the output of this function to a new object `data`. I like to keep the object `import` as the original imported file and any changes will be passed onto a new data frame, such as `data`. This makes it easy to go back and see what the original data is. Because if we were to overwrite `import` then we would have to execute the `read_csv()` import function again to be able to see the original data file, just a little more tedious.

9.5 filter()

`filter()` is an inclusive filter and requires the use of *logical* statements. In **Chapter 2: Basic R** I talked a little bit about logical statements. Figure 4.1 shows a list of logical operators in R.

In addition to the logical operators, other functions can be used in `filter()`, such as:

- `is.na()` - include if missing
- `!is.na()` - include if not missing
- `between()` - values that are between a certain range of numbers
- `near()` - values that are near a certain value

We do not want to include practice trials when calculating the mean on RTs. We will use `filter()` to remove these rows. First let's evaluate the values in these columns

```
unique(import$TrialProc)
## [1] "practice" "real"
unique(import$Condition)
## [1] "incongruent" "congruent"
```

We can specify our `filter()` in a couple of different ways

```
data <- filter(data,
               TrialProc != "practice",
               Condition != "neutral")
```

or

```
data <- filter(import,
               TrialProc == "real",
               Condition == "congruent" | Condition == "incongruent")
```

Specifying multiple arguments separated by a comma `,` in `filter()` is equivalent to an `&` (`and`) statement.

In the second option, since there are two types of rows on `Condition` that we want to keep we need to specify `Condition ==` twice, separated by `|` (`or`). We want to keep rows where `Condition == "congruent" or Condition == "incongruent"`

Notice that the arguments have been separated on different lines. This is okay to do and makes it easier to read the code. Just make sure the end of the line still has a comma.

Go ahead and view `data`. Did it properly remove `practice` trials? How about `neutral` trials?

```
unique(data$TrialProc)
## [1] "real"
unique(data$Condition)
## [1] "incongruent" "congruent"
```

Again you should type these in the console NOT in the R Script!

There is a lot of consistency of how you specify arguments in the dplyr package.

1) You always first specify the data frame that the function is being performed on, followed by the arguments for that function.
2) Column names can be called just like regular R objects, that is without putting the column name in `" "` like you do with strings. If all you know is dplyr, then this might not seem like anything special but it is. Most non-tidyverse functions will require you to put `" "` around column names.

9.6 select()

`select()` allows you to select which columns to keep and/or remove.

Let's keep `Subject`, `Condition`, `RT`, `Trial`, and `Accuracy` and remove `TrialProc`, `TargetArrowDirection`, `SessionDate`, and `SessionTime`.

`select()` is actually quite versatile - you can remove columns by specifying certain patterns. I will only cover a couple here, but to learn more Visit the select() webpage

We could just simply select all the columns we want to keep

```
data <- select(data, Subject, Condition, RT, Trial, Accuracy)
```

alternatively we can specify which columns we want to remove by placing a `-` in front of the columns

```
data <- select(data, -TrialProc, -TargetArrowDirection,
               -SessionDate, -SessionTime)
```

or we can remove (or keep) columns based on a pattern. For instance `SessionDate` and `SessionTime` both start with `Session`

```
data <- select(data, -TrialProc, -TargetArrowDirection,
               -starts_with("Session"))
```

You might start realizing that there is always more than one way to perform the same operation. It is good to be aware of all the ways you can use a function because there might be certain scenarios where it is better or even required to use one method over another. In this example, you only need to know the most straightforward method of simply selecting which columns to keep.

You can also rename variables as you `select()` them... let's change `Accuracy` back to `ACC`... just because we are crazy!

```
data <- select(data, Subject, Condition, RT, Trial, ACC = Accuracy)
```

We are keeping `Subject`, `Condition`, `RT`, `Trial`, and renaming `ACC` to `Accuracy`.

9.7 mutate()

`mutate()` is a very powerful function. It basically allows you to do any computation or transformation on the values in the data frame. You can

1) change the values in already existing columns
2) create new columns based on transformation of other columns

9.7.1 Changing values in an existing column

Reaction times that are less than 200 milliseconds most likely do not reflect actual processing of the task. Therefore, it would be a good idea to not include these when calculating means.

What we are going to do is is set any RTs that are less than `200` milliseconds to missing, `NA`. First let's make sure we even have trials that are less than `200` milliseconds. Two ways to do this. 1) View the data frame and click on the `RT` column to sort by RT. You can see there are RTs that are as small as 1 millisecond! Oh my, that is definitely not a real reaction time. 2) you can just evaluate the minimum value in the RT column:

```
min(data$RT)
## [1] 0
```

Now lets `mutate()`

```r
data <- mutate(data, RT = ifelse(RT < 200, NA, RT))
```

Since we are replacing values in an already existing column we can just specify that column name, `RT =` followed by the transformation. Here we need to specify an if...then... else statement. To do so within the `mutate()` function we use the function called `ifelse()`.

`ifelse()` evaluates a logical statement specified in the first argument, `RT < 200`. `mutate()` works on a row-by-row basis. So for each row it will evaluate whether `RT` is less than 200. If this logical statement is `TRUE` then it will perform the next argument, in this case sets `RT = NA`. If the logical statement is `FALSE` then it will perform the last argument, in this case sets `RT = RT` (leaves the value unchanged).

9.7.2 Creating a new column

Let's say for whatever reason we want to calculate the difference between the RT on a trial minus the overall grand mean RT (for now, across all subjects and all trials). This is not necessary for what we want in the end but what the heck, let's be a little crazy. (I just need a good example to illustrate what `mutate()` can do.)

So first we will want to calculate a "grand" mean RT. We can use the `mean()` function to calculate a mean.

```r
mean(data$RT, na.rm = TRUE)
## [1] 529.1414
```

Since we replaced some of the `RT` values with `NA` we need to make sure we specify in the `mean()` function to remove `NAs` by setting `na.rm = TRUE`.

We can use the `mean()` function inside of a `mutate()` function. Let's put this "grand" mean in a column labeled `grandRT`.

First take note of how many columns there are in `data`

```r
ncol(data)
## [1] 5
```

So after calculating the grandRT we should expect there to be one additional column for a total of 6 columns

```r
data <- mutate(data, grandRT = mean(RT, na.rm = TRUE))
```

Cool!

Now let's calculate another column that is the difference between `RT` and `grandRT`.

```
data <- mutate(data, RTdiff = RT - grandRT)
```

We can put all these `mutate()s` into one `mutate()`

```
data <- mutate(data,
               RT = ifelse(RT < 200, NA, RT),
               grandRT = mean(RT, na.rm = TRUE),
               RTdiff = RT - grandRT)
```

Notice how I put each one on a separate line. This is just for ease of reading and so the line doesn't extend too far off the page. Just make sure the commas are still there at the end of each line.

9.8 case_when()

Often times you will want to `mutate()` values conditionally based on values in other columns. There are two functions that will help you do this, `ifelse()` and `case_when()`. `ifelse()` is a base R function and `case_when()` is a dplyr function.

`ifelse()` takes the format: `ifelse(conditional argument, value if TRUE, value if FALSE)`

As an example, lets say we want to code a new variable that indicates whether the reaction time on a trial met a certain response deadline or not. Let's call this column `Met_ResponseDeadline` and give a value of `1` to trials that met the deadline and `0` to trials that did not meet the deadline. Let's set the response deadline at a reaction time of `500` milliseconds.

The conditional argument will take the form: `RT` is less than or equal to 500. If this statement is `TRUE`, then we will assign a value of `1` to the column `Met_ResponseDeadline`. If this statement is `FALSE`, then we will assign a value of `0` to the column `Met_ResponseDeadline`.

The code looks like:

```
data <- import |>
  mutate(Met_ResponseDeadline = ifelse(RT <= 500, 1, 0))
```

Check out `data` to make sure it worked.

You can even combine multiple `ifelse()` statements into one. Let's say we actually want to recode the column `ACC` to reflect not just correct and incorrect response but also whether they met the response deadline or not. That is, a value of `1` will represent responses that were

correct AND met the response deadline and values of `0` represent responses that were either incorrect, did not meet the response deadline, or both.

```
data <- import |>
  mutate(ACC = ifelse(ACC == 1, ifelse(RT <= 500, 1, 0), 0))
```

The arguments for the first `ifelse()` are as follows: Accuracy is equal to 1. If TRUE, then second `ifelse()` statement. If FALSE, then 0.

This makes sense because if the accuracy is `0` (incorrect), then the value needs to remain `0`. However, if the accuracy is `1`, the value will depend on whether the reaction time is less than `500` (thus the second `ifelse()`).

If accuracy is equal to `1`, then if reaction time is less than or equal to `500`, then set accuracy to `1`. If FALSE, then set accuracy to `0`.

Know that you can place the additional `ifelse()` statement in either the TRUE or FALSE argument and can keep iterating on `ifelse()` statements for as long as you need (however that can get pretty complicated).

`case_when()` is an alternative to an `ifelse()`. Anytime you need multiple `ifelse()` statements `case_when()` tends to simplify the code and logic involved.

Let's see examples of the two examples provided for `ifelse()` as a comparison.

```
data <- import |>
  mutate(Met_ResponseDeadline = case_when(RT <= 500 ~ 1,
                                          RT > 500 ~ 0))
```

Notice that the notation is quite different here. Each argument contains the format: conditional statement followed by the symbol `~` (this should be read as "then set as") and then a value to be assigned **when** the conditional statement is `TRUE`. There is no value to specify **when** it is `FALSE`.

Therefore, it is important when using the `case_when()` function to either 1) include enough TRUE statement arguments to cover ALL possible values or 2) use the uncharacteristically non-intuitive notation - `TRUE ~ "some value"`. In the example above, all possible `RT` values are included in the two arguments `RT <= 500` and `RT > 500`.

To provide an example of the second option:

```
data <- import |>
  mutate(Met_ResponseDeadline = case_when(RT <= 500 ~ 1,
                                          TRUE ~ 0))
```

The `case_when()` function will evaluate each argument in sequential order. So when it gets to the last argument (and this should always be the last argument), this is basically saying, **when** it is `TRUE` that none of the above arguments were `TRUE` (hence why this argument is being evaluated) then (`~`) set the value to "some value" (whatever value you want to specify).

Now this function gets a little more complicated if you want to set values to `NA`. `NA` values are technically logical values like `TRUE` or `FALSE`. The values in a column can only be of one type; numerical, character, logical, etc. Therefore, if you have numerical values in a column but want to set some to `NA`, then this becomes an issue when using `case_when()` (hopefully this will be fixed in future updates to `dplyr`). For now, how to get around this is changing the type of value that `NA` is. For instance; `as.numeric(NA)`, `as.character(NA)`.

```
data <- import |>
  mutate(Met_ResponseDeadline = case_when(RT <= 500 ~ 1,
                                          RT > 500 ~ 0,
                                          TRUE ~ as.numeric(NA)))
```

Now on to the example in which we used two `ifelse()` statements.

```
data <- import |>
  mutate(ACC = case_when(ACC == 1 & RT <= 500 ~ 1,
                         ACC == 1 & RT > 500 ~ 0,
                         ACC == 0 ~ 0))
```

When you have multiple `ifelse()` statements `case_when()` becomes easier to read. Compare this use of `case_when()` with the equivalent `ifelse()` above.

The `case_when()` function makes it very explicit what is happening. There are three conditional statements, therefore three categories of responses.

1. A correct response and reaction time that meets the deadline.
2. A correct response and reaction time that DOES NOT meet the deadline.
3. An incorrect response

These three options cover all possible combinations between the the two columns `ACC` and `RT`.

Accuracy should only be set to `1` (correct) with the first option and that is made quite clearly because it is the only one with `~ 1`.

This is not as obvious in the `ifelse()` example.

Let's move on to the next `dplyr` function.

9.9 group_by()

This function is very handy if we want to perform functions separately on different groups or splits of the data frame. For instance, maybe instead of calculating an overall "grand" mean we want to calculate a "grand" mean for each Subject separately. Instead of manually breaking the data frame up by Subject, the `group_by()` function does this automatically in the background. Like this...

```
data <- data |>
  group_by(Subject) |>
  mutate(RT = ifelse(RT < 200, NA, RT),
         grandRT = mean(RT, na.rm = TRUE),
         RTdiff = RT - grandRT) |>
  ungroup()
```

> **Caution**
>
> I suggest exercising caution when using `group_by()` because the grouping will be maintained until you specify a different `group_by()` or until you ungroup it using `ungroup()`. So I always like to `ungroup()` immediately after I am done with it.

You will now notice that each subject has a different `grandRT`, simply because we specified `group_by(data, Subject)`. Let's say we want to do it not just grouped by Subject, but also Condition.

```
data <- data |>
  group_by(Subject, Condition) |>
  mutate(RT = ifelse(RT < 200, NA, RT),
         grandRT = mean(RT, na.rm = TRUE),
         RTdiff = RT - grandRT) |>
  ungroup()
```

`group_by()` does not only work on `mutate()` - it will work on any other functions you specify after `group_by()`. Therefore, it can essentially replace most uses of for loops.

9.10 .by vs. group_by()

A new feature was recently introduced to replace some uses of `group_by()`. You can use the argument `.by =` inside of `mutate()` and `summarise()` (and other functions) to avoid having to use `ungroup()`.

`.by =` will only group the data frame for that one function and the returned output will be an ungrouped data frame. In general, I would suggest using `.by =` unless there is good reason to use `group_by()`.

The above grouped mutate can be performed with `by =` like this:

```
data <- data |>
  mutate(.by = c(Subject, Condition),
         RT = ifelse(RT < 200, NA, RT),
         grandRT = mean(RT, na.rm = TRUE),
         RTdiff = RT - grandRT)
```

You can see that `.by =` is also more concise (fewer lines of code) than `group_by()`

9.11 summarise()

The `summarise()` function will **reduce** a data frame by summarizing values in one or multiple columns. The values will be summarised on some statistical value, such as a mean, median, or standard deviation.

Remember that in order to calculate the `FlankerEffect` for each subject, we first need to calculate each subject's mean `RT` on `incongruent` trials and their mean `RT` on `congruent` trials

We've done our filtering, selecting, mutating, now let's aggregate RTs across `Condition` to calculate mean `RT`. We will use a combo of `.by =` and `summarise()`. `summarise()` is almost always used in conjunction with `.by =` or `group_by()`.

Let's also summarise the mean accuracy across conditions.

```
data <- data |>
  summarise(.by = c(Subject, Condition),
            RT.mean = mean(RT, na.rm = TRUE),
            ACC.mean = mean(ACC, na.rm = TRUE))
```

To `summarise()` you need to create new column names that will contain the aggregate values. `RT.mean` seems to make sense to me.

What does the resulting data frame look like? There should be three rows per subject, one for incongruent trials, one for congruent trials, and one for neutral trials. You can see that we now have mean RTs on all conditions for each subject.

Also, notice how non-grouped columns got removed: `Trial`, and `ACC`.

9.12 pivot_wider()

Our data frame now looks like

```
head(data)
## # A tibble: 6 x 4
##   Subject Condition   RT.mean ACC.mean
##     <dbl> <chr>         <dbl>    <dbl>
## 1   14000 incongruent    510.    0.574
## 2   14000 congruent      401.    0.931
## 3   14001 incongruent    423.    0.852
## 4   14001 congruent      392.    0.980
## 5   14002 congruent      462.    0.765
## 6   14002 incongruent    536.    0.463
```

Ultimately, we want to have one row per subject and to calculate the difference in mean RT between incongruent and congruent conditions. It is easier to calculate the difference between two values when they are in the same row. Currently, the mean RT for each condition is on a different row. What we need to do is **reshape** the data frame. To do so we will use the `pivot_wider()` function from the `tidyr` package.

The `tidyr` package, like `readr` and `dplyr`, is from the **tidyverse** set of packages. The `pivot_wider()` function will convert a long data frame to a wide data frame. In other words, it will spread values on different rows across different columns.

In our example, what we want to do is `pivot_wider()` the mean RT values for the two conditions across different columns. So we will end up with is one row per subject and one column for each condition. Rather than incongruent, and congruent trials being represented down rows we are spreading them across columns (widening the data frame).

The three main arguments to specify in `pivot_wider()` are

- **id_cols**: The column names that uniquely identifies (e.g. "Subject") each observation and that you want to be retained when **reshaping** the data frame.
- **names_from**: The column name that contains the variables to create new columns by (e.g. "Condition"). The values in this column will become Column names in the wider data format
- **values_from**: The column name that contains the values (e.g. "RT").

```
data_wide <- data |>
  pivot_wider(id_cols = "Subject",
              names_from = "Condition",
```

```
              values_from = "RT.mean")
```

Now our data frame looks like

```
head(data_wide)
## # A tibble: 6 x 3
##   Subject incongruent congruent
##     <dbl>       <dbl>     <dbl>
## 1   14000        510.      401.
## 2   14001        423.      392.
## 3   14002        536.      462.
## 4   14003        679.      567.
## 5   14004        655.      548.
## 6   14005        559.      472.
```

Notice that the ACC.mean column and values were dropped. To add more transparency to our data frame it would be a good idea to label what values the "congruent" and "incongruent" columns contain. You can do this with the optional `names_prefix` argument. For instance:

```
data_wide <- data |>
  pivot_wider(id_cols = "Subject",
              names_from = "Condition",
              values_from = "RT.mean",
              names_prefix = "RT_")
```

```
head(data_wide)
## # A tibble: 6 x 3
##   Subject RT_incongruent RT_congruent
##     <dbl>          <dbl>        <dbl>
## 1   14000           510.         401.
## 2   14001           423.         392.
## 3   14002           536.         462.
## 4   14003           679.         567.
## 5   14004           655.         548.
## 6   14005           559.         472.
```

Now a stranger (or a future YOU) will be able to look at this data frame and immediately know that reaction time values are contained in these columns.

From here it is pretty easy, we just need to create a new column that is the difference between incongruent and congruent columns. We can use the `mutate()` function to do this

```r
data_wide <- data_wide |>
  mutate(FlankerEffect_RT = RT_incongruent - RT_congruent)
```

```r
head(data_wide)
```

```
## # A tibble: 6 x 4
##   Subject RT_incongruent RT_congruent FlankerEffect_RT
##     <dbl>          <dbl>        <dbl>            <dbl>
## 1   14000           510.         401.            109.
## 2   14001           423.         392.             31.3
## 3   14002           536.         462.             74.0
## 4   14003           679.         567.            113.
## 5   14004           655.         548.            107.
## 6   14005           559.         472.             87.1
```

Perfect! Using the `readr`, `dplyr`, and `tidyr` packages we have gone from a "tidy" raw data file to a data frame with one row per subject and a column of FlankerEffect scores.

What if we have multiple columns we want to get **id_cols**, **names_from**, or **values_from**? `pivot_wider()` allows for this very easily. For instance:

```r
data_wide <- data |>
  pivot_wider(id_cols = "Subject",
              names_from = "Condition",
              values_from = c("RT.mean", "ACC.mean"))
```

```r
head(data_wide)
```

```
## # A tibble: 6 x 5
##   Subject RT.mean_incongruent RT.mean_congruent ACC.mean_incongruent
##     <dbl>               <dbl>             <dbl>                <dbl>
## 1   14000                510.              401.                0.574
## 2   14001                423.              392.                0.852
## 3   14002                536.              462.                0.463
## 4   14003                679.              567.                0.0926
## 5   14004                655.              548.                0.0370
## 6   14005                559.              472.                0.204
## # i 1 more variable: ACC.mean_congruent <dbl>
```

Now you can see that we have four columns corresponding to reaction times and accuracy values across the two conditions. You can use the same notation `c()` if you want to use multiple column for **id_cols**, **names_from**, **values_from**.

Now we can calculate a FlankerEffect for both RT and Accuracy values

```
data_wide <- data_wide |>
  mutate(FlankerEffect_RT = RT.mean_incongruent - RT.mean_congruent,
         FlankerEffect_ACC = ACC.mean_incongruent - ACC.mean_congruent)
```

9.13 pivot_longer()

For our goal with this data set, we do not need to switch back to a **longer** data format, however **reshaping** your data to a **longer** format may be something you want to do one day.

Let's try to reshape the `data_wide` back to a long format that we originally started with.

When you have multiple **value** columns this is not as intuitive as `pivot_wider()`. To see more documentation and examples use `?tidyr::pivot_longer()`.

```
data_long <- data_wide |>
  pivot_longer(contains("mean"),
               names_to = c(".value", "Condition"),
               names_sep = "_")
```

9.14 All Together Now

We can pipe the relevant functions in the chapter together as such

```
## Setup
library(readr)
library(dplyr)
library(tidyr)

## Import
import <- read_csv("data/tidyverse_example.csv")

## Score
data <- import |>
  rename(Accuracy = ACC) |>
  filter(TrialProc == "real") |>
  select(Subject, Condition, RT, Trial, ACC = Accuracy) |>
  group_by(Subject, Condition) |>
```

```
mutate(RT = ifelse(RT < 200, NA, RT),
       grandRT = mean(RT, na.rm = TRUE),
       RTdiff = RT - grandRT) |>
summarise(RT.mean = mean(RT, na.rm = TRUE),
          ACC.mean = mean(ACC, na.rm = TRUE)) |>
ungroup() |>
pivot_wider(id_cols = "Subject",
            names_from = "Condition",
            values_from = c("RT.mean", "ACC.mean")) |>
mutate(FlankerEffect_RT = RT.mean_incongruent - RT.mean_congruent,
       FlankerEffect_ACC = ACC.mean_incongruent - ACC.mean_congruent)
```

Virtually all the R scripts you write will require the `dplyr` package. The more you know what it can do, the easier it will be for you to write R Scripts. I highly suggest checking out these introductions to `dplyr`.

https://dplyr.tidyverse.org https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html

Part III

Project Organization

10 Reproducible Workflow

While it may be tempting to jump right into working with data and conducting analyses, it's crucial to consider how you organize your projects, files, and data. This is especially important when working with data in R and creating fully reproducible workflows.

Part of the scientific process involves carefully documenting every step in our procedures. Doing so not only ensures higher quality research, but also enables your future self and others to fully reproduce what you did, go back and analyze the data in a different way, or catch errors in the data analysis process. Without a fully reproducible project, it may be difficult or impossible to catch errors that were made.

The typical data analysis workflow looks something like this:

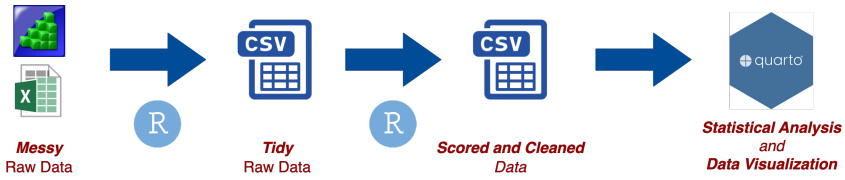


Figure 10.1: Data analysis workflow

1. The first step in working with raw data files is to prepare the data so that it is usable and easy to understand. This often involves cleaning up the **messy** raw data and transforming it into a **tidy** format.
2. Once the data is tidy, the next step is typically to aggregate the data across responses (e.g., accuracy, RT) and clean the data to create a **scored and cleaned** data file.
3. With the data cleaned and scored, it is then ready for **statistical analysis** and **data visualization**. Depending on the project, there may be additional steps required before conducting statistical analysis and visualization.

The first two steps can be accomplished using R scripts. The general process is to import the raw data file, transform the data (e.g., tidy the data or aggregate responses), and save the transformed data as an output file.

The third step is best done using Quarto documents. This allows for easy rendering and sharing of the results.

10.1 What does reproducibility mean?

Reproducibility in data processing and analysis means that all processing and analysis steps can be fully replicated using only the original raw data files and the execution of the R (or other program) scripts. There are different levels of reproducibility:

1. Partially reproducible - only some data processing and analysis steps can be reproduced, which may be due to a lack of original raw data files or the use of non-reproducible software.
2. Minimal level of reproducibility (acceptable) - full reproduction of data processing and analysis steps on your computer (or a lab computer) without any modifications needed.
3. Moderately reproducible (desired) - meets the minimal level plus other people not involved in the research project can reproduce the steps with minimal modifications.
4. Highly reproducible (good luck!) - full reproduction of steps without modification needed by people not involved in the research project 5 - 10+ years from now.

Note that these levels are arbitrarily defined by myself and what I came up with on the spot. A minimal level of reproducibility is still acceptable, as achieving more requires significant time and effort. Though, we should strive for a moderate amount of reproducibility, and achieving it requires more than just writing code. Your code must be organized, easy to understand, and include notes and documentation. Even if you or someone else attempts to reproduce your steps in the future, they can modify the code to make it work. The highest level of reproducibility is difficult to achieve due to software and code updates. Your code may only work with the current version of R or other packages. There are solutions to this problem of software and code updates, but who knows if those will work in the future!

Simply using R for data analysis does not guarantee that your workflow is reproducible. In fact, there are many non-reproducible ways to use R. To ensure at least a moderate level of reproducibility, consider the following criteria (this is not an exhaustive list):

- Your statistical analysis can be fully reproduced using only the raw data files and R scripts
- Your code can be reproduced on other computers without any modifications
- **Your data and R scripts are organized and documented in a way that makes them easily understandable to unfamiliar parties**

This last criterion is extremely important, but is often overlooked. Simply posting your data and scripts to an open access repository like OSF is not enough to guarantee reproducibility. If others cannot understand your workflow, then it is not reproducible. Therefore, it is crucial to take the time to think about the organization of your project, files, data, and scripts.

11 File Organization

Good project organization starts with easy to understand folder and file organization. For a data analysis project, the following folders are typically needed for a fully reproducible workflow:

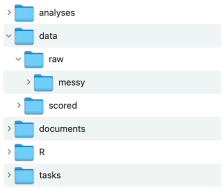


Figure 11.1: Folder organization

Notice how the structure of the **data** folder follows the data analysis workflow. The original **messy** raw data files are stored in **data / raw / messy**. The **tidy** raw data files are stored in **data / raw**. And the **scored and cleaned** data files are stored in **data / scored**.

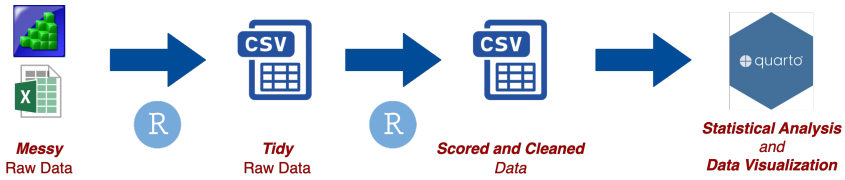


Figure 11.2: Data analysis workflow

This organizational structure makes it easy to understand how the various **R Scripts**, stored in the **R** folder, correspond to each step in the data analysis workflow, what data the script is importing and where it is coming from, and what data file the script is outputting and saving to.

To help with this even further, I typically follow a particular file naming scheme.

- I like to just keep the original data file name for the **messy** raw data file.
- For **tidy** raw data files I use **taskname_raw.csv**
- For **scored** data files I use **taskname_scored.csv**

- For R Scripts I will append a prefix number corresponding to the order in which the scripts need to be ran and a suffix corresponding to what stage of the data analysis workflow it is in:
  - **1_taskname_raw.R**
  - **2_taskname_score.R**
  - **3_merge_scores.R**

It could be tempting to just throw all your R code into one script or RMarkdown document, however, I advise against that because it will make it more difficult to manage, reproduce, and understand your data processing workflow.

11.1 RProjects and here()

You need to be using RStudio Projects for anything you do in R.

In fact, **you should be opening RStudio by opening an RProject file**.

RStudio Projects allow you to open isolated instances of R and RStudio for each of your projects. In combination with the `here` package it will also provide a simple and fool proof way of specifying file paths.

Every time you use `here()` you know that the file path will start at where you have your .Rproj file saved. Instead of messing around with working directories using `setwd()` or `getwd()`, just use `here()` and RStudio Projects. This becomes especially helpful when working with Quarto documents.

For instance, I have an .Rproj file saved in my "useRguide" folder. When I use `here()`, the function will start at a file path to that location.

```
library(here)
## here() starts at /Users/jtsukahara3/GitHub Repos/useRguide
here()
## [1] "/Users/jtsukahara3/GitHub Repos/useRguide"
```

You can then use a relative file path inside of `here()`.

```
here("data/raw/flanker_raw.csv")
## [1] "/Users/jtsukahara3/GitHub Repos/useRguide/data/raw/flanker_raw.csv"
```

This is equivalent to

```r
here("data", "raw", "flanker_raw.csv")
## [1] "/Users/jtsukahara3/GitHub Repos/useRguide/data/raw/flanker_raw.csv"
```

I typically like to set the first argument as the relative file path and the second argument as the file name. This visually separates the file path and the file name, making your script easier to read.

```r
here("data/raw", "flanker_raw.csv")
## [1] "/Users/jtsukahara3/GitHub Repos/useRguide/data/raw/flanker_raw.csv"
```

You can then use `here()` directly in import and output functions:

```r
import <- read_csv(here("data/raw", "flanker_raw.csv"))
```

```r
write_csv(data, here("data/scored", "flanker_scored.csv"))
```

11.2 mainscript file

When you break up your data processing stages into separate R scripts, this can lead to having a lot of R scripts to source. It can be tedious to open each script, source it, then repeat for every R script while making sure you are doing everything in the right order.

Instead, I prefer to create a **mainscript.Rmd** file that uses `source()` to source each R script file in a particular order, and `render_quarto()` to render Quarto documents.

Using a Quarto (.qmd) document for the mainscript file is useful if you want to add in documentation about the study and data analysis workflow. For instance, you can add brief descriptions of how outliers were detected and dealt with, how reliability was calculated, etc.

It also makes it easier for yourself, your future self, or someone else to look at your **mainscript.qmd** file and easily understand your data processing workflow without having to go through each of your R scripts line-by-line.

11.3 Use Templates!

Don't start from scratch, like ever.

It can be a ton of work, and mental effort, to create an efficient and reproducible project organization from scratch. Generally, I would try to avoid starting a project from blank, empty, R scripts. More than likely what will happen is that any organization you had thought

of just breaks down because you are struggling enough with writing the R code itself, and you need to get a report on the data to your advisor so you decide to use shortcuts **"for now"**. But we all know that **"for now"** always turns into **forever**.

Unfortunately, not starting from scratch usually means that you have already setup an organization system with R script templates that you can use. This is likely not the case. So what to do?

Well. in the next chapter, I will introduce you to my `psyworkflow` package that allows you to automatically setup the organizational structure I have outlined in this chapter, along with R script templates for each data processing stage.

12 psyworkflow

In this chapter, I introduce you to my `psyworkflow` package that allows you to automatically setup the organizational structure I have outlined in the previous chapter, along with R script templates for each data processing stage.

12.1 Install

First, if you do not have the `devtools` package installed:

```
install.packages("devtools")
```

Install the `psyworkflow` package from my GitHub repository using the `devtools` package:

```
devtools::install_github("dr-JT/psyworkflow")
```

Restart R: Session -> Resart R

12.2 Download R Script Templates

If you already have an RProject setup and just want to download some of the R script templates you can do so with the `get_template()` function.

```
workflow::get_template()
```

To see what the options are type in the console window

```
?workflow::get_template
```

12.3 Create a New Project

Close RStudio and reopen a new instance of RStudio (not from an RProject file).

Once you have the `psyworkflow` package installed you will be able to create a new RProject directory and file from my **Research Study** template. This will automatically create the directory structure outlined in the previous chapter. It will also add template R scripts in **R / templates** and a **masterscript.Rmd** file.

Using this template will allow you to get right to working with your data in R, without having to spend too much time thinking about organization (I have already done that for you).

To create an RProject from this template:

File -> New Project… -> New Directory -> Research Study (you might need to scroll down to see it)

This will bring up a window to customize the template:

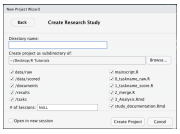


Figure 12.1: Create new project window

Type in whatever you want for the **Directory Name** - this will end up being the name of the project folder and RProject file.

Click on **Browse...** and create the project on your desktop, for now.

Keep all the defaults and select **Create Project**.

Give it some time, and it will reopen RStudio from the newly created RProject. Take a look at the file pane and you can see that the folders have been created, and R Script templates downloaded.

Part IV

Data Processing

13 Processing Steps

There are three stages when processing data: 1) data preparation, 2) scoring and cleaning data, 3) preparing a single data file for statistical analysis.

13.1 Data Preparation

There are two scenarios in which you may need to start processing and analyzing data:

- Before data collection has finished
- After data collection has finished

For both of these scenarios, you will start with messy raw data files in some file format. Messy raw data files are hard to understand, have poor column and value labels, contain way too many columns and rows, and are just hard to work with. Data preparation is all about getting raw data files that are easy to work with.

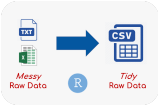


Figure 13.1: Data preparation workflow

The end product of the data preparation stage is *tidy* raw data files. Tidy raw data files are easy to understand, have sensible column and value labels, contain only relevant columns and rows, and are very easy to work with.

The **Data Preparation** stage is required because the data files created from the E-Prime or other software program are usually not in a format that is easy to use or understand. I am referring to this format as a **messy raw data** file. Also, there are typically other preparation steps one needs to take before they can start looking at the data. These might include merging individual subject data files and exporting the data to a non-proprietary format so we can import the data into R. The purpose of the data preparation stage is simply to create **tidy raw data** files from the messy raw data files.

Tidy raw data files are easy to use and understand. There will be one row per trial, column labels that are easy to understand (e.g. Trial, Condition, RT, Accuracy, etc.), and values in

columns that make sense. If values in a column are categorical, then the category names will be used rather than numerical values. Ideally, someone not involved in the research project should be able to look at a tidy raw data file and understand what each column represents, and what the values in the column refer to.

13.2 Scoring and Cleaning Data

Once you have tidy raw data files, you can start scoring the tasks and implement any data cleaning procedures. In general, there are three steps that are typically done at this stage:

1. Scoring the data

   This involves aggregating performance across trials on a task for each subject.

2. Cleaning the data

   This involves removing subjects that are identified to be problematic and / or outliers.

3. Calculating reliability

   For individual differences, it is important to report the reliability of each task.

13.3 Preparing a Single Data File

The scoring and cleaning data stage will produce scored data files for each task, but ultimately we want a single data file to use for statistical analysis.

Therefore, the next stage is to join all the scored data files into a single dataframe and select only relevant variables (columns) that we need for analysis.

13.4 Setup Project Folder

Before moving on to the other chapters in this section, create a research study project folder. See Chapter 12 on how to do this.

14 Tidy Raw Data

> **Tip**
>
> Watch a video tutorial of this chapter!

For this chapter you will need to:

- See Chapter 9 for working with data in R if you are completely new to R or the tidyverse
- install the `purrr` package

```
install.packages("purrr")
```

- install the `psyworkflow` package

```
# if you do not have devtools installed then do this first
install.packages("devtools")
# install workflow
devtools::install_github("dr-JT/psyworkflow")
```

- Setup a project folder, see Chapter 12
- download this data file to **data/raw** in your project folder
- get an R script template for tidying raw data

```
psyworkflow::get_template(raw_script = TRUE)
```

14.1 Overview of Template

14.1.1 Setup

```
# ---- Setup ----
# packages
library(here)
library(readr)
library(dplyr)
library(purrr) # delete if importing a single file, not a batch of files

# directories
import_dir <- "data/raw/messy"
output_dir <- "data/raw"

# file names
task <- "taskname"
import_file <- paste(task, ".txt", sep = "")
output_file <- paste(task, "raw.csv", sep = "_")
# ---------------
```

- Load packages

  Any packages required for this script are loaded at the top. The `here`, `readr`, and `dplyr` packages will be used. The `purrr` package will be used if you are importing a batch of files. If you are only importing a single file you will not need this and therefore should be deleted.

- Set Import/Output Directories

  To make this example easier you will not have to actually import/output any files.

- Set Import/Output Filenames

  The only line we need to change here is the `task <- "taskname"` to `task <- "VAorient_S"`.

14.1.2 Import

I have provided two different import options. The first one is if you are importing a single file. The second is if you are importing a batch of files. For more details on importing a batch of files see Section 8.6.1

```
# ---- Import Data ----
# to import a single file
data_import <- read_delim(here(import_dir, import_file), delim = "\t",
                          escape_double = FALSE, trim_ws = TRUE)

# alternatively to import a batch of files...
# change the arguments in purrr::map_df() depending on type of data files
# this example is for files created from eprime and needs encoding = "UCS-2LE"
files <- list.files(here(import_dir, task), pattern = ".txt", full.names = TRUE)
data_import <- files |>
  map_df(read_delim, locale = locale(encoding = "UCS-2LE"),
         delim = "\t", escape_double = FALSE, trim_ws = TRUE, na = "NULL")
# ---------------------
```

14.1.3 Tidy raw data

This is the meat of the script, where the action happens. It will also be different for every task - obviously. I will cover these steps in more detail below.

```
# ---- Tidy Data ----
data_raw <- data_import |>
  filter() |>
  rename() |>
  mutate() |>
  select()
# -------------------
```

14.1.4 Output data

No need to change anything here. Isn't that nice?

```
# ---- Save Data ----
write_csv(data_raw, here(output_dir, output_file))
# -------------------

rm(list = ls())
```

14.2 Filter Rows

One of the first things that is useful to do is get rid of rows in the messy data file that you don't need.

For E-Prime data, `Procedure[Trial]` is usually the column name you need to only keep rows for practice and real trials procedures.

> **Tip**
>
> Type `colnames(data_import)` in the console window to get a read out of all the column names in your data. It is much faster and easier to see column names in the console than navigating the data frame itself.

You need to figure out the value names that correspond to the rows you want to keep. Use `unique(``Procedure[Trial]``)`

Let's say we only want to keep rows that have a value in the `Procedure[Trial]` column as either `TrialProc` or `PracProc`.

```
rename(TrialProc = `Procedure[Trial]`)
filter(TrialProc == "TrialProc" | TrialProc == "PracProc")
```

14.3 Change Values in Columns

You will likely want to change some of the value labels in columns to make more sense and standardize it across tasks. In general, you should avoid numeric labels for categorical data. Instead, you should just use word strings that describe the category intuitively (e.g., "red", "blue", "green" instead of 1, 2, 3).

Let's change the `TrialProc` values so they are more simple and easy to read.

```
mutate(TrialProc = case_when(TrialProc == "TrialProc" ~ "real",
                             TrialProc == "PracProc" ~ "practice",
                             TRUE ~ as.character(NA)))
```

You may want to do more complex changing of values or creating entirely new columns. See the **Working with Data* section for more details.

14.4 Keep only a few Columns

You will also likely want to select only a subset of columns to keep in the tidy raw data file.

```
select(Subject, TrialProc, Trial, Condition, Accuracy, RT, Response,
       CorrectResponse, AdminTime, SessionDate, SessionTime)
```

15 Score and Clean Data

For this chapter you will need to:

- See Chapter 9 for working with data in R if you are completely new to R or the tidyverse
- install the `psyworkflow` package

```
# if you do not have devtools installed then do this first
install.packages("devtools")
# install workflow
devtools::install_github("dr-JT/psyworkflow")
```

- install the `psych` package

```
install.packages("psych")
```

- Setup a project folder, see Chapter 12
- download this data file to **data/raw** in your project folder
- get an R script template for scoring raw data

```
psyworkflow::get_template(score_script = TRUE)
```

15.1 Overview

At this stage of data processing you have tidy raw data files for each task. The next step is to process the raw data into aggregate summary variables. This involves several steps:

1. Calculate task scores
2. Evaluate and remove problematic subjects and outliers
3. Calculate reliability

15.2 Setup

```
# ---- Setup ----
# packages
library(here)
library(readr)
library(dplyr)
library(englelab)    # for data cleaning functions
library(tidyr)       # for pivot_wider. delete if not using
library(psych)       # for cronbach's alpha. delete if not using

# directories
import_dir <- "data/raw"
output_dir <- "data/scored"

# file names
task <- "taskname"
import_file <- paste(task, "raw.csv", sep = "_")
output_file <- paste(task, "Scores.csv", sep = "_")

## data cleaning parameters
outlier_cutoff <- 3.5
# --------------
```

- packages

  Any packages required for this script are loaded at the top. For this task we will need the `here`, `readr`, `dplyr`, `tidyr`, `englelab`, and `psych` packages.

- directories

  If you are using my default project organization, you do not need to change anything here.

- file names

  The only line we need to change here is the `task <- "taskname"` to `task <- "VAorient_S"`.

- data cleaning parameters

  In this section of the script we can set certain data cleaning criterion to variables. This makes it easy to see what data cleaning criteria were used right at the top of the script rather than having to read through and try to interpret the script.

For the visual arrays task we should remove subjects who had low accuracy scores - lets say those with accuracy lower 50% (chance level performance on this task)

Add `acc_criterion <- .5` to this section of the script.

There is already an outlier criterion added to this section by default. This criterion will remove outliers that have scores on the task greater or less than 3.5 standard deviations from the mean.

15.3 Import

```
# ---- Import Data ----
data_import <- read_csv(here(import_dir, import_file))
# ---------------------
```

Because of this template, you can now quickly import the data file by running the line of code in the import section. Take a look at the dataframe.

Notice how there are rows that correspond to **practice** trials. We do not want to use these rows when calculating scores on the task. So let's filter for only **real** trials right when we import the data, so that we won't have to remember to do it later.

Add a row to keep only **real trials**

```
# ---- Import Data ----
data_import <- read_csv(here(import_dir, import_file)) |>
  filter(TrialProc == "real")
# ---------------------
```

15.4 Calculate Task Scores

This is where the action happens. It will also be different for every task - obviously. However, there are a few steps that are pretty common.

```
# ---- Score Data ----
data_scores <- data_import |>
  muatate(.by = Subject) |>
  summarise(.by = Subject)
# --------------------
```

15.4.1 Trim Reaction Time

To trim reaction times less than 200ms:

```
mutate(RT = ifelse(RT < 200, NA, RT))
```

To trim reaction times less than 200ms or greater than 10000ms

```
mutate(RT = ifelse(RT < 200 | RT > 10000, NA, RT))
```

Alternatively, using `dplyr::case_when()`

```
mutate(RT = case_when(RT < 200 ~ as.numeric(NA),
                      RT > 10000 ~ as.numeric(NA),
                      TRUE ~ RT))
```

15.4.2 Summary Statistic

Once you group and clean the data, you can calculate a summary statistic such as a mean, median, or standard deviation.

To calculate the mean accuracy and reaction time for each subject and condition:

```
summarise(.by = c(Subject, Condition),
          Accuracy.mean = mean(Accuracy, na.rm = TRUE),
          RT.mean = mean(RT, na.rm = TRUE))
```

15.4.3 Transform Data to Wide

If you are grouping by Subject and Condition, then you will likely want to transform the aggregated data into a wide format. This is because `summarise()` will produce a row for each Condition per Subject. What you might want is a single row per subject, with the conditions spread out across columns.

> **Tip**
>
> If you forget how to use a function or what the argument names are then type `?functionName()` in the console (e.g. `?pivot_wider()`).

```r
pivot_wider(id_cols = Subject,
            names_from = Condition,
            values_from = Accuracy.mean)
```

15.4.4 More Complex Scoring

This is an example of how to calculate k scores for the visual arrays task. You can see this is a little more involved.

```r
data_scores <- data_import |>
  summarise(.by = c(Subject, SetSize),
            CR.n = sum(CorrectRejection, na.rm = TRUE),
            FA.n = sum(FalseAlarm, na.rm = TRUE),
            M.n = sum(Miss, na.rm = TRUE),
            H.n = sum(Hit, na.rm = TRUE)) |>
  mutate(CR = CR.n / (CR.n + FA.n),
         H = H.n / (H.n + M.n),
         k = SetSize * (H + CR - 1)) |>
  pivot_wider(id_cols = Subject,
              names_from = SetSize,
              names_prefix = "VA.k_",
              values_from = k) |>
  mutate(VA.k = (VA.k_5 + VA.k_7) / 2)
```

15.5 Data Cleaning

The next section of the script template is for cleaning the data by removing problematic subjects and/or removing outliers.

15.5.1 Remove Problematic Subjects

Depending on the task, problematic subjects can be detected in different ways. For this task we will simply remove subjects that had less than chance performance (were just guessing or did not understand the task).

To do so, we will use a custom function created for this purpose, `remove_problematic()` from our `englelab` package. `?englelab::remove_problematic`

The main argument is `filter =`. You need to put in here, what you would put inside of `dplyr::filter()` to remove anyone with less than chance performance: `dplyr::filter(Accuracy.mean < acc_criterion)`. Remember at the top of the script we added the accuracy criterion by setting `acc_criterion <- .5` (chance performance).

The other argument, that is more optional, is `log_file =`. This allows us to save a data file containing only the subjects that were removed. This is good if we later on want to inspect how many subjects were removed, and why.

```
data_cleaned <- data_scores |>
  remove_problematic(
    filter = "Accuracy.mean < acc_criterion",
    log_file = here("data/logs", paste(task, "_problematic.csv", sep = "")))
```

15.5.2 Remove Outliers

Remove outliers based on their final task scores. A typical way we remove outliers is by setting their score to missing `NA` if it is 3.5 standard deviations above or below the mean.

To do so, we will use a custom function created for this purpose, `replace_outliers` from our `englelab` package.

There are several arguments that need to be defined. See `?englelab::replace_outliers` for a description on this.

The fully piped `|>` code for this entire section looks like:

```
# ---- Clean Data ----
data_cleaned <- data_scores |>
  remove_problematic(
    filter = "Accuracy.mean < acc_criterion",
    log_file = here("data/logs", paste(task, "_problematic.csv", sep = ""))) |>
  replace_outliers(
    variables = "VA.k",
    cutoff = outlier_cutoff,
    with = "NA",
    log_file = here("data/logs", paste(task, "_outliers.csv", sep = ""))) |>
  filter(!is.na())
# -------------------
```

Notice that `filter(!is.na())` is specified after `replace_outliers`. This removes any of the outliers from the dataframe.

15.6 Calculate Reliability

There are two standard ways of calculating reliability: split-half and cronbach's alpha.

It is best to calculate reliability only on the subjects that made it passed data cleaning.

```
reliability <- data_import |>
  filter(Subject %in% data_cleaned$Subject)
```

`reliability` will be the raw data frame that we will use to calculate reliabilty estimates on.

15.6.1 Split-half reliability

Here is an example if the task score was an aggregate of accuracy.

```
splithalf <- reliability |>
  mutate(.by = Subject,
         Split = ifelse(Trial %% 2, "odd", "even")) |>
  summarise(.by = c(Subject, Split),
            Accuracy.mean = mean(Accuracy, na.rm = TRUE)) |>
  pivot_wider(id_cols = Subject,
              names_from = Split,
              values_from = Accuracy.mean) |>
  summarise(r = cor(even, odd)) |>
  mutate(r = (2 * r) / (1 + r))

data_cleaned$Score_splithalf <- splithalf$r
```

For tasks, like the visual arrays, where the score is not a simple aggregate but a more complicated calculation this is more involved and we would basically want to score the data set for even and odd trials separately.

To calculate the split-half reliability of **k scores** on the visual arrays task would look something like:

```
splithalf <- reliability |>
  mutate(.by = c(Subject, SetSize),
         Split = ifelse(Trial %% 2, "odd", "even")) |>
  summarise(.by = c(Subject, SetSize, Split),
            CR.n = sum(CorrectRejection, na.rm = TRUE),
            FA.n = sum(FalseAlarm, na.rm = TRUE),
            M.n = sum(Miss, na.rm = TRUE),
```

```
            H.n = sum(Hit, na.rm = TRUE)) |>
  mutate(CR = CR.n / (CR.n + FA.n),
         H = H.n / (H.n + M.n),
         k = SetSize * (H + CR - 1)) |>
  pivot_wider(id_cols = Subject,
              names_from = c(SetSize, Split),
              names_prefix = "VA.k_",
              values_from = k) |>
  mutate(VA.k_even = (VA.k_5_even + VA.k_7_even) / 2,
         VA.k_odd = (VA.k_5_odd + VA.k_7_odd) / 2)
  summarise(r = cor(VA.k_even, VA.k_odd)) |>
  mutate(r = (2 * r) / (1 + r))

data_cleaned$Score_splithalf <- splithalf$r
```

15.6.2 Cronbach's alpha

Here is an example if the task score was an aggregate of accuracy.

```
cronbachalpha <- reliability |>
  select(Subject, Trial, Accuracy) |>
  pivot_wider(id_cols = Subject,
              names_from = Trial,
              values_from = Accuracy) |>
  select(-Subject) |>
  alpha()  # from the psych package

data_cleaned$Score_cronbachalpha <- cronbachalpha$total$std.alpha
```

16 Single Merged File

For this chapter you will need to:

- See Chapter 9 for working with data in R if you are completely new to R or the tidyverse
- install the `purrr` package

```
install.packages("purrr")
```

- install the `psyworkflow` package

```
# if you do not have devtools installed then do this first
install.packages("devtools")
# install workflow
devtools::install_github("dr-JT/psyworkflow")
```

- Setup a project folder, see Chapter 12
- download this data file to **data/raw** in your project folder
- get an R script template to merge multiple scored files

```
psyworkflow::get_template(merge_script = TRUE)
```

16.1 Overview

At this stage of data processing you have created multiple data files containing the task scores, reliabilities, and other variables for each task. The next step is to create a single merged data file containing the primary task scores for each task that you will be performing data analysis on. We can also create data files with the reliabilities, administration times, and a log of the data cleaning steps.

16.2 Setup

```
# ---- Setup ----
# packages
library(here)
library(readr)
library(dplyr)
library(purrr)
library(tidyr)

# directories
import_dir <- "data/scored"
output_dir <- "data"

# file names
output_scores <- "TaskScores.csv"
output_reliabilities <- "Reliabilities.csv"
output_admintimes <- "AdminTimes.csv"
output_datacleaning <- "DataCleaning_log.csv"
# --------------
```

- packages

  Any packages required for this script are loaded at the top. For this task we will need the `here`, `readr`, `dplyr`, `purrr`, and `tidyr` packages.

- directories

  If you are using my default project organization, you do not need to change anything here.

- file names

  Feel free to change these as you like. E.g., You may want to add the study name in front of these file names, "StudyName_TaskScores.csv".

16.3 Import

For more details on importing a batch of files see Section 8.6.2

```
# ---- Import Data ----
files <- list.files(here(import_dir), pattern = "Scores", full.names = TRUE)
```

```r
data_import <- files %>%
  map(read_csv) %>%
  reduce(full_join, by = "Subject")
# ----------------------
```

16.4 Task Scores

```r
# ---- Select Task Scores ----
data_scores <- data_import %>%
  select() %>%
  filter()

# list of final subjects
subjlist <- select(data_scores, Subject)
# ---------------------------
```

At this step, you may also want to filter out subjects that have missing data on specific tasks, or too much missing data across all the tasks.

I advise creating a final subject list of all subjects that made it to this state of data processing.

16.5 Reliabilities

```r
# ---- Reliabilities ----
data_reliabilities <- data_import %>%
  select(contains("splithalf"), contains("cronbachalpha")) %>%
  drop_na() %>%
  distinct() %>%
  pivot_longer(everything(),
               names_to = c("Task", "metric"),
               names_pattern = "(\\w+.\\w+).(\\w+)",
               values_to = "value") %>%
  pivot_wider(id_col = Task,
              names_from = metric,
              values_from = value)
# ----------------------
```

The code in `pivot_longer(names_pattern = "(\\w+.\\w+).(\\w+)")` follows a specific naming scheme used for column names. Underscores (optional) can be used for task names

and column descriptions, but a period (required) is ONLY used to seperate the task name / description from the description of the task score type / what the value represents (e.g., RT = reaction time, splithalf = split-half reliability)

Task_Name_MoreStuff.ScoreType

e.g., **StroopDL_Last4Rev.ResponseDeadline**, **VAorient_S.k**, **Antisaccade.ACC**

This should also be used for reliabilities and admin times:

e.g., **StroopDL_Last4Rev.splithalf**, **StroopDL.AdminTime**

16.6 Admin Times

```
# ---- Admin Times ----
data_merge <- data_import %>%
  select(contains("AdminTime")) %>%
  summarise_all(list(mean = mean, sd = sd), na.rm = TRUE) %>%
  pivot_longer(everything(),
               names_to = c("Task", "metric"),
               names_pattern = "(\\w+.\\w+).(\\w+)",
               values_to = "value") %>%
  mutate(value = round(value, 3)) %>%
  pivot_wider(id_col = Task,
              names_from = metric,
              names_prefix = "AdminTime.",
              values_from = "value")
# ---------------------
```

Part V

Data Visualization

17 Introduction to ggplot2

17.1 Fundamentals of Data Visualization

Data visualization is an essential skill for anyone working with data. It is a combination of statistical understanding and design principles and is really about **graphical data analysis** and **communication and perception**.

Data visualization is often times glossed over in our stats courses. This is unfortunate because it is so important for better understanding our data, for communicating our results to others, and frankly it is too easy to create poorly designed visualizations.

As a scientist, there are two purposes for visualizing our data.

1) Data exploration: it is difficult to fully understand our data just by looking at numbers on a screen arranged in rows and columns. Being skilled in data visualization will help you better understand your data.
2) Explain and Communicate: You will also need to explain and communicate your results to colleagues or in scientific publications.

The same data visualization principles apply to both purposes, however for communicating your results you may want to place more emphasis on aesthetics and readability. For data exploration your visualizations do not have to be pretty.

Leland Wilkinson (Grammar of Graphics, 1999) formalized two main principles in his plotting framework:

1) Graphics = distinct *layers* of grammatical elements
2) Meaningful plots through aesthetic mappings

The essential grammatical elements to create any visualization are:

Element	Description
Data	The dataset being plotted.
Aesthetics	The scales onto which we *map* our data.
Geometries	The visual elements used for our data.

Figure 17.1: Essential grammatical elements for data visualization

17.1.1 Plotting Functions in R

It is possible to create plots in R using the base R function `plot()`. The neat thing about `plot()` is that it is really good at knowing what kind of plot you want without you having to specify. However, these are not easy to customize and the output is a static image not an R object that can be modified.

To allow for data visualization that is more in line with the principles for a grammar of graphics, Hadley Wickham created the `ggplot2` package. This by far the most popular package for data visualization in R.

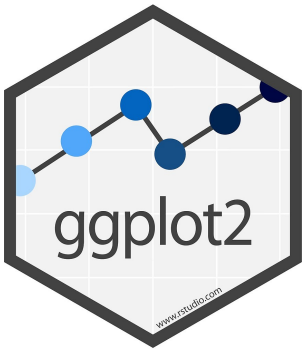


Figure 17.2: ggplot2 logo

17.2 Grammar of Graphics

We saw from the last chapter that the two main components in a grammar of graphics are:

1) Graphics = distinct *layers* of grammatical elements

2) Meaningful plots through aesthetic mappings

We also saw that the three essential elements are the data layer, aesthetics layer, and geometrics layer. In ggplot2 there are a total of 7 layers we can add to a plot

Element	Description
Data	The dataset being plotted.
Aesthetics	The scales onto which we *map* our data.
Geometries	The visual elements used for our data.
Facets	Plotting small multiples.
Statistics	Representations of our data to aid understanding.
Coordinates	The space on which the data will be plotted.
Themes	All non-data ink.

Figure 17.3: ggplot2 elements

17.3 Data layer

The Data Layer specifies the data being plotted.

Figure 17.4: ggplot2 data layer

Let's see what this means more concretely with an example data set. A very popular data set used for teaching data science is the `iris` data set. In this data set various species of iris were measured on their sepal and petal length and width.

This data set actually comes pre-loaded with R, so you can simply view it by typing in your console

```
View(iris)
```

```
head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

We can see that this data is in **wide** format. What type of graph we can visualize will depend on the format of the data set. On occasion, in order to visualize a certain pattern of the data will require you to change the formatting of the data.

Let's go ahead and start building our graphical elements in `ggplot2`. Load the `ggplot2` library. Then:

```
library(ggplot2)
ggplot(data = iris)
```

You can see that we only have a blank square. This is because we have not added any other layers yet, we have only specified the data layer.

17.4 Aesthetics Layer

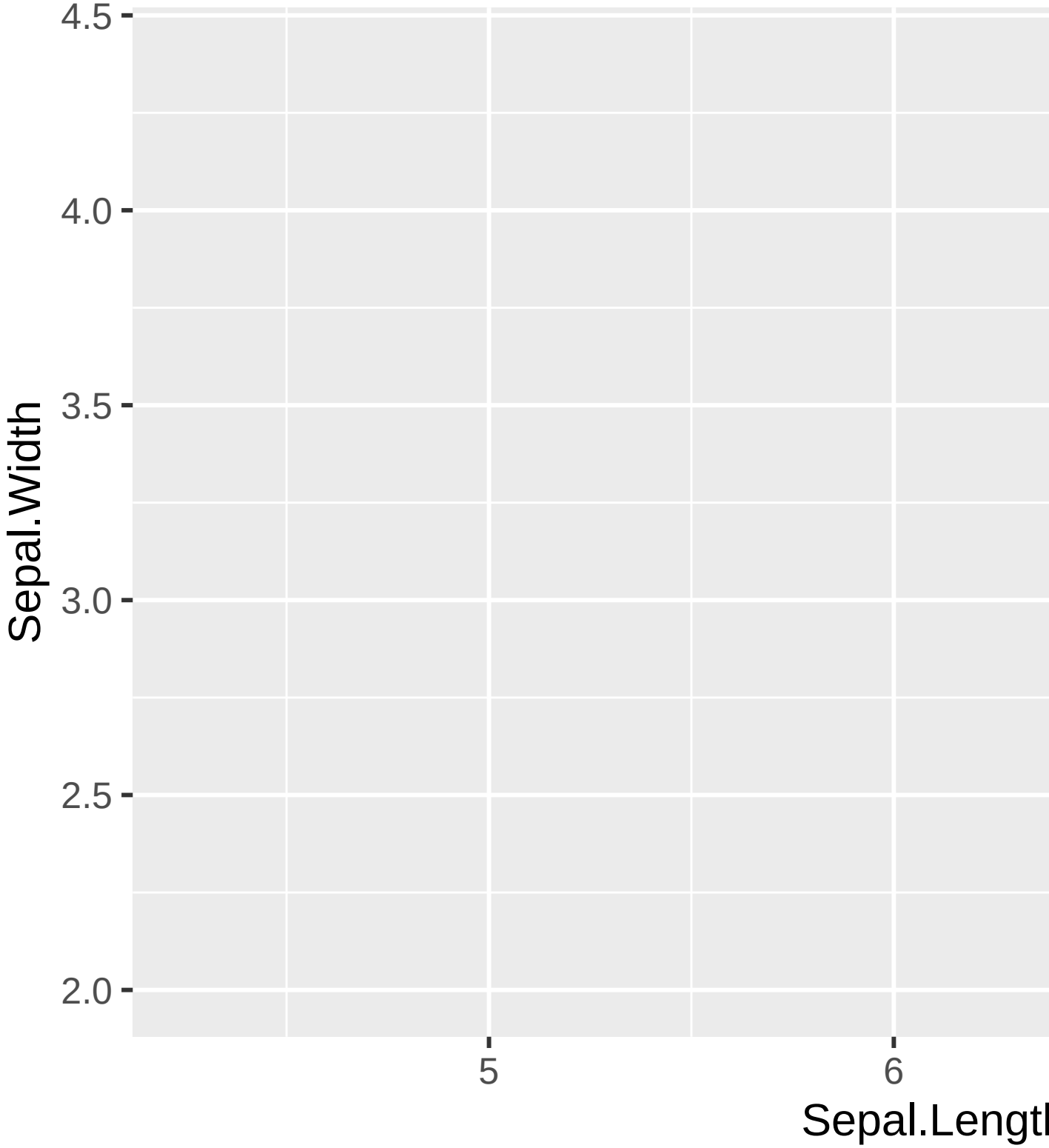
The next grammatical element is the aesthetic layer, or **aes** for short. This layer specifies how we want to *map* our data onto the scales of the plot.

> **Note**
>
> Note that aesthetics refers to a mapping function. That is, how certain elements (e.g., color, shape, size, etc.) map onto variables in our data.
> You can also set specific values for these elements (not mapped onto any variables in the data). These are not typically referred to as aesthetics in the grammar of graphics.
> E.g., to make all points in a plot red: geom_point(color = "red")

Figure 17.5: ggplot2 aesthetic (aes) layer

The aesthetic layer *maps* variables in our data onto scales in our graphical visualization, such as the x and y coordinates. In `ggplot2` the aesthetic layer is specified using the `aes()` function. Let's create a plot of the relationship between Sepal.Length and Sepal.Width, putting them on the x and y axis respectively.

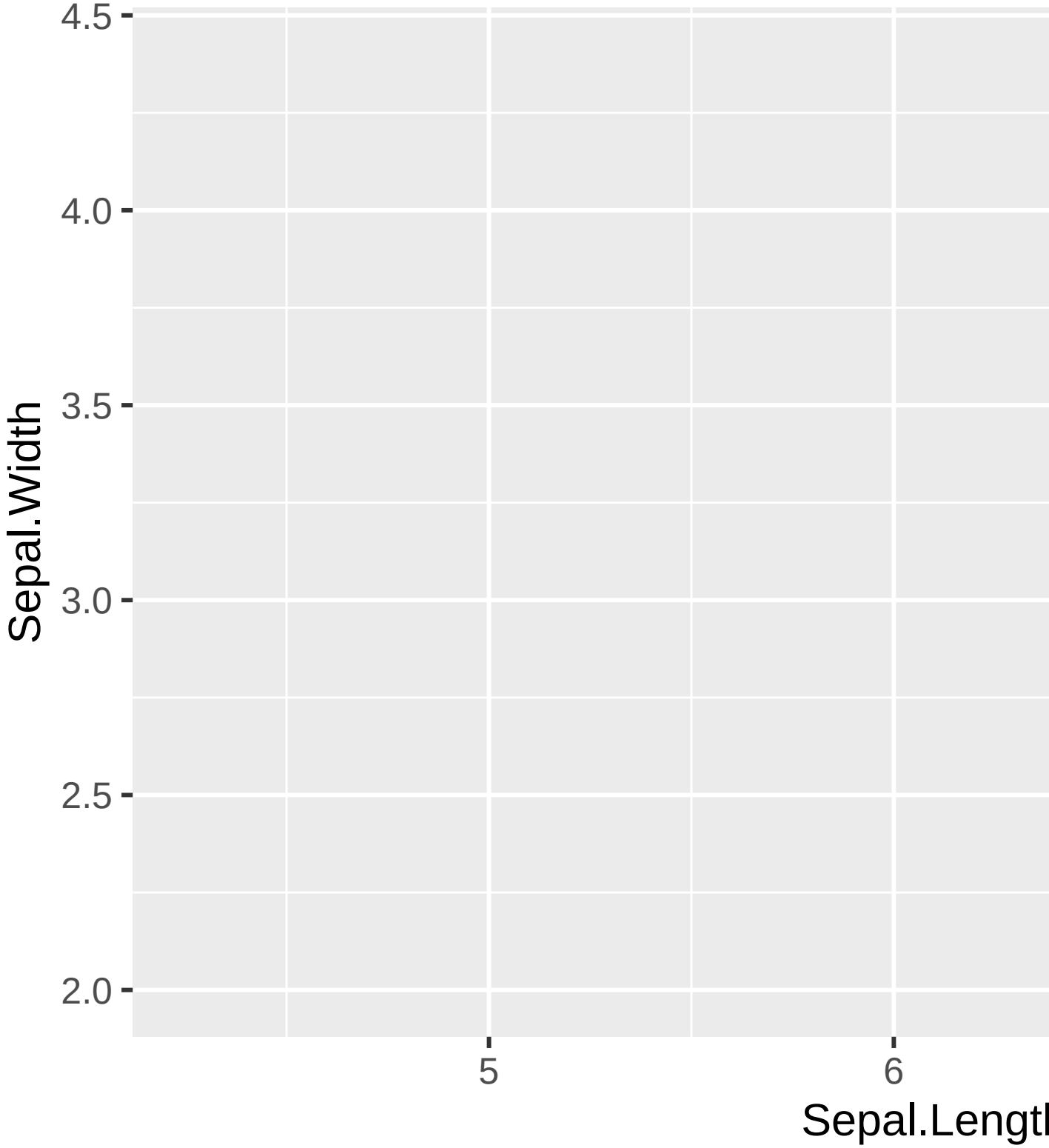
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
Sepal.Length

You can see we went from a blank box to a graph with the variable and scales of Sepal.Length mapped onto the x-axis and Sepal.Width on the y-axis.

The aeshetic layer also maps variables in our data to other elements in our graphical visualization, such as color, size, fill, etc.

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species))
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
Sepal.Length

17.5 Geometries Layer

The next *essential* element for data visualization is the geometries layer or **geom** layer for short.

Figure 17.6: ggplot2 geometrics (geom) layer

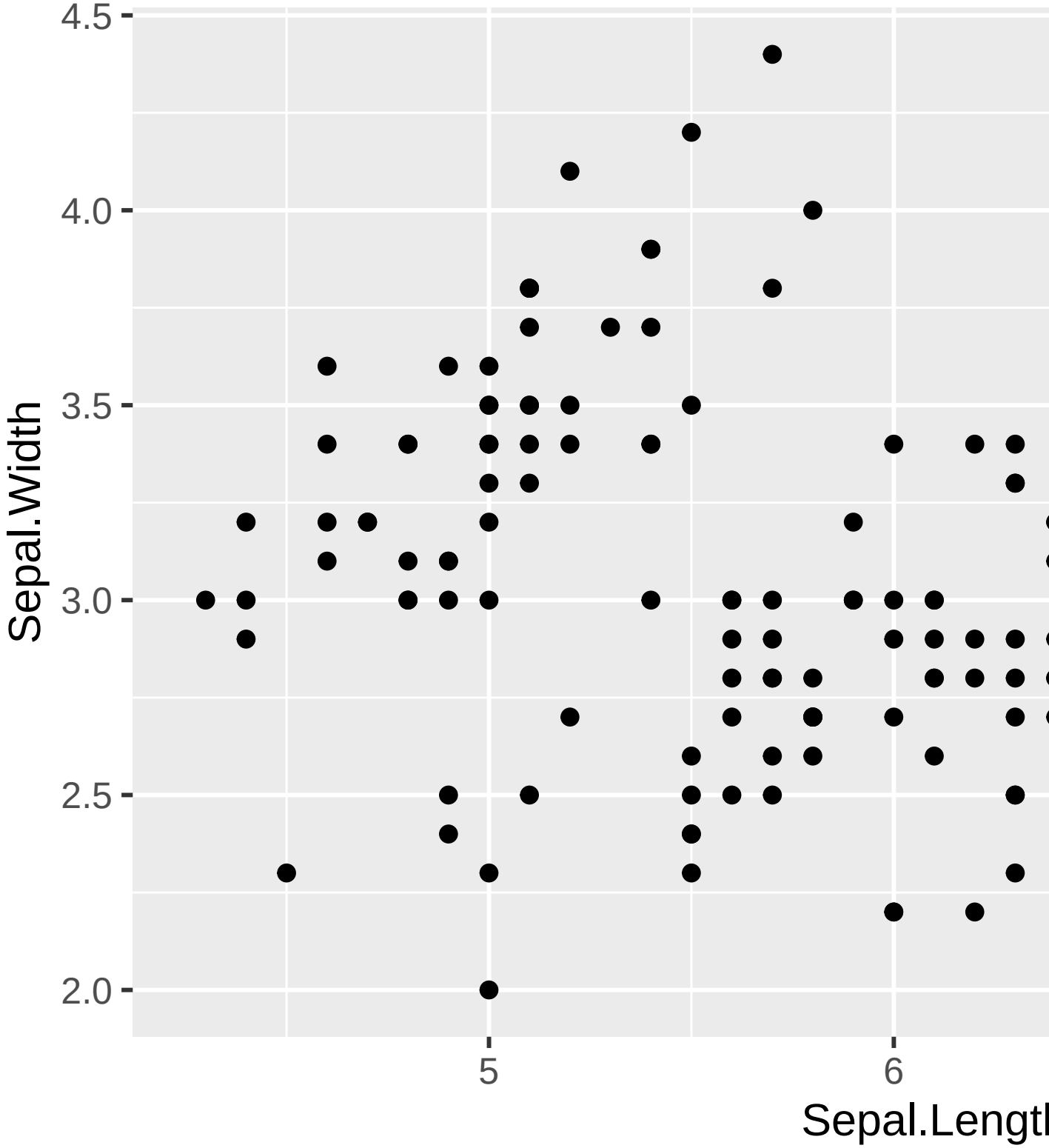
Just to demonstrate to you that `ggplot2` is creating R graphic objects that you can modify and not just static images, let's assign the previous graph with data and aesthetics layers only onto an R object called `p`, for plot.

```
p <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))
```

Now let's say we want to add the individual raw data points to create a scatterplot. To do this we can use the function `geom_point()`. This is a geom layer and the type of geom we want to add are points.

In `ggplot2` there is a special notation that is similar to the pipe operator `%>%` seen before. Except it is plus sign `+`

```
p + geom_point()
```

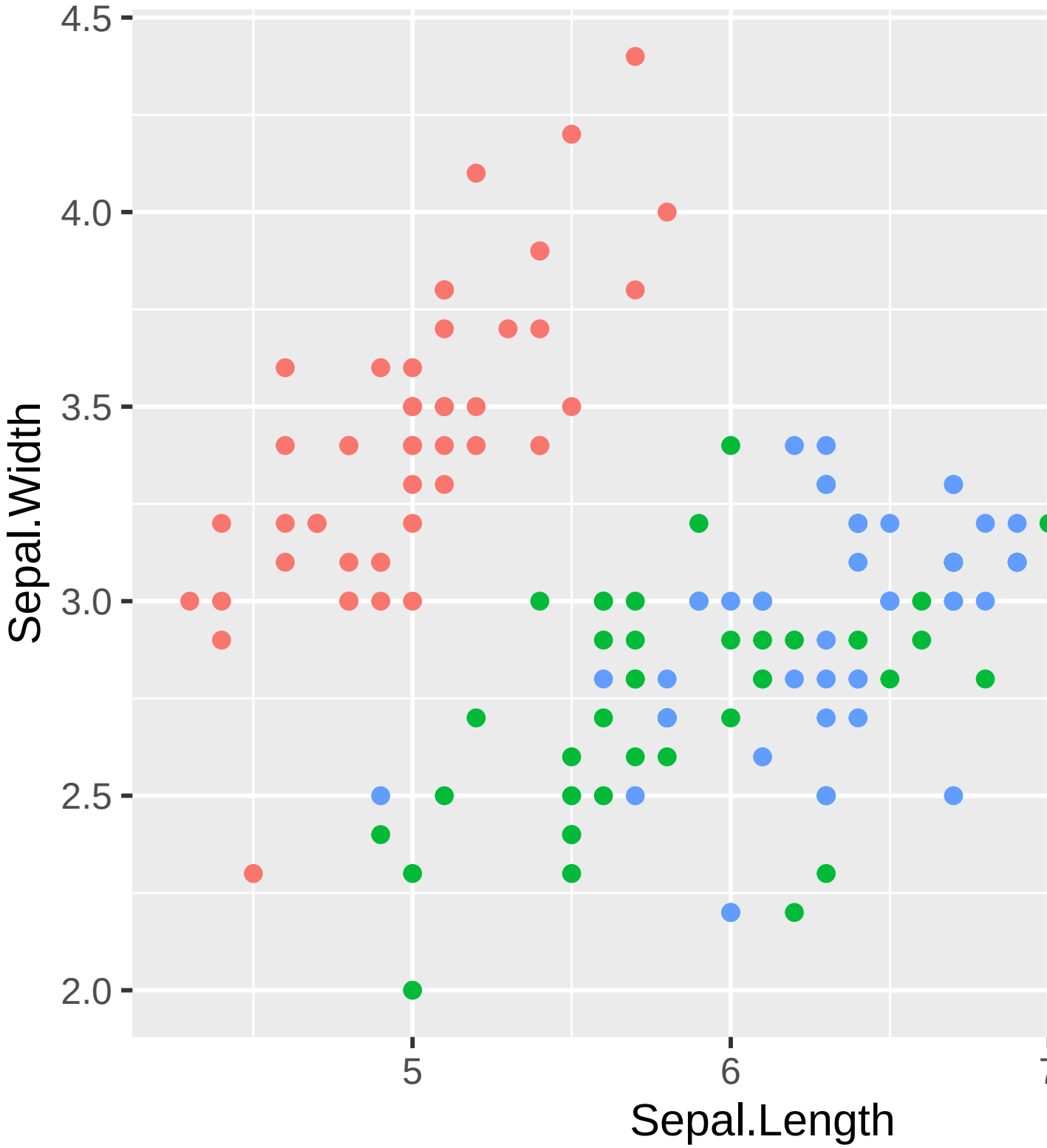

4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
Sepal.Length

And walla! Now we have a scatterplot of the relationship between Sepal.Length and Sepal.Width. Cool.

If we look at the scatterplot it appears that there are at least two groups or clusters of points. These clusters might represent the different species of flowers, represented in the `Species` column. There are different ways we can visualize or separate this grouping structure.

First, let's create an aesthetic to map the color of the points to the variable `Species` in our data.

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point()
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
Sepal.Length

Next, we will consider how to plot `Species` in separate plots within the same visualization.

17.6 Facets Layer

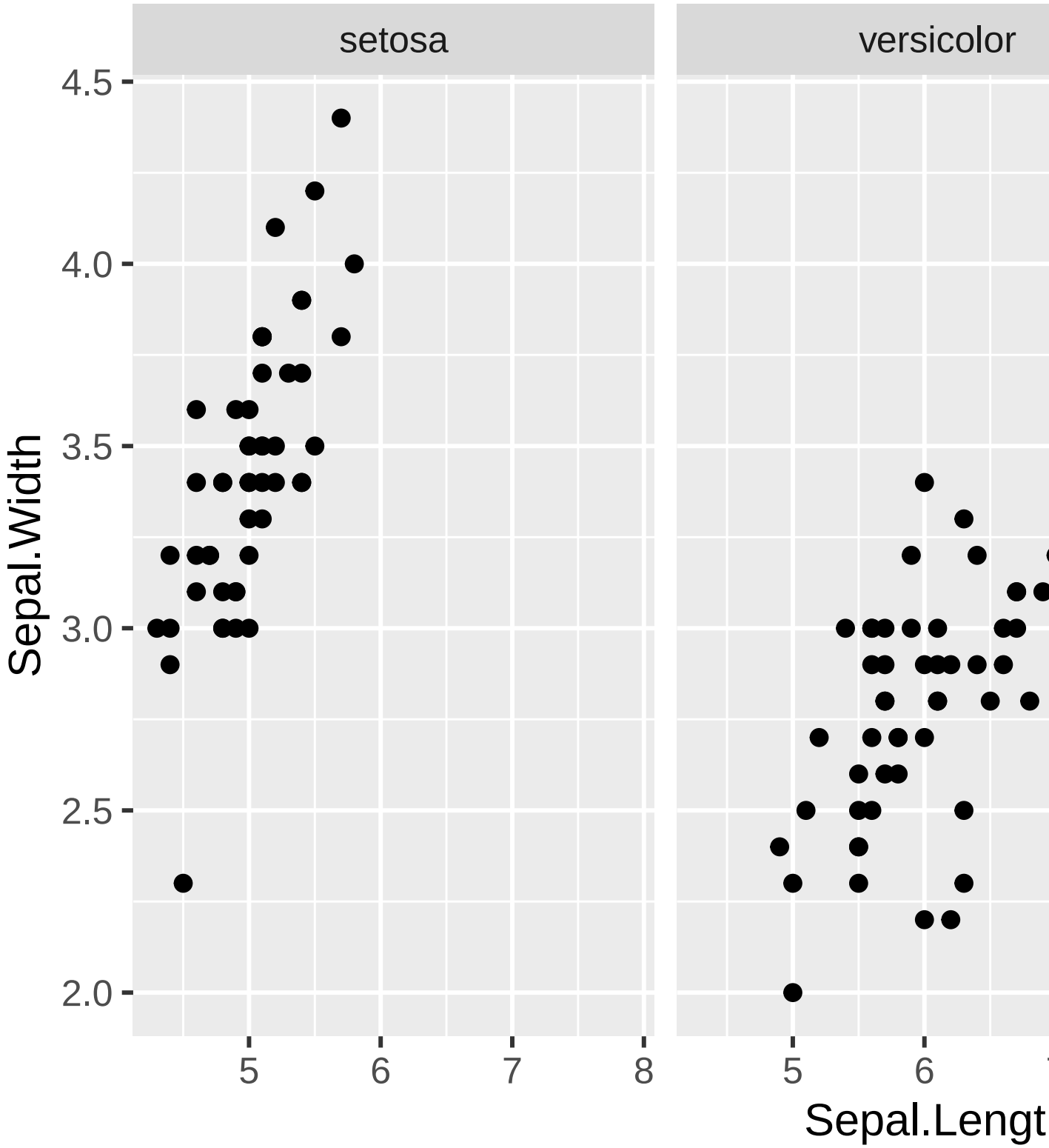
The facet layer allows you to create subplots within the same graphic object

Figure 17.7: ggplot2 facet layer

The previous three layers are the **essential** layers. The facet layer is not essential, however given your data you may find it helps you to explore or communicate your data.

Let's create facets of our scatterplot by `Species`

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_grid(~ Species)
```


setosa
versicolor
Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal.Lengt

17.7 Statistics Layer

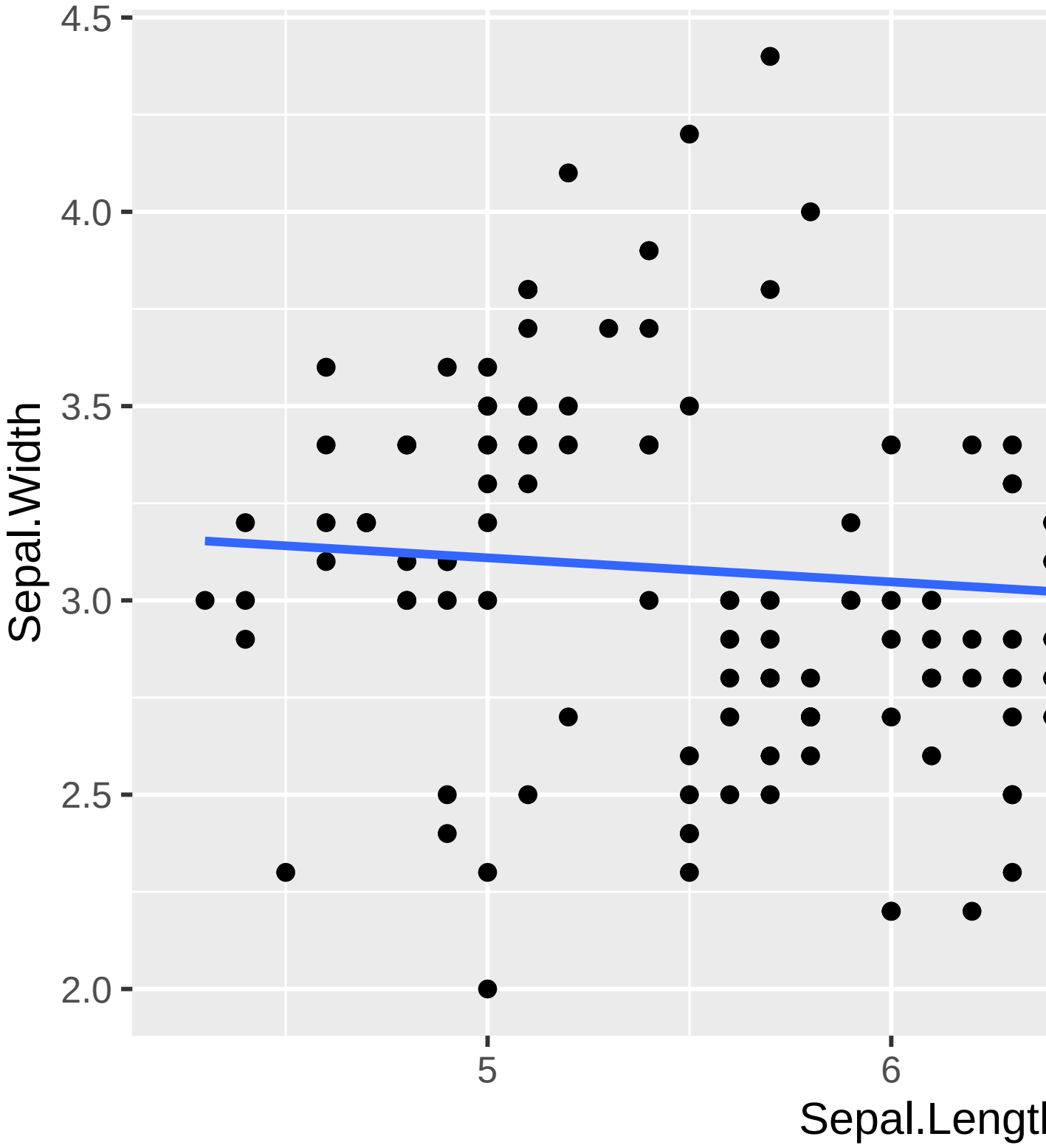
The statistics layer allows you plot statistical values calculated from the data

Figure 17.8: ggplot2 statistics (stat) layer

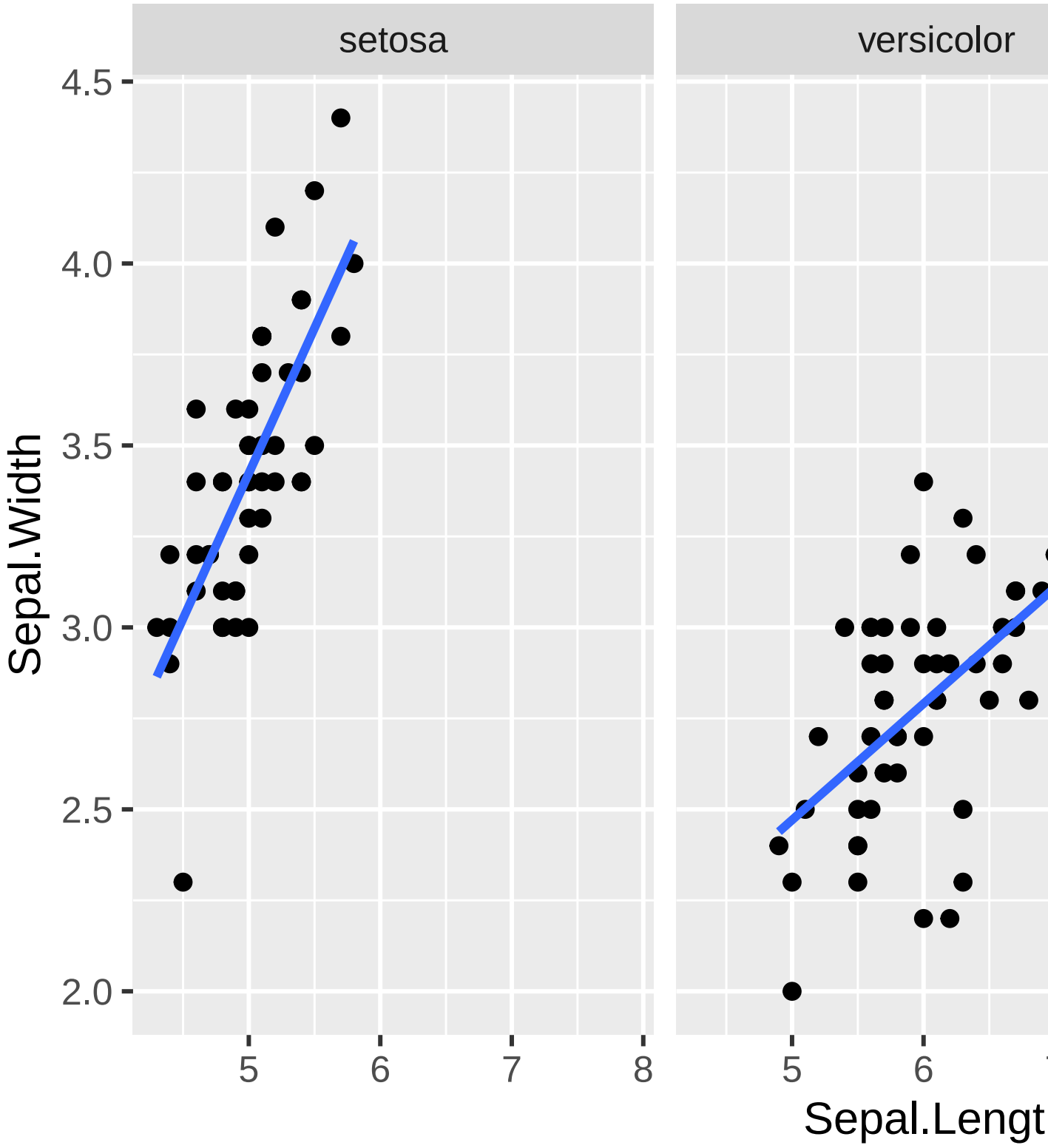
So far we have only plotted the raw data values. However, we may be interested in plotting some statistics or calculated values, such as a regression line, means, standard error bars, etc.

Let's add a regression line to the scatterplot. First without the facet layer then with the facet layer

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE)
## `geom_smooth()` using formula = 'y ~ x'
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
Sepal.Length

```r
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_grid(~ Species) +
  stat_smooth(method = "lm", se = FALSE)
## `geom_smooth()` using formula = 'y ~ x'
```


setosa
versicolor
Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
5
6
Sepal.Lengt

17.8 Coordinates Layer

The coordinate layer allows you to adjust the x and y coordinates

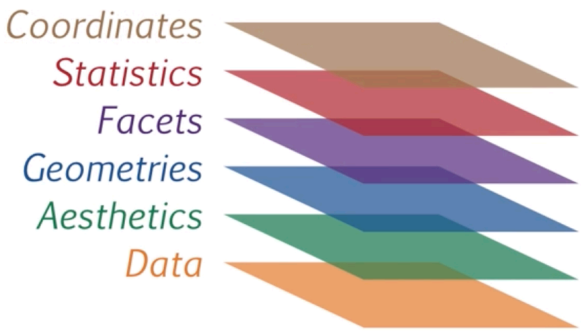


Figure 17.9: ggplot2 coordinates (coord) layer

There are two main groups of functions that are useful for adjusting the x and y coordinates.

17.8.1 axis limits

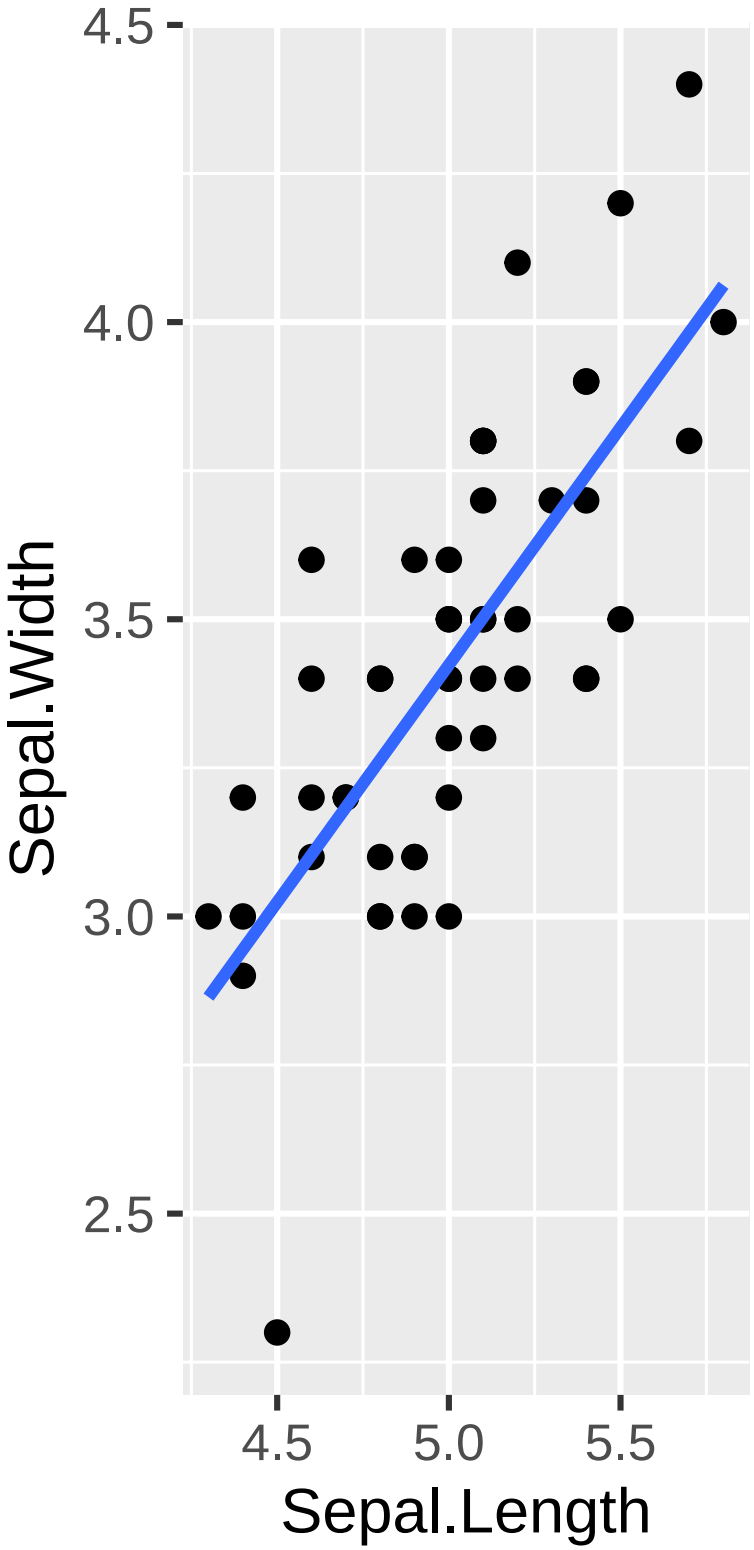
You can adjust limits (min and max) of the x and y axes using the `coord_cartesian(xlim = "", ylim = "")` function.

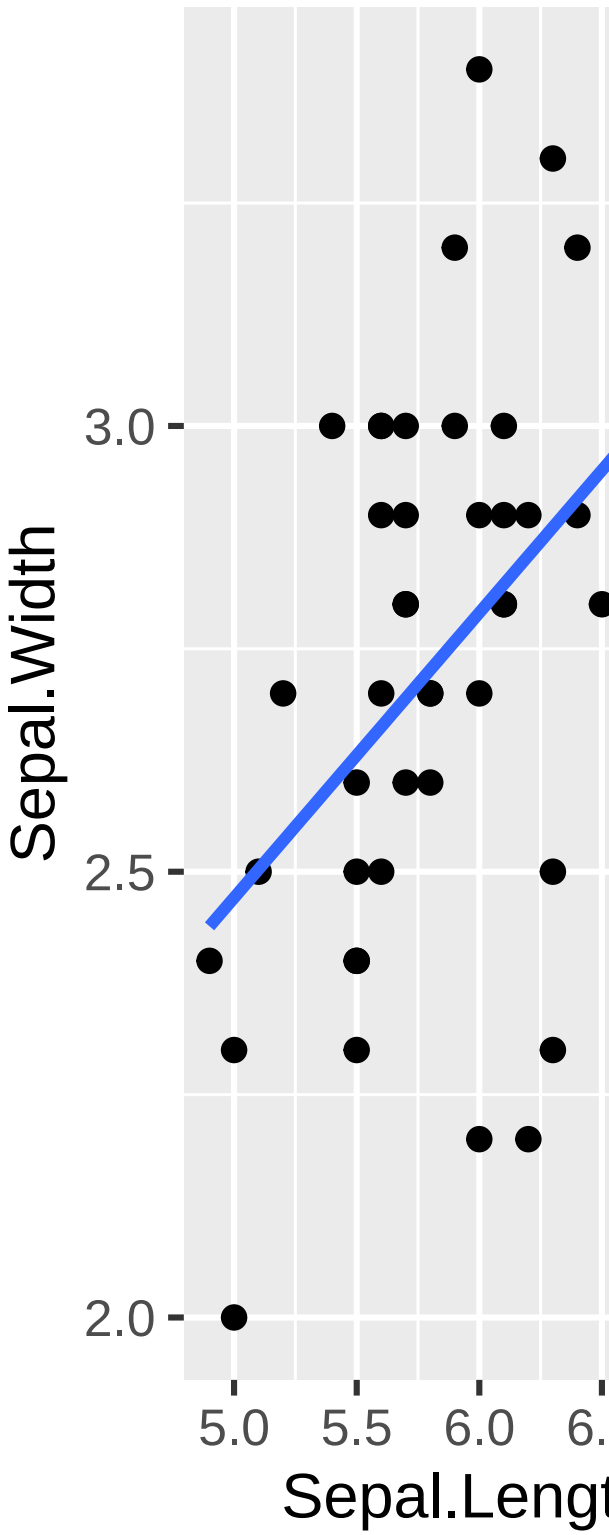
If you want to compare two separate graphs, then they need to be on the same scale! This is actually a very important design principle in data visualization.

Compare these two sets of plots:

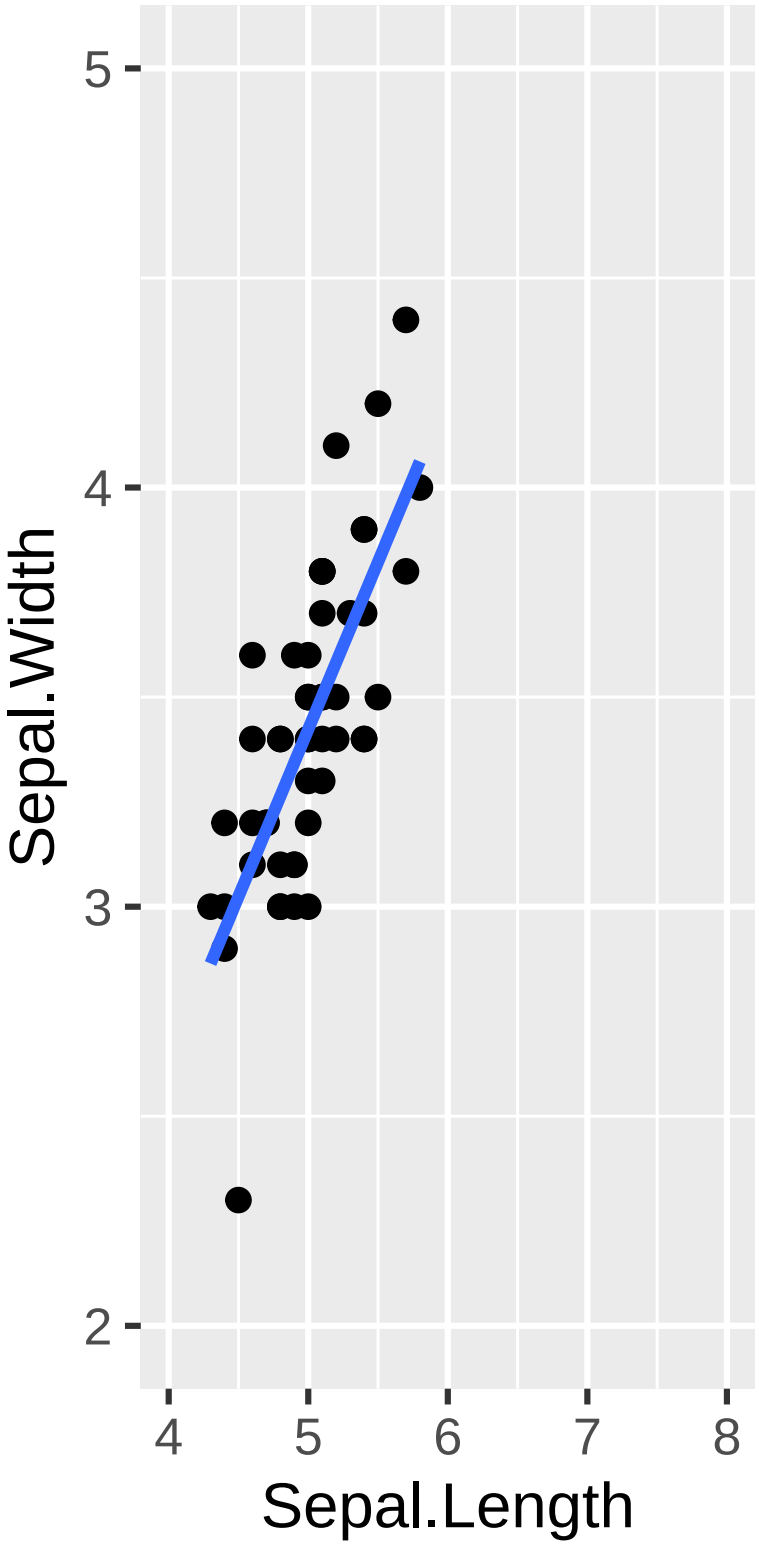
```
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
library(patchwork)
p1 <- ggplot(filter(iris, Species == "setosa"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE)
```

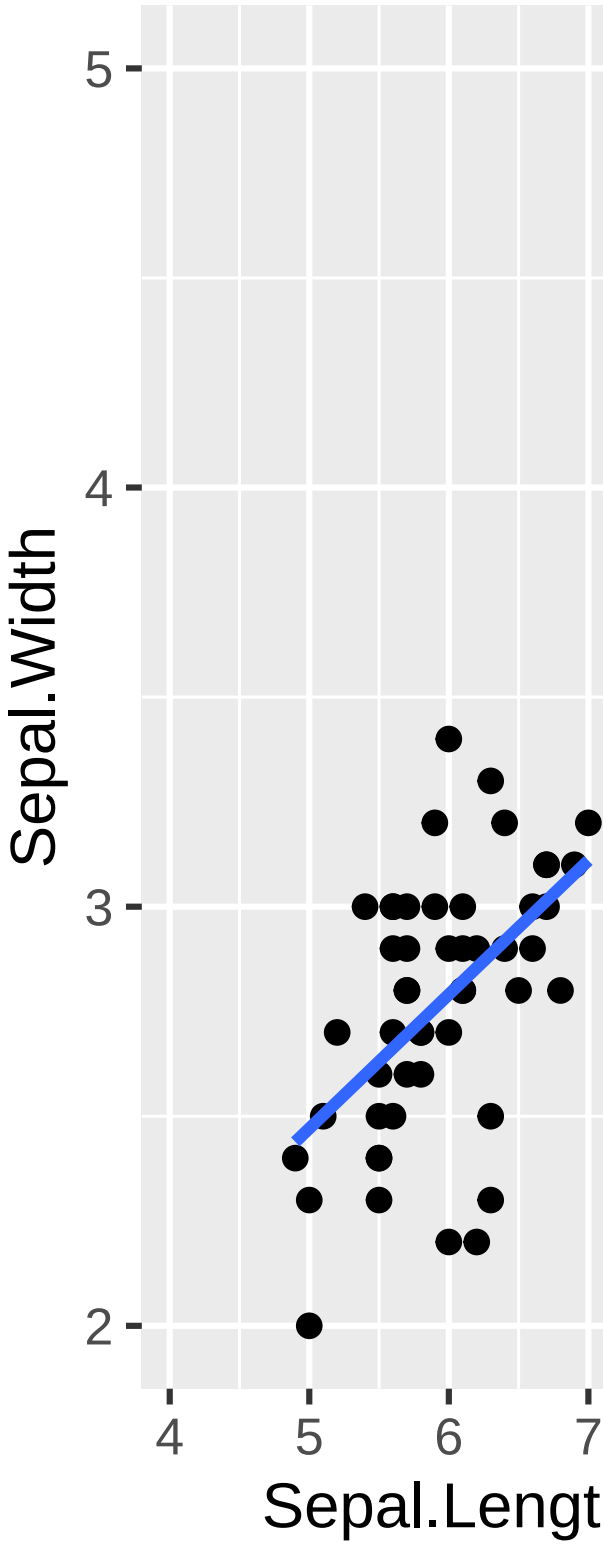
```r
p2 <- ggplot(filter(iris, Species == "versicolor"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE)

p3 <- ggplot(filter(iris, Species == "virginica"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE)

p1 + p2 + p3
## `geom_smooth()` using formula = 'y ~ x'
## `geom_smooth()` using formula = 'y ~ x'
## `geom_smooth()` using formula = 'y ~ x'
```


4.5
4.0
3.5
3.0
2.5
Sepal.Width
4.5
5.0
5.5
Sepal.Length


3.0
2.5
2.0
Sepal.Width
5.0
5.5
6.0
Sepal.Lengt

```r
library(dplyr)
library(patchwork)

p1 <- ggplot(filter(iris, Species == "setosa"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  coord_cartesian(xlim = c(4, 8), ylim = c(2, 5))

p2 <- ggplot(filter(iris, Species == "versicolor"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  coord_cartesian(xlim = c(4, 8), ylim = c(2, 5))

p3 <- ggplot(filter(iris, Species == "virginica"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  coord_cartesian(xlim = c(4, 8), ylim = c(2, 5))

p1 + p2 + p3
## `geom_smooth()` using formula = 'y ~ x'
## `geom_smooth()` using formula = 'y ~ x'
## `geom_smooth()` using formula = 'y ~ x'
```


5
4
3
2
Sepal.Width
4
5
6
7
8
Sepal.Length


5
4
3
2
Sepal.Width
4
5
6
7
Sepal.Lengt

> **Note**
>
> The `patchwork` package was used to easily create plot of two separate plots side-by-side. The `patchwork` package is excellent for arranging and combining multiple plots into one figure in all sorts of configurations.

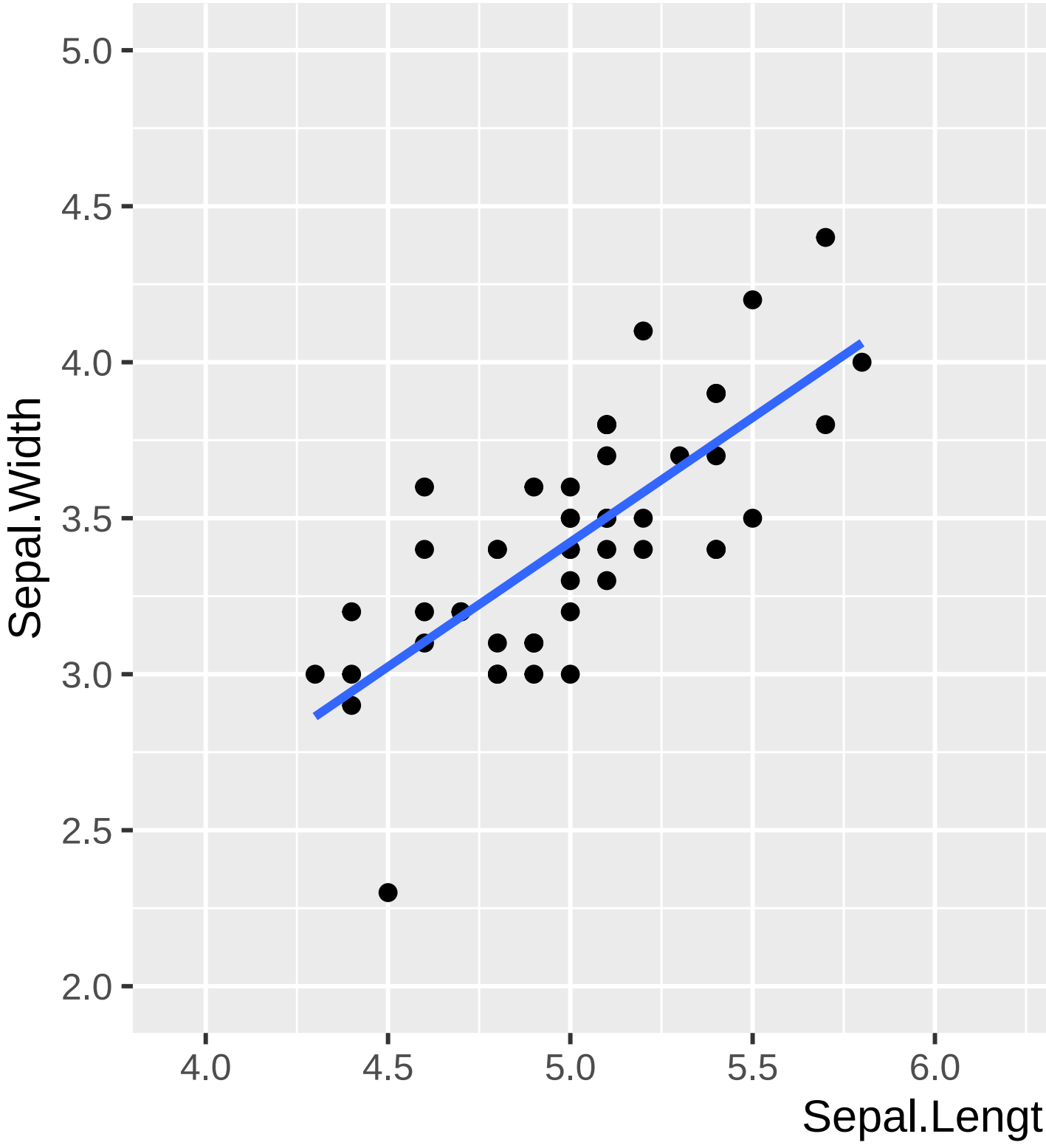
17.8.2 axis ticks and labels

You can adjust the scale (major and minor ticks) of the x and y axes using the `scale_x_` and `scale_y_` sets of functions. The two main sets of functions to know are for continuous and discrete scales:

- continuous: `scale_x_continuous(breaks = seq())` and `scale_y_continuous(breaks = seq())`
- discrete: `scale_x_discrete(breaks = c())` and `scale_y_continuous(breaks = c())`

For example:

```
ggplot(filter(iris, Species == "setosa"),
             aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  coord_cartesian(xlim = c(4, 8), ylim = c(2, 5)) +
  scale_x_continuous(breaks = seq(4, 8, by = .5)) +
  scale_y_continuous(breaks = seq(2, 5, by = .5))
## `geom_smooth()` using formula = 'y ~ x'
```


5.0
4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
4.0
4.5
5.0
5.5
6.0
Sepal.Lengt

> **Warning**
>
> It is advisable to set the limits of `breaks =` to be the same as the `xlim` and `ylim` specified in `coord_cartesian()`

17.9 Themes Layer

The Themes Layer refers to any visual elements not mapped to data variables

Figure 17.10: ggplot2 themes layer

You can change the labels of x or y axis, add a plot title, modify a legend title, add text anywhere on the plot, change the background color, axis lines, plot lines, etc.

There are three types of elements within the Themes Layer; text, line, and rectangle. Together these three elements can control all the non-data ink in the graph. Underneath these three elements are sub-elements and this can be represented in a hierarchy such as:

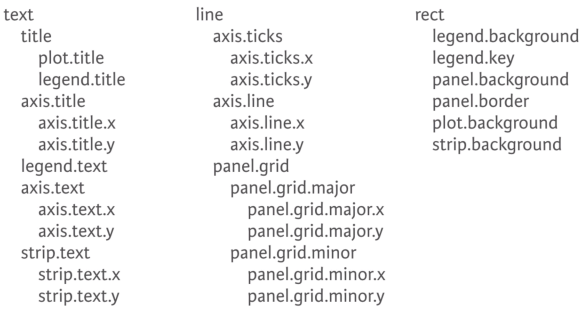


Figure 17.11: ggplot2 theme elements

For instance, you can see that you can control the design of the text for the plot title and legend title `theme(title = element_text())` or individually with `theme(plot.title = element_text(), legend.title = element_text())`.

- Any text element can be modified with `element_text()`
- Any line element can be modified with `element_line()`
- Any rect element can be modified with `element_rect()`

You can then control different features such as the color, linetype, size, font family, etc.

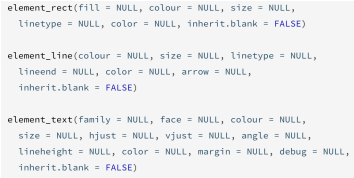
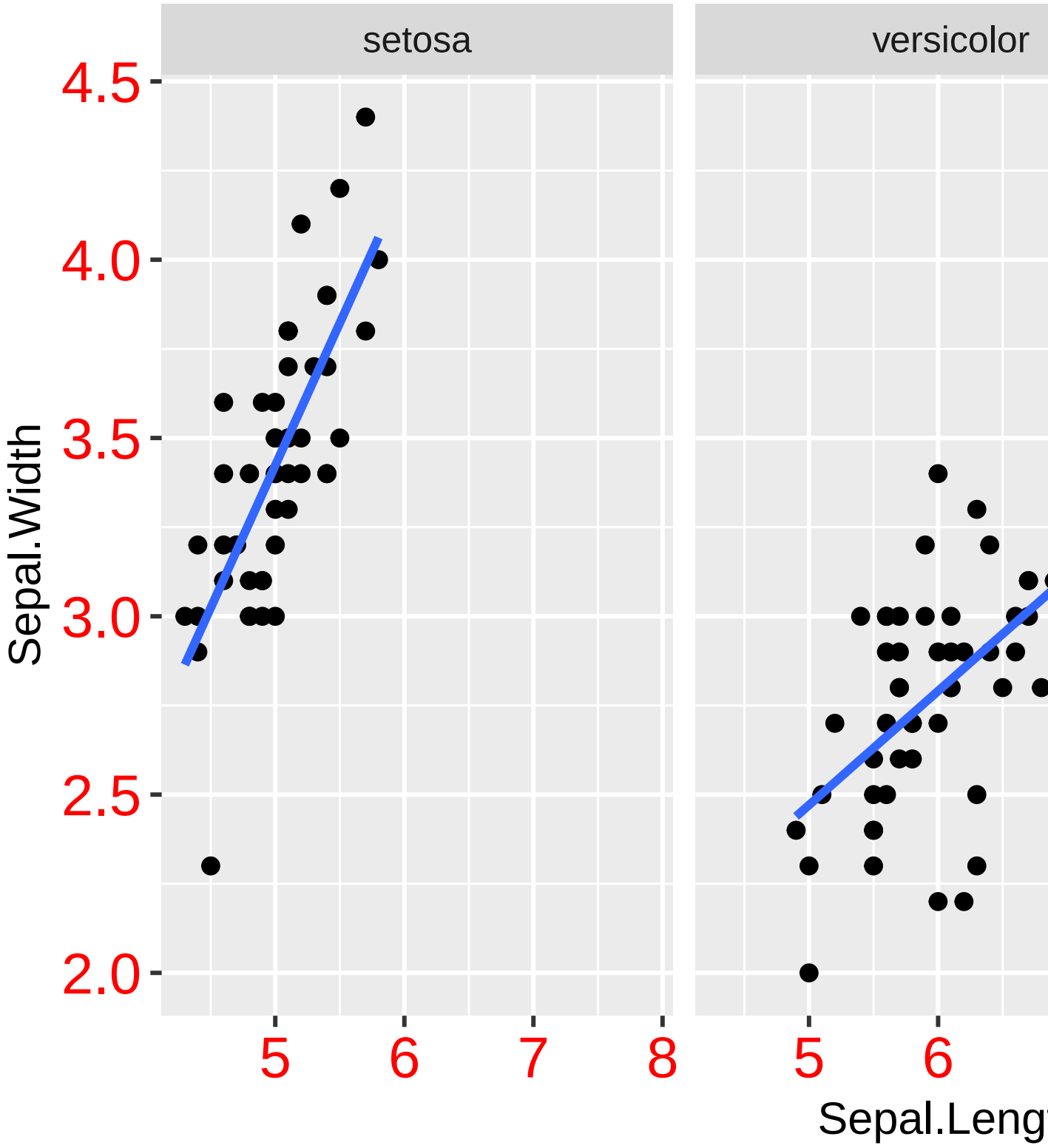


Figure 17.12: ggplot2 theme functions

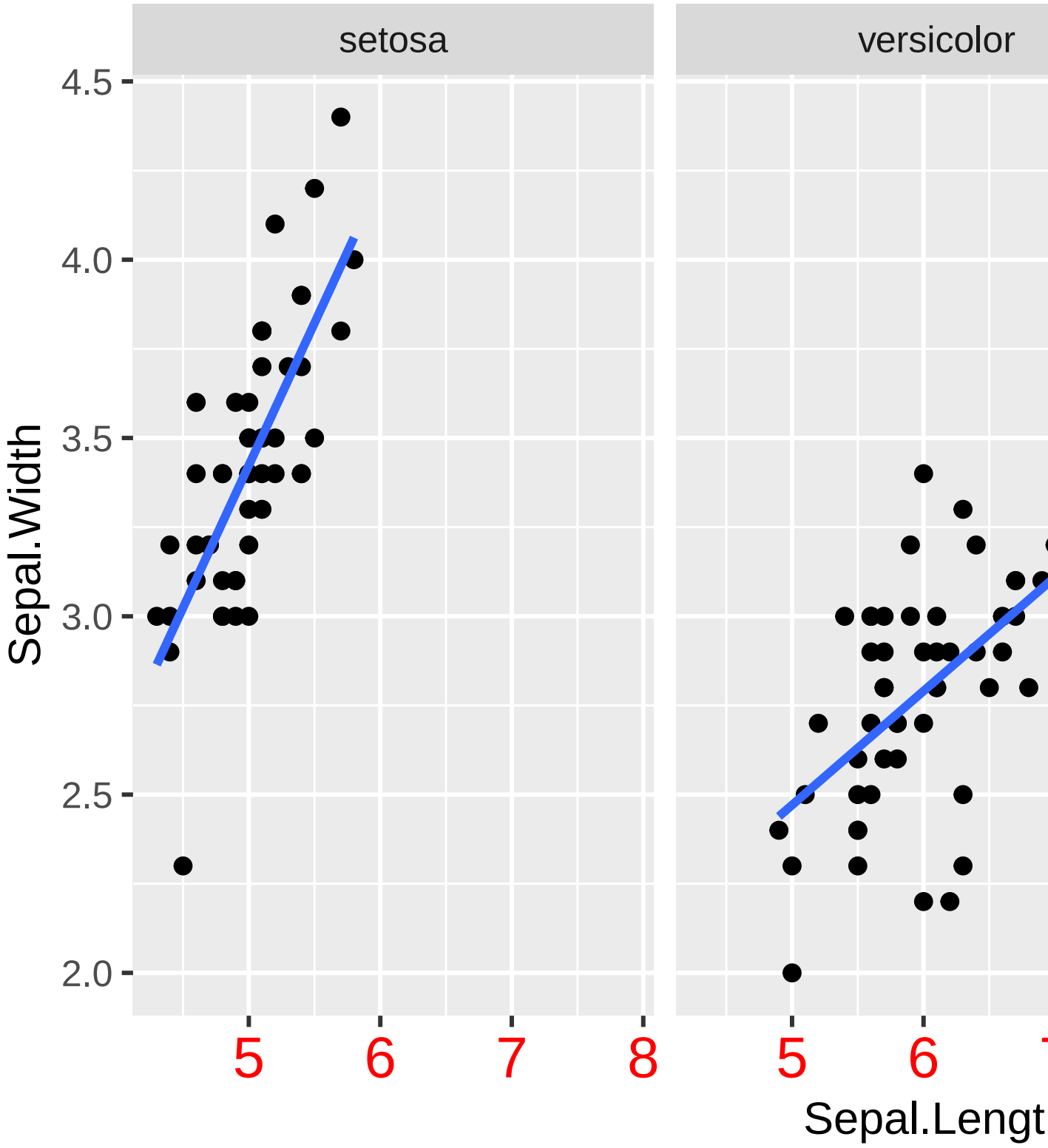
As an example let's change some theme elements to our facet plot. Let's change the axis value labels to **red** font and increase the **size**

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  stat_smooth(method = "lm", se = FALSE) +
  theme(axis.text = element_text(color = "red", size = 14))
## `geom_smooth()` using formula = 'y ~ x'
```


setosa
versicolor
Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
5
6
Sepal.Leng

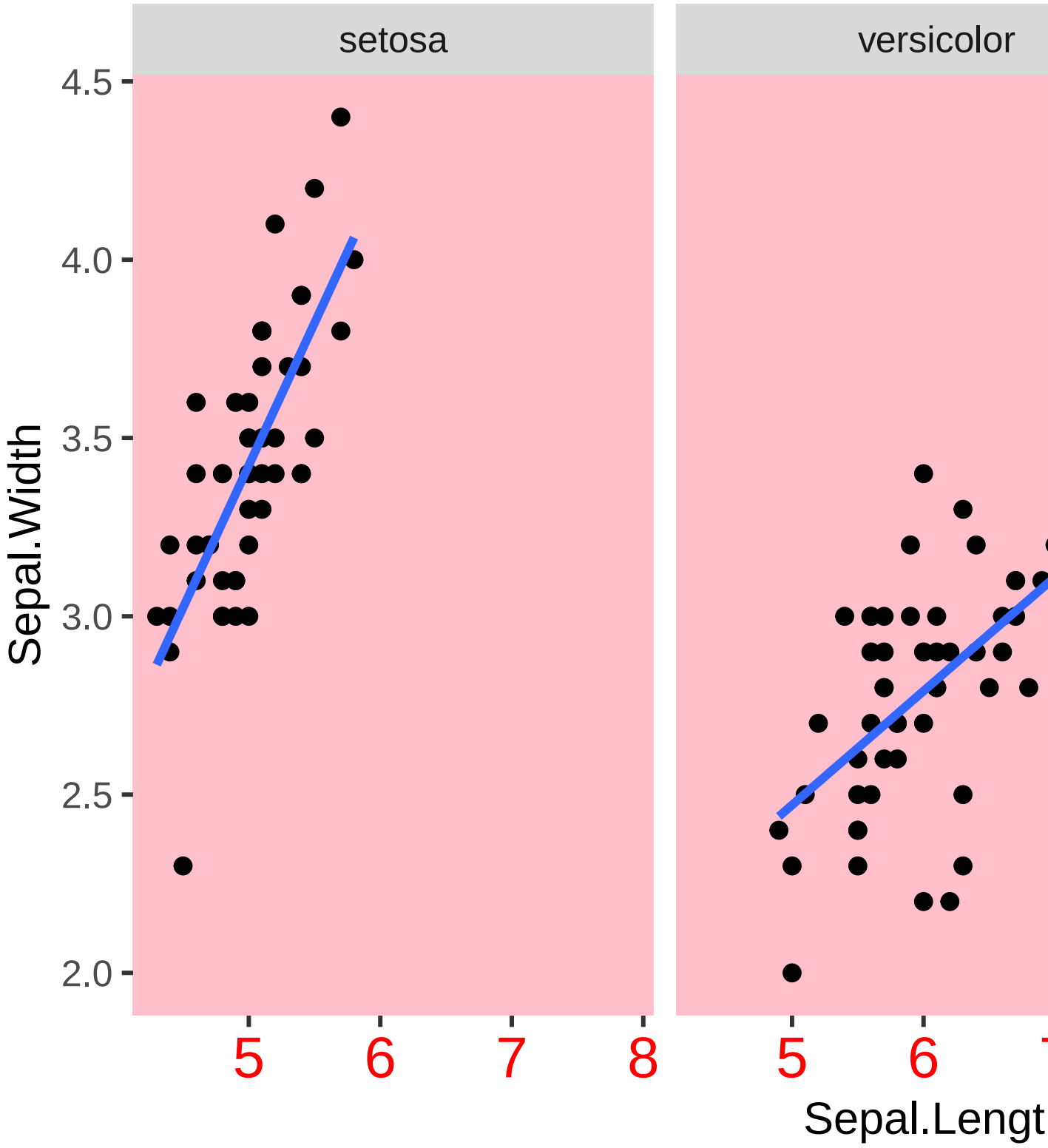
Now let's only change the x-axis text and not the y-axis text.

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  stat_smooth(method = "lm", se = FALSE) +
  theme(axis.text.x = element_text(color = "red", size = 14))
## `geom_smooth()` using formula = 'y ~ x'
```


setosa
versicolor
Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
5
6
Sepal.Lengt

It is a good idea to have a consistent theme across all your graphs. And so you might want to just create a theme object that you can add to all your graphs.

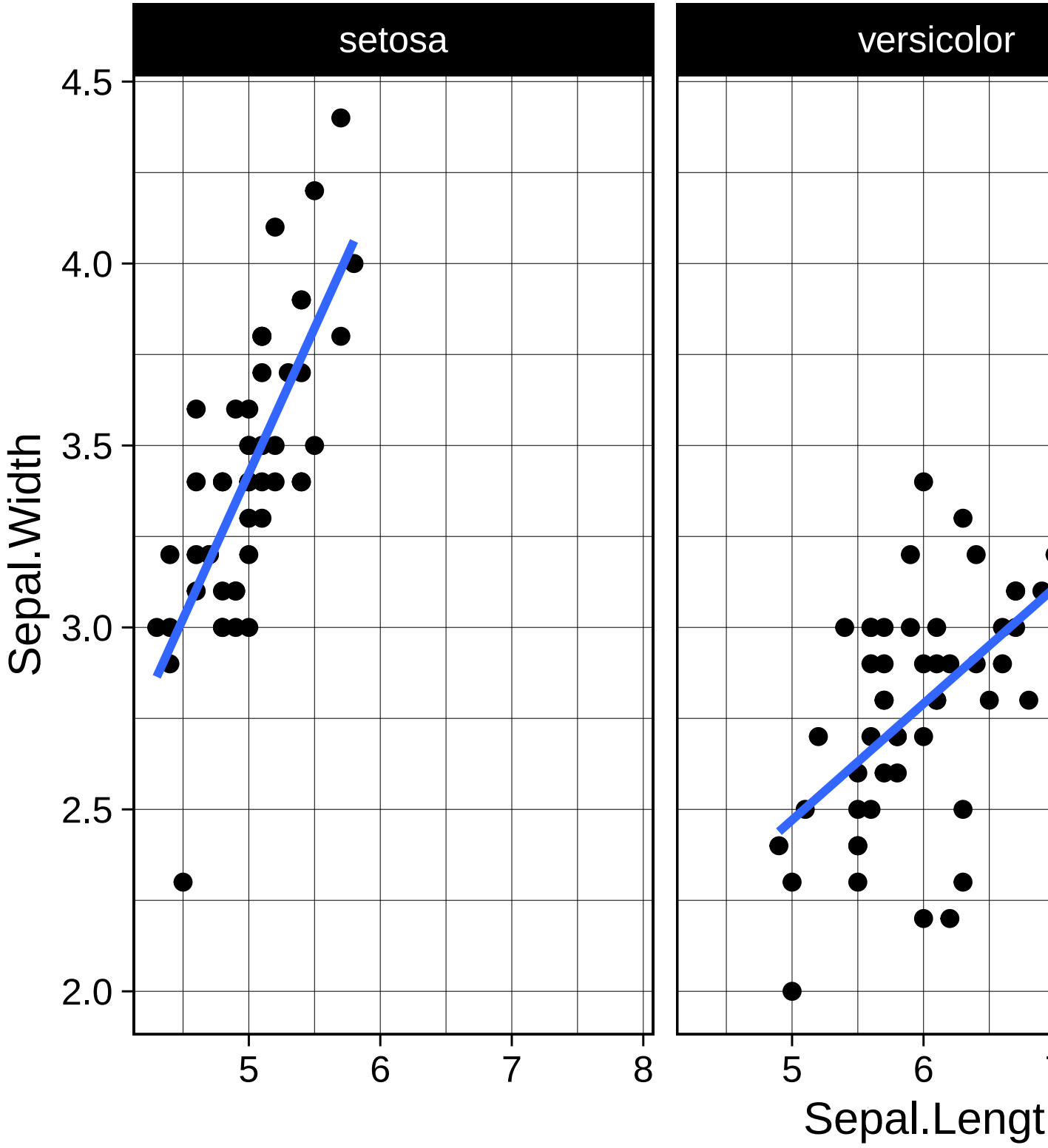
```
a_theme <- theme(axis.text.x = element_text(color = "red", size = 14),
                 panel.grid = element_blank(),
                 panel.background = element_rect(fill = "pink"))

ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  stat_smooth(method = "lm", se = FALSE) +
  theme(axis.text.x = element_text(color = "red", size = 14)) +
  a_theme
## `geom_smooth()` using formula = 'y ~ x'
```


setosa
versicolor
Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
5
6
Sepal.Lengt

17.9.1 Built-in Themes

For the most part you can probably avoid the `theme()` function by using built-in themes, unless there is a specific element you want to modify.

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  stat_smooth(method = "lm", se = FALSE) +
  theme(axis.text.x = element_text(color = "red", size = 14)) +
  theme_linedraw()
## `geom_smooth()` using formula = 'y ~ x'
```


setosa
versicolor
Sepal.Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
5
6
Sepal.Lengt

You can also set a default theme for the rest of your ggplots at the top of your script. That way you do not have to keep on specifying the theme for evey ggplot.

```
theme_set(theme_linedraw())
```

Now you can create a ggplot with `theme_linedraw()` without specifying `theme_linedraw()` every single time.

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species) +
  stat_smooth(method = "lm", se = FALSE)
```

You can do a google search to easily find different types of theme templates.