STA 444/5 - Data wrangling using R

Derek L. Sonderegger

July 02, 2019

Contents

Preface

This book is intended to provide students with a resource for learning R while using it during an introductory statistics course. The Introduction section covers common issues that students in a typical statistics course will encounter and provides a simple examples and does not attempt to be exhaustive. The Deeper Details section addresses issues that commonly arise in many data wrangling situations and is intended to give students a deep enough understanding of R that they will be able to use it as their primary computing resource to manipulate, graph and model data.

Acknowledgements

Finally, I am deeply appreciative of the support given to me by my wife, Aubrey.

Introduction

Chapter 1

Familiarization

R is a open-source program that is commonly used in statistics and machine learning. It runs on almost every platform and is completely free and is available at www.r-project.org. Most cutting-edge statistical research is first available on R.

The basic editor that comes with R works fairly well, but you should consider running R through the program RStudio which is located at rstudio.com. This is a completely free Integrated Developement Environment that works on Macs, Windows and a couple of flavors of Linux. It simplifies a bunch of more annoying aspects of the standard R GUI and supports things like tab completion.

R is a script based language, and there isn't a point-and-click interface for data wrangling and statistical modeling. While the initial learning curve will be steeper, understanding how to write scripts will be valuable because scripts leave a clear description of what steps were performed. This is a critical aspect of what is known as *reproducable research* and a good practice.

While it may seem tempting to type commands into the console directly, but because the goal is to create a script that contains all of the necessary commands to perform an analysis, users should get into the habit of always writing their commands into their R script (or Rmarkdown file) and executing the command from there.

1.1 R file Types

One of the worst things about a pocket calculator is there is no good way to go several steps and easily see what you did or fix a mistake (there is nothing more annoying than re-typing something because of a typo. To avoid these issues I always work with RMarkdown (or script) files instead of typing directly into the console. You will quickly learn that it is impossible to write R code correctly the first time and you'll save yourself a huge amount of work by just embracing this from the beginning. Furthermore, having an R file fully documents how you did your analysis, which can help when writing the methods section of a paper. Finally, having a file makes it easy to re-run an analysis after a change in the data (additional data values, transformed data, or removal of outliers).

There are three common ways to store R commands in some file: scripts, notebooks, and Rmarkdown files. The distinction between the R scripts and the other two is substantial as R scripts just store R commands, but don't make any attampt to save the results in any distinct format. Both notebooks and Rmarkdown files save the results of an analysis and present the results in a nice readable fashion. I encourage people to use Rmarkdown files over notebooks because the Rmarkdown knitting enforces a reproducable workflow. Rmarkdown files are written in a way to combine the R commands, commentary, and the command outputs all together into one coherent document. For most people that use R to advance their research, using Rmarkdown is the most useful.

1.1.1 R Scripts (.R files)

The first type of file that we'll discuss is a traditional script file. To create a new .R script in RStudio go to `File -> New File -> R Script`. This opens a new window in RStudio where you can type commands and functions as a common text editor. Type whatever you like in the script window and then you can execute the code line by line (using the run button or its keyboard shortcut to run the highlighted region or whatever line the curser is on) or the entire script (using the source button). Other options for what piece of code to run are available under the Code dropdown box.

It often makes your R files more readable if you break a single command up into multiple lines. R scripts will disregard all whitespace (including line breaks) so you can safely spread your command over as multiple lines. Finally, it is useful to leave comments in the script for things such as explaining a tricky step, who wrote the code and when, or why you chose a particular name for a variable. The `#` sign will denote that the rest of the line is a comment and R will ignore it.

An R script for a homework assignment might look something like this:

```r
# Problem 1
# Calculate the log of a couple of values and make a plot
# of the log function from 0 to 3
log(0)
log(1)
log(2)
x <- seq(.1,3, length=1000)
plot(x, log(x))

# Problem 2
# Calculate the exponential function of a couple of values
# and make a plot of the function from -2 to 2
exp(-2)
exp(0)
exp(2)
x <- seq(-2, 2, length=1000)
plot(x, exp(x))
```

This looks perfectly acceptable as a way of documenting what you did, but this script file doesn't contain the actual results of commands I ran, nor does it show you the plots. Also anytime I want to comment on some output, it needs to be offset with the commenting character `#`. It would be nice to have both the commands and the results merged into one document. This is what the R Markdown file does for us.

1.1.2 R Markdown (.Rmd files)

When I was a graduate student, I had to tediously copy and past tables of output from the R console and figures I had made into my Microsoft Word document. Far too often I would realize I had made a small mistake in part (b) of a problem and would have to go back, correct my mistake, and then redo all the laborious copying. I often wished that I could write both the code for my statistical analysis and the long discussion about the interpretation all in the same document so that I could just re-run the analysis with a click of a button and all the tables and figures would be updated by magic. Fortunately that magic now exists.

To create a new R Markdown document, we use the `File -> New File -> R Markdown...` dropdown option and a menu will appear asking you for the document title, author, and preferred output type. In order to create a PDF, you'll need to have LaTeX installed, but the HTML output nearly always works and I've had good luck with the MS Word output as well.

The R Markdown is an implementation of the Markdown syntax that makes it extremely easy to write webpages and give instructions for how to do typesetting sorts of things. This syntax was extended to allow use to embed R commands directly into the document. Perhaps the easiest way to understand the syntax is to look at an at the RMarkdown website.

The R code in my document is nicely separated from my regular text using the three backticks and an instruction that it is R code that needs to be evaluated. The output of this document looks good as a HTML, PDF, or MS Word document. I have actually created this entire book using RMarkdown. To see what the the Rmarkdown file looks like for any chapter, just click on the pencil icon at the top of the online notes.

While writing an Rmarkdown file, each of the code chunks can be executed in a couple of different ways.

```
1. Press the green arrow at the top of the code chunk to run the entire chunk.
2. The run button has several options has several options.
3. There are keyboard shortcuts, on the Mac it is Cmd-Return.
```

To insert a new code chunk, a user can type it in directly, use the green Insert button, or the keyboard shortcut.

To produce a final output document that you'll present to your boss/collegues/client where you want to combine the code, output, and commentary you'll "knit" the document which causes all of the R code to be run in a new R session, and then weave together the output into your document. This can be done using the knit button at the top of the Editor Window.

1.1.3 R Notebooks (.Rmd files)

Notebooks are just very specialized types of Rmarkdown file. Here, the result of each code chunk that is run manually is saved, but when previewing the output, all of the R code is NOT re-run. Therefore it is possible to run the code, then modify the code, and then produce a document where the written code and output do not match up. As a result of this "feature" I strongly discourage the use of notebooks in favor of the standard Rmarkdown files.

1.2 R as a simple calculator

Assuming that you have started R on whatever platform you like, you can use R as a simple calculator. In either your Rmarkdown file code chunk (or just run this in the console), run the following

```r
# Some simple addition
2+3
```

```
## [1] 5
```

In this fashion you can use R as a very capable calculator.

```r
6*8
```

```
## [1] 48
```

```r
4^3
```

```
## [1] 64
```

```r
exp(1)   # exp() is the exponential function
```

```
## [1] 2.718282
```

R has most constants and common mathematical functions you could ever want. `sin()`, `cos()`, and other trigonometry functions are available, as are the exponential and log functions `exp()`, `log()`. The absolute value is given by `abs()`, and `round()` will round a value to the nearest integer.

```
pi     # the constant 3.14159265...
```

```
## [1] 3.141593
```

```
sin(0)
```

```
## [1] 0
```

```
log(5) # unless you specify the base, R will assume base e
```

```
## [1] 1.609438
```

```
log(5, base=10)  # base 10
```

```
## [1] 0.69897
```

Whenever I call a function, there will be some arguments that are mandatory, and some that are optional and the arguments are separated by a comma. In the above statements the function `log()` requires at least one argument, and that is the number(s) to take the log of. However, the base argument is optional. If you do not specify what base to use, R will use a default value. You can see that R will default to using base e by looking at the help page (by typing `help(log)` or `?log` at the command prompt).

Arguments can be specified via the order in which they are passed or by naming the arguments. So for the `log()` function which has arguments `log(x, base=exp(1))`. If I specify which arguments are which using the named values, then order doesn't matter.

```
# Demonstrating order does not matter if you specify
# which argument is which
log(x=5, base=10)
```

```
## [1] 0.69897
```

```
log(base=10, x=5)
```

```
## [1] 0.69897
```

But if we don't specify which argument is which, R will decide that `x` is the first argument, and `base` is the second.

```
# If not specified, R will assume the second value is the base...
log(5, 10)
```

```
## [1] 0.69897
```

```
log(10, 5)
```

```
## [1] 1.430677
```

When I specify the arguments, I have been using the `name=value` notation and a student might be tempted to use the `<-` notation here. Don't do that as the `name=value` notation is making an association mapping and not a permanent assignment.

1.3 Assignment

We need to be able to assign a value to a variable to be able to use it later. R does this by using an arrow `<-` or an equal sign `=`. While R supports either, for readability, I suggest people pick one assignment operator and stick with it. I personally prefer to use the arrow. Variable names cannot start with a number, may not include spaces, and are case sensitive.

```
tau <- 2*pi        # create two variables
my.test.var = 5    # notice they show up in 'Environment' tab in RStudio!
tau
```

```
## [1] 6.283185
```

```
my.test.var
```

```
## [1] 5
```

```
tau * my.test.var
```

```
## [1] 31.41593
```

As your analysis gets more complicated, you'll want to save the results to a variable so that you can access the results later. *If you don't assign the result to a variable, you have no way of accessing the result.*

1.4 Packages

One of the greatest strengths about R is that so many people have devloped add-on packages to do some additional function. For example, plant community ecologists have a large number of multivariate methods that are useful but were not part of R. So Jari Oksanen got together with some other folks and put together a package of functions that they found useful. The result is the package `vegan`.

To download and install the package from the Comprehensive R Archive Network (CRAN), you just need to ask RStudio it to install it via the menu `Tools` -> `Install Packages...`. Once there, you just need to give the name of the package and RStudio will download and install the package on your computer.

Many major analysis types are available via downloaded packages as well as problem sets from various books (e.g. `Sleuth3` or `faraway`) and can be easily downloaded and installed via the menu.

Once a package is downloaded and installed on your computer, it is available, but it is not loaded into your current R session by default. The reason it isn't loaded is that there are thousands of packages, some of which are quite large and only used occasionally. So to improve overall performance only a few packages are loaded by default and the you must explicitly load packages whenever you want to use them. You only need to load them once per session/script.

```
library(vegan)   # load the vegan library
```

For a similar performance reason, many packages do not automatically load their datasets unless explicitly asked. Therefore when loading datasets from a package, you might need to do a *two-step* process of loading the package and then loading the dataset.

```
library(faraway)       # load the package into memory
```

```
##
## Attaching package: 'faraway'
```

```
## The following object is masked from 'package:lattice':
##
##     melanoma
```

```
data("butterfat")      # load the dataset into memory
```

If you don't need to load any functions from a package and you just want the datasets, you can do it in one step.

```
data('butterfat', package='faraway')   # just load the dataset, not anything else
butterfat[1:6, ]                        # print out the first 6 rows of the data
```

```
##   Butterfat    Breed    Age
## 1      3.74 Ayrshire Mature
## 2      4.01 Ayrshire  2year
## 3      3.77 Ayrshire Mature
## 4      3.78 Ayrshire  2year
## 5      4.10 Ayrshire Mature
## 6      4.06 Ayrshire  2year
```

Similarly, if I am not using many functions from a package, I might choose call the functions using the notation `package::function()`. This is particularly important when two packages both have functions with the same name and it gets confusing which function you want to use. For example the packages `mosaic` and `dplyr` both have a function `tally`. So if I've already loaded the `dplyr` package but want to use the `mosaic::tally()` function I would use the following:

```
mosaic::tally( c(0,0,0,1,1,1,1,2) )
```

```
## X
## 0 1 2
## 3 4 1
```

1.5 Finding Help

There are many complicated details about R and nobody knows everything about how each individual package works. As a result, a robust collection of resources has been developed and you are undoubtably not the first person to wonder how to do something.

1.5.1 How does this function work?

If you know the function you need, but just don't know how to use it, the built-in documentation is really quite good. Suppose I am interested in how the `rep` function works. We could access the `rep` help page by searching in the help window or from the console via `help(rep)`. The document that is displayed shows what arguments the function expects and what it will return. At the bottom of the help page is often a set of examples demonstrating different ways to use the function. As you get more proficient in R, these help files become quite handy, but initially they feel quite overwhelming.

1.5.2 How does this package work?

If a package author really wants their package to be used by a wide audience, they will provide a "vignette". These are a set of notes that explain enough of how a package works to get a user able to utilize the package effectively. This documentation is targetted towards people the know some R, but deep technical knowledge is not expected. Whenever I encounter a new package that might be applicable to me, the first thing I do is see if it has a vignette, and if so, I start reading it. If a package doesn't have a vignette, I'll google "R package XXXX" and that will lead to documentation on CRAN that gives a list of functions in the package.

1.5.3 How do I do XXX?

Often I find myself asking how to do something but I don't know the function or package to use. In those cases, I will use the coding question and answer site stackoverflow. This is particularly effective and I encourage students to spend some time to understand the solutions presented instead of just copying working code. By digging into why a particular code chunk works, you'll learn all sorts of neat tricks and you'll find yourself utilizing the site less frequently.

1.6 Exercises

Create an RMarkdown file that solves the following exercises.

1. Calculate $\log(6.2)$ first using base e and second using base 10. To figure out how to do different bases, it might be helpful to look at the help page for the `log` function.

2. Calculate the square root of 2 and save the result as the variable named sqrt2. Have R display the decimal value of sqrt2. *Hint: use Google to find the square root function. Perhaps search on the keywords "R square root function".*

3. This exercise walks you through installing a package with all the datasets used in the textbook *The Statistical Sleuth*.

   a) Install the package `Sleuth3` on your computer using RStudio.
   b) Load the package using the `library()` command.
   c) Print out the dataset `case0101`

Chapter 2

Data Frames

```r
# Load my favorite packages: dplyr, ggplot2, forcats, readr, and stringr
library(tidyverse)
```

```
## -- Attaching packages --------
```

```
## v ggplot2 3.1.1       v purrr   0.3.2
## v tibble  2.1.1       v dplyr   0.8.0.1
## v tidyr   0.8.3       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0
```

```
## -- Conflicts -----------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Data frames are the fundamental unit of data storage that casual users of R need to work with. Conceptually they are just like a single tab in a spreadsheet (e.g. Excel) file. There are multiple rows and columns and each column is of the same type of information (e.g. numerical values, dates, or character strings) and each row represents a single observation.

Because the columns have meaning and we generally give them column names, it is desirable to want to access an element by the name of the column as opposed to the column number. While writing formulas in large Excel spreadsheets I often get annoyed trying to remember which column something was in and muttering "Was total biomass in column P or Q?" A system where I could just name the column `Total_Biomass` and then always refer to it that way, is much nicer to work with and I make fewer dumb mistakes.

In this chapter we will briefly cover the minimal set of tools for working with data frames. First we discuss how to import data sets, both packages from packages and from appropriately formated Excel and .csv files. Next we'll create new columns based on some calculation, and finally we'll look at how to summarize information across rows.

2.1 Introduction to Importing Data

2.1.1 From a Package

For many students, they will be assigned homework that utilizes data sets that are stored in some package. To access those, we would need to first install the package if we haven't already. Recall to do that, we can use the Rstudio menu bar "Tools -> Install Packages…" mouse action.

Because we might have thousands of packages installed on a computer, and those packages might all have data sets associated with them, they aren't loaded into memory by default. Instead we have to go through a two-step process of making sure that the package is installed on the computer, and then load the desired data set into the running session of R. Once the package is intalled, we can load the data into our session via the following command:

```
data('alfalfa', package='faraway')    # load the data set 'alfalfa' from the package 'faraway'
```

Because R tries to avoid loading datasets until it is sure that you need them, the object `alfalfa` isn't initially loaded as a `data.frame` but rather as a "promise" that it eventually will be loaded whenever you first use it. So lets first access it by viewing it.

```
View(alfalfa)
```

There are two ways to enter the view command. Either executing the `View()` function from the console, or clicking on either the white table or the object name in the `Environment` tab.

```
# Show the image of the environment tab with the white table highlighted
```

2.1.2 Import from `.csv` or `.xls` files

Often times data is stored in a "Comma Separated Values" file (with the file suffix of .csv) where the rows in the file represent the data frame rows, and the columns are just separated by commas. The first row of the file is usually the column titles.

Alternatively, the data might be stored in an Excel file and we just need to tell R where the file is and which worksheet tab to import.

The hardest part for people that are new to programming is giving the path to the data file. In this case, I recommend students use the data import wizard that RStudio includes which is accessed via 'File -> Import Dataset'. This will then give you a choice of file types to read from (.csv files are in the "Text" options). Once you have selected the file type to import, the user is presented with a file browser window where the desired file should be located. Once the file is chosen, we can import of the file.

Critically, we should notice that the import wizard generates R code that does the actual import. We MUST copy that code into our Rmarkdown file or else the import won't happen when we try to knit the Rmarkdown into an output document because knitting always occurs in a completely fresh R session. So only use the import wizard to generate the import code! The code generated by the import wizard ends with a `View()` command and I typically remove that as it can interfer with the knitting process.

2.2 Data Frame Manipulation

Many of the tools to manipulate data frames in R were written without a consistent syntax and are difficult use together. To remedy this, Hadley Wickham (the writer of `ggplot2`) introduced a package called plyr which was quite useful. As with many projects, his first version was good but not great and he introduced an improved version that works exclusively with data.frames called `dplyr` which we will investigate. The package `dplyr` strives to provide a convenient and consistent set of functions to handle the most common data frame manipulations and a mechanism for chaining these operations together to perform complex tasks.

The Dr Wickham has put together a very nice introduction to the package that explains in more detail how the various pieces work and I encourage you to read it at some point. [http://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html].

One of the aspects about the `data.frame` object is that R does some simplification for you, but it does not do it in a consistent manner. Somewhat obnoxiously character strings are always converted to factors and subsetting might return a `data.frame` or a `vector` or a `scalar`. This is fine at the command line, but can be problematic when programming. Furthermore, many operations are pretty slow using `data.frame`. To get around this, Dr Wickham introduced a modified version of the `data.frame` called a `tibble`. A `tibble` is a `data.frame` but with a few extra bits. For now we can ignore the differences.

The pipe command `%>%` allows for very readable code. The idea is that the `%>%` operator works by translating the command `a %>% f(b)` to the expression `f(a,b)`. This operator works on any function and was introduced in the `magrittr` package. The beauty of this comes when you have a suite of functions that takes input arguments of the same type as their output.

For example, if we wanted to start with `x`, and first apply function `f()`, then `g()`, and then `h()`, the usual R command would be `h(g(f(x)))` which is hard to read because you have to start reading at the *innermost* set of parentheses. Using the pipe command `%>%`, this sequence of operations becomes `x %>% f() %>% g() %>% h()`.

Written	Meaning
`a %>% f(b)`	`f(a,b)`
`b %>% f(a, .)`	`f(a, b)`
`x %>% f() %>% g()`	`g( f(x) )`

In `dplyr`, all the functions below take a *data set as its first argument* and *outputs an appropriately modified data set*. This will allow me to chain together commands in a readable fashion. The pipe command works with any function, not just the `dplyr` functions and I often find myself using it all over the place.

2.2.1 Verbs

The foundational operations to perform on a data set are:

- Subsetting - Returns a with only particular columns or rows
  - `select` - Selecting a subset of columns by name or column number.
  - `filter` - Selecting a subset of rows from a data frame based on logical expressions.
  - `slice` - Selecting a subset of rows by row number.
- `arrange` - Re-ordering the rows of a data frame.
- `mutate` - Add a new column that is some function of other columns.
- `summarise` - calculate some summary statistic of a column of data. This collapses a set of rows into a single row.

Each of these operations is a function in the package `dplyr`. These functions all have a similar calling syntax, the first argument is a data set, subsequent arguments describe what to do with the input data frame and you can refer to the columns without using the `df$column` notation. All of these functions will return a data set.

2.2.1.1 Subsetting with `select`, `filter`, and `slice`

These function allows you select certain columns and rows of a data frame.

2.2.1.1.1 `select()`

Often you only want to work with a small number of columns of a data frame and want to be able to *select* a subset of columns or perhaps remove a subset. The function to do that is `dplyr::select()`

```
# Create a tiny data frame that is easy to see what is happening
grades <- data.frame(
  l.name = c('Cox', 'Dorian', 'Kelso', 'Turk'),
  Exam1 = c(93, 89, 80, 70),
  Exam2 = c(98, 70, 82, 85),
  Final = c(96, 85, 81, 92) )

grades
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
## 2 Dorian    89    70    85
## 3  Kelso    80    82    81
## 4   Turk    70    85    92
```

I could select the columns Exam columns by hand, or by using an extension of the `:` operator

```
# select( grades,  Exam1, Exam2 )   # from `grades`, select columns Exam1, Exam2
grades %>% select( Exam1, Exam2 )   # Exam1 and Exam2
```

```
##   Exam1 Exam2
## 1    93    98
## 2    89    70
## 3    80    82
## 4    70    85
```

```
grades %>% select( Exam1:Final )    # Columns Exam1 through Final
```

```
##   Exam1 Exam2 Final
## 1    93    98    96
## 2    89    70    85
## 3    80    82    81
## 4    70    85    92
```

```
grades %>% select( -Exam1 )         # Negative indexing by name drops a column
```

```
##   l.name Exam2 Final
## 1    Cox    98    96
## 2 Dorian    70    85
## 3  Kelso    82    81
## 4   Turk    85    92
```

```
grades %>% select( 1:2 )            # Can select column by column position
```

```
##   l.name Exam1
## 1    Cox    93
## 2 Dorian    89
## 3  Kelso    80
## 4   Turk    70
```

The `select()` command has a few other tricks. There are functional calls that describe the columns you wish to select that take advantage of pattern matching. I generally can get by with `starts_with()`, `ends_with()`, and `contains()`, but there is a final operator `matches()` that takes a regular expression.

```
grades %>% select( starts_with('Exam') )   # Exam1 and Exam2
```

```
##   Exam1 Exam2
## 1    93    98
## 2    89    70
## 3    80    82
## 4    70    85
```

The `dplyr::select` function is quite handy, but there are several other packages out there that have a `select` function and we can get into trouble with loading other packages with the same function names. If I encounter the `select` function behaving in a weird manner or complaining about an input argument, my first remedy is to be explicit about it is the `dplyr::select()` function by appending the package name at the start.

2.2.1.1.2 `filter()`

It is common to want to select particular rows where we have some logical expression to pick the rows.

```r
# select students with Final grades greater than 90
grades %>% filter(Final > 90)
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
## 2   Turk    70    85    92
```

You can have multiple logical expressions to select rows and they will be logically combined so that only rows that satisfy all of the conditions are selected. The logicals are joined together using `&` (and) operator or the `|` (or) operator and you may explicitly use other logicals. For example a factor column type might be used to select rows where type is either one or two via the following: `type==1 | type==2`.

```r
# select students with Final grades above 90 and
# average score also above 90
grades %>% filter(Exam2 > 90, Final > 90)
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
```

```r
# we could also use an "and" condition
grades %>% filter(Exam2 > 90 & Final > 90)
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
```

2.2.1.1.3 `slice()`

When you want to filter rows based on row number, this is called slicing.

```r
# grab the first 2 rows
grades %>% slice(1:2)
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
## 2 Dorian    89    70    85
```

2.2.1.2 `arrange()`

We often need to re-order the rows of a data frame. For example, we might wish to take our grade book and sort the rows by the average score, or perhaps alphabetically. The `arrange()` function does exactly that. The first argument is the data frame to re-order, and the subsequent arguments are the columns to sort on. The order of the sorting column determines the precedent… the first sorting column is first used and the second sorting column is only used to break ties.

```
grades %>% arrange(l.name)
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
## 2 Dorian    89    70    85
## 3  Kelso    80    82    81
## 4   Turk    70    85    92
```

The default sorting is in ascending order, so to sort the grades with the highest scoring person in the first row, we must tell arrange to do it in descending order using `desc(column.name)`.

```
grades %>% arrange(desc(Final))
```

```
##   l.name Exam1 Exam2 Final
## 1    Cox    93    98    96
## 2   Turk    70    85    92
## 3 Dorian    89    70    85
## 4  Kelso    80    82    81
```

In a more complicated example, consider the following data and we want to order it first by Treatment Level and secondarily by the y-value. I want the Treatment level in the default ascending order (Low, Medium, High), but the y variable in descending order.

```
# make some data
dd <- data.frame(
  Trt = factor(c("High", "Med", "High", "Low"),
               levels = c("Low", "Med", "High")),
  y = c(8, 3, 9, 9),
  z = c(1, 1, 1, 2))
dd
```

```
##    Trt y z
## 1 High 8 1
## 2  Med 3 1
## 3 High 9 1
## 4  Low 9 2
```

```
# arrange the rows first by treatment, and then by y (y in descending order)
dd %>% arrange(Trt, desc(y))
```

```
##    Trt y z
## 1  Low 9 2
## 2  Med 3 1
## 3 High 9 1
## 4 High 8 1
```

2.2.1.3 mutate()

I often need to create a new column that is some function of the old columns. In the `dplyr` package, this is a `mutate` command. To do ths, we give a `mutate( NewColumn = Function of Old Columns )` command.

```
# Modify the grades data frame and replace the old version with the new
# that contains the newly created "average" column
grades <- grades %>%
  mutate( average = (Exam1 + Exam2 + Final)/3 )
```

You can do multiple calculations within the same `mutate()` command, and you can even refer to columns that were created in the same `mutate()` command.

```
grades %>% mutate(
  average = (Exam1 + Exam2 + Final)/3,
  grade = cut(average, c(0, 60, 70, 80, 90, 100),  # cut takes numeric variable
                       c( 'F','D','C','B','A')) )  # and makes a factor
```

```
##   l.name Exam1 Exam2 Final  average grade
## 1    Cox    93    98    96 95.66667     A
## 2 Dorian    89    70    85 81.33333     B
## 3  Kelso    80    82    81 81.00000     B
## 4   Turk    70    85    92 82.33333     B
```

2.2.1.4 summarise()

By itself, this function is quite boring, but will become useful later on. Its purpose is to calculate summary statistics using any or all of the data columns. Notice that we get to chose the name of the new column. The way to think about this is that we are collapsing information stored in multiple rows into a single row of values.

```
# calculate the mean of exam 1
grades %>% summarise( mean.E1=mean(Exam1) )
```

```
##   mean.E1
## 1      83
```

We could calculate multiple summary statistics if we like.

```
# calculate the mean and standard deviation
grades %>% summarise( mean.E1=mean(Exam1), stddev.E1=sd(Exam1) )
```

```
##   mean.E1 stddev.E1
## 1      83  10.23067
```

If we want to apply the same statistic to each column, we use the `summarise_all()` command. We have to be a little careful here because the function you use has to work on every column (that isn't part of the grouping structure (see `group_by()`)). There are two variants `summarize_at()` and `summarize_if()` that give you a bit more flexibility.

```
# calculate the mean and standard devation of each of the score
# columns. Notice I have a funny way of specifying the functions
# mean() and sd() that I want to use.
grades %>%
  select( Exam1:Final ) %>%
  summarise_all( list( ~mean , ~sd) )
```

```
##   Exam1_mean Exam2_mean Final_mean Exam1_sd Exam2_sd Final_sd
## 1         83      83.75       88.5 10.23067     11.5 6.757712
```

```
grades  %>%
  summarise_if(is.numeric, list( ~mean , ~sd) )
```

```
##   Exam1_mean Exam2_mean Final_mean average_mean Exam1_sd Exam2_sd Final_sd
## 1         83      83.75       88.5     85.08333 10.23067     11.5 6.757712
##   average_sd
## 1   7.078266
```

2.2.1.5 Miscellaneous functions

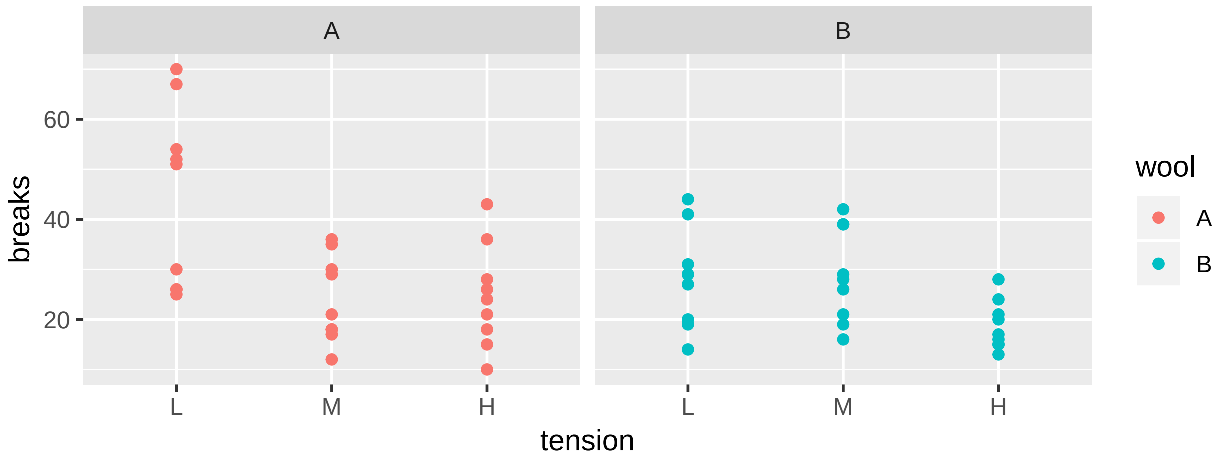
There are some more function that are useful but aren't as commonly used. For sampling the functions `sample_n()` and `sample_frac()` will take a sub-sample of either n rows or of a fraction of the data set. The function `n()` returns the number of rows in the data set. Finally `rename()` will rename a selected column.

2.2.2 Split, apply, combine

Aside from unifying the syntax behind the common operations, the major strength of the `dplyr` package is the ability to split a data frame into a bunch of sub-data frames, apply a sequence of one or more of the operations we just described, and then combine results back together. We'll consider data from an experiment from spinning wool into yarn. This experiment considered two different types of wool (A or B) and three different levels of tension on the thread. The response variable is the number of breaks in the resulting yarn. For each of the 6 `wool:tension` combinations, there are 9 replicated observations per `wool:tension` level.

```
data(warpbreaks)
str(warpbreaks)
```

```
## 'data.frame':    54 obs. of  3 variables:
##  $ breaks : num  26 30 54 25 70 52 51 26 67 18 ...
##  $ wool   : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
##  $ tension: Factor w/ 3 levels "L","M","H": 1 1 1 1 1 1 1 1 1 2 ...
```



The first we must do is to create a data frame with additional information about how to break the data into sub-data frames. In this case, I want to break the data up into the 6 wool-by-tension combinations. Initially we will just figure out how many rows are in each wool-by-tension combination.

```
# group_by:  what variable(s) shall we group on.
# n() is a function that returns how many rows are in the
#   currently selected sub-dataframe
warpbreaks %>%
  group_by( wool, tension) %>%     # grouping
  summarise(n = n() )              # how many in each group
```

```
## # A tibble: 6 x 3
## # Groups:   wool [2]
##   wool  tension     n
##   <fct> <fct>   <int>
## 1 A     L           9
## 2 A     M           9
## 3 A     H           9
## 4 B     L           9
## 5 B     M           9
## 6 B     H           9
```

The `group_by` function takes a data.frame and returns the same data.frame, but with some extra information so that any subsequent function acts on each unique combination defined in the `group_by`. If you wish to remove this behavior, use `group_by()` to reset the grouping to have no grouping variable.

Using the same `summarise` function, we could calculate the group mean and standard deviation for each wool-by-tension group.

```
warpbreaks %>%
  group_by(wool, tension) %>%
  summarise( n           = n(),            # I added some formatting to show the
             mean.breaks = mean(breaks),   # reader I am calculating several
             sd.breaks   = sd(breaks))     # statistics.
```

```
## # A tibble: 6 x 5
## # Groups:   wool [2]
##   wool  tension     n mean.breaks sd.breaks
##   <fct> <fct>   <int>       <dbl>     <dbl>
## 1 A     L           9        44.6     18.1
## 2 A     M           9        24        8.66
## 3 A     H           9        24.6     10.3
## 4 B     L           9        28.2      9.86
## 5 B     M           9        28.8      9.43
## 6 B     H           9        18.8      4.89
```

If instead of summarizing each split, we might want to just do some calculation and the output should have the same number of rows as the input data frame. In this case I'll tell `dplyr` that we are mutating the data frame instead of summarizing it. For example, suppose that I want to calculate the residual value

$$e_{ijk} = y_{ijk} - \bar{y}_{ij\cdot}$$

where $\bar{y}_{ij\cdot}$ is the mean of each `wool:tension` combination.

```
warpbreaks %>%
   group_by(wool, tension) %>%                 # group by wool:tension
   mutate(resid = breaks - mean(breaks)) %>%   # mean(breaks) of the group!
   head(  )                                    # show the first couple of rows
```

```
## # A tibble: 6 x 4
## # Groups:   wool, tension [1]
##   breaks wool  tension  resid
##    <dbl> <fct> <fct>    <dbl>
## 1     26 A     L       -18.6
## 2     30 A     L       -14.6
## 3     54 A     L         9.44
## 4     25 A     L       -19.6
## 5     70 A     L        25.4
## 6     52 A     L         7.44
```

2.2.3 Chaining commands together

In the previous examples we have used the %>% operator to make the code more readable but to really appreciate this, we should examine the alternative.

Suppose we have the results of a small 5K race. The data given to us is in the order that the runners signed up but we want to calculate the results for each gender, calculate the placings, and the sort the data frame by gender and then place. We can think of this process as having three steps:

1. Splitting
2. Ranking
3. Re-arranging.

```r
# input the initial data
race.results <- data.frame(
  name=c('Bob', 'Jeff', 'Rachel', 'Bonnie', 'Derek', 'April','Elise','David'),
  time=c(21.23, 19.51, 19.52, 23.45, 20.23, 24.22, 28.33, 15.48),
  gender=c('M','M','F','F','M','F','F','M'))
```

We could run all the commands together using the following code:

```r
arrange(
  mutate(
    group_by(
      race.results,          # using race.results
      gender),               # group by gender
    place = rank( time )),   # mutate to calculate the place column
  gender, place)             # arrange the result by gender and place
```

```
## # A tibble: 8 x 4
## # Groups:   gender [2]
##   name    time gender place
##   <fct>  <dbl> <fct>  <dbl>
## 1 Rachel  19.5 F          1
## 2 Bonnie  23.4 F          2
## 3 April   24.2 F          3
## 4 Elise   28.3 F          4
## 5 David   15.5 M          1
## 6 Jeff    19.5 M          2
## 7 Derek   20.2 M          3
## 8 Bob     21.2 M          4
```

This is very difficult to read because you have to read the code *from the inside out.*

Another (and slightly more readable) way to complete our task is to save each intermediate step of our process and then use that in the next step:

```r
temp.df0 <- race.results %>% group_by( gender)
temp.df1 <- temp.df0 %>% mutate( place = rank(time) )
temp.df2 <- temp.df1 %>% arrange( gender, place )
```

It would be nice if I didn't have to save all these intermediate results because keeping track of temp1 and temp2 gets pretty annoying if I keep changing the order of how things or calculated or add/subtract steps. This is exactly what %>% does for me.

```r
race.results %>%
  group_by( gender ) %>%
  mutate( place = rank(time)) %>%
  arrange( gender, place )
```

```
## # A tibble: 8 x 4
## # Groups:   gender [2]
##   name    time gender place
##   <fct>  <dbl> <fct>  <dbl>
## 1 Rachel  19.5 F          1
## 2 Bonnie  23.4 F          2
## 3 April   24.2 F          3
## 4 Elise   28.3 F          4
## 5 David   15.5 M          1
## 6 Jeff    19.5 M          2
## 7 Derek   20.2 M          3
## 8 Bob     21.2 M          4
```

2.3 Exercises

1. The dataset `ChickWeight` tracks the weights of 48 baby chickens (chicks) feed four different diets.
   a. Load the dataset using

      ```
      data(ChickWeight)
      ```

   b. Look at the help files for the description of the columns.
   c) Remove all the observations except for observations from day 10 or day 20.
   d) Calculate the mean and standard deviation of the chick weights for each diet group on days 10 and 20.
2. The OpenIntro textbook on statistics includes a data set on body dimensions.
   a) Load the file using

      ```
      Body <- read.csv('http://www.openintro.org/stat/data/bdims.csv')
      ```

   b) The column sex is coded as a 1 if the individual is male and 0 if female. This is a non-intuitive labeling system. Create a new column `sex.MF` that uses labels Male and Female. *Hint: the ifelse() command will be very convenient here.*
   c) The columns `wgt` and `hgt` measure weight and height in kilograms and centimeters (respectively). Use these to calculate the Body Mass Index (BMI) for each individual where

$$BMI = \frac{Weight\,(kg)}{[Height\,(m)]^2}$$

   d) Double check that your calculated BMI column is correct by examining the summary statistics of the column (e.g. `summary(Body)`). BMI values should be between 18 to 40 or so. Did you make an error in your calculation?

   e) The function `cut` takes a vector of continuous numerical data and creates a factor based on your give cut-points.

      ```
      # Define a continuous vector to convert to a factor
      x <- 1:10

      # divide range of x into three groups of equal length
      cut(x, breaks=3)
      ```

      ```
      ##  [1] (0.991,4] (0.991,4] (0.991,4] (0.991,4] (4,7]     (4,7]     (4,7]
      ##  [8] (7,10]    (7,10]    (7,10]
      ## Levels: (0.991,4] (4,7] (7,10]
      ```

```
# divide x into four groups, where I specify all 5 break points
cut(x, breaks = c(0, 2.5, 5.0, 7.5, 10))
```

```
##  [1] (0,2.5]  (0,2.5]  (2.5,5]  (2.5,5]  (2.5,5]  (5,7.5]  (5,7.5]
##  [8] (7.5,10] (7.5,10] (7.5,10]
## Levels: (0,2.5] (2.5,5] (5,7.5] (7.5,10]
```

```
# (0,2.5] (2.5,5] means 2.5 is included in first group
# right=FALSE changes this to make 2.5 included in the second

# divide x into 3 groups, but give them a nicer
# set of group names
cut(x, breaks=3, labels=c('Low','Medium','High'))
```

```
##  [1] Low    Low    Low    Low    Medium Medium Medium High   High   High
## Levels: Low Medium High
```

Create a new column of in the data frame that divides the age into decades (10-19, 20-29, 30-39, etc). Notice the oldest person in the study is 67.

```
Body <- Body %>%
  mutate( Age.Grp = cut(age,
                        breaks=c(10,20,30,40,50,60,70),
                        right=FALSE))
```

f) Find the average BMI for each Sex-by-Age combination.

Chapter 3

Graphing using ggplot2

```r
library(ggplot2)  # my favorite graphing system
library(dplyr)    # data frame manipulations
```

There are three major "systems" of making graphs in R. The basic plotting commands in R are quite effective but the commands do not have a way of being combined in easy ways. Lattice graphics (which the `mosaic` package uses) makes it possible to create some quite complicated graphs but it is very difficult to do make non-standard graphs. The last package, `ggplot2` tries to not anticipate what the user wants to do, but rather provide the mechanisms for pulling together different graphical concepts and the user gets to decide which elements to combine.

To make the most of `ggplot2` it is important to wrap your mind around "The Grammar of Graphics". Briefly, the act of building a graph can be broken down into three steps.

1. Define what data we are using.
2. What is the major relationship we wish to examine?
3. In what way should we present that relationship? These relationships can be presented in multiple ways, and the process of creating a good graph relies on building layers upon layers of information. For example, we might start with printing the raw data and then overlay a regression line over the top.

Next, it should be noted that `ggplot2` is designed to act on data frames. It is actually hard to just draw three data points and for simple graphs it might be easier to use the base graphing system in R. However for any real data analysis project, the data will already be in a data frame and this is not an annoyance.

These notes are sufficient for creating simple graphs using `ggplot2`, but are not intended to be exhaustive. There are many places online to get help with `ggplot2`. One very nice resource is the website, http://www.cookbook-r.com/Graphs/, which gives much of the information available in the book R Graphics Cookbook which I highly recommend. Second is just googling your problems and see what you can find on websites such as StackExchange.

One way that `ggplot2` makes it easy to form very complicated graphs is that it provides a large number of basic building blocks that, when stacked upon each other, can produce extremely complicated graphs. A full list is available at http://docs.ggplot2.org/current/ but the following list gives some idea of different building blocks. These different geometries are different ways to display the relationship between variables and can be combined in many interesting ways.

Geom	Description	Required Aesthetics
`geom_histogram`	A histogram	`x`
`geom_bar`	A barplot	`x`

Geom	Description	Required Aesthetics
`geom_density`	A density plot of data. (smoothed histogram)	`x`
`geom_boxplot`	Boxplots	`x, y`
`geom_line`	Draw a line (after sorting x-values)	`x, y`
`geom_path`	Draw a line (without sorting x-values)	`x, y`
`geom_point`	Draw points (for a scatterplot)	`x, y`
`geom_smooth`	Add a ribbon that summarizes a scatterplot	`x, y`
`geom_ribbon`	Enclose a region, and color the interior	`ymin, ymax`
`geom_errorbar`	Error bars	`ymin, ymax`
`geom_text`	Add text to a graph	`x, y, label`
`geom_label`	Add text to a graph	`x, y, label`
`geom_tile`	Create Heat map	`x, y, fill`

A graph can be built up layer by layer, where:

- Each layer corresponds to a `geom`, each of which requires a dataset and a mapping between an aesthetic and a column of the data set.
  - If you don't specify either, then the layer inherits everything defined in the `ggplot()` command.
  - You can have different datasets for each layer!
- Layers can be added with a `+`, or you can define two plots and add them together (second one over-writes anything that conflicts).

3.1 Basic Graphs

3.1.1 Bar Charts

Bar charts and histograms are how we think about displaying informtion about a single covariate. That is to say, we are not trying to make a graph of the relationship between x and y, but rather understanding what values of x are present and how frequently they show up.

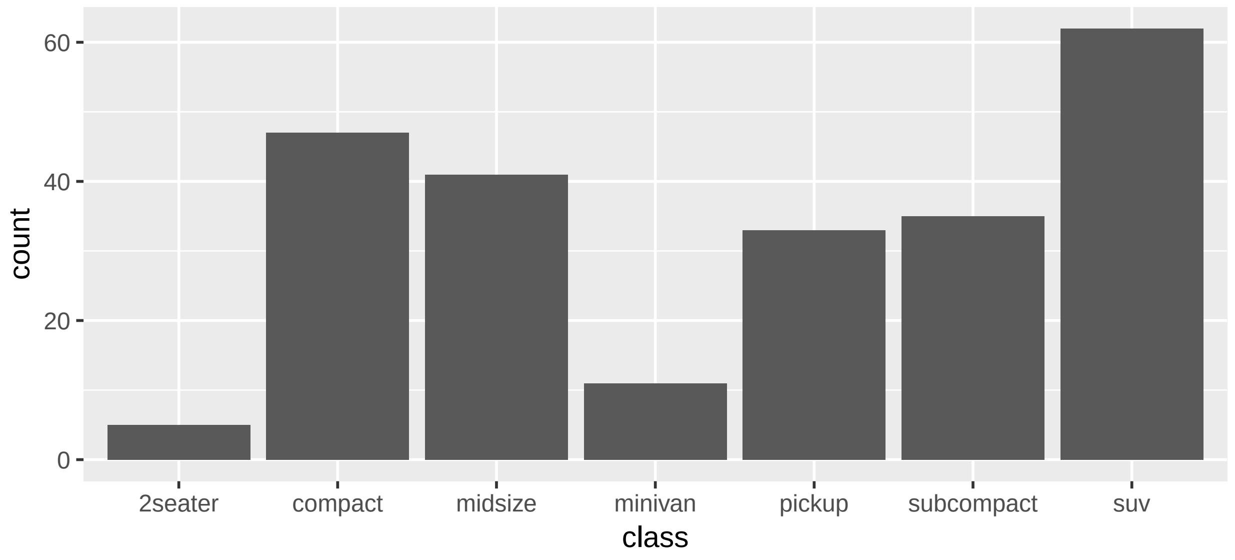
For displaying a categorical variable on the x-axis, a bar chart is a good option. Here we consider a data set that gives the fuel efficiency of different classes of vehicles in two different years. This is a subset of data that the EPA makes available on http://fueleconomy.gov. It contains only model which had a new release every year between 1999 and 2008 and therefore represents the most popular cars sold in the US. It includes information for each model for years 1999 and 2008. The dataset is included in the `ggplot2` package as `mpg`.

```r
data(mpg, package='ggplot2')  # load the dataset
str(mpg)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    234 obs. of  11 variables:
##  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
##  $ model       : chr  "a4" "a4" "a4" "a4" ...
##  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr  "p" "p" "p" "p" ...
##  $ class       : chr  "compact" "compact" "compact" "compact" ...
```

First we could summarize the data by how many models there are in the different classes.

```r
ggplot(data=mpg, aes(x=class)) +
  geom_bar()
```



1. The data set we wish to use is specified using `data=mpg`. This is the first argument defined in the function, so you could skip the `data=` part if the input data.frame is the first argument.

2. The column in the data that we wish to investigate is defined in the `aes(x=class)` part. This means the x-axis will be the car's class, which is indicated by the column named `class`.

3. The way we want to display this information is using a bar chart.

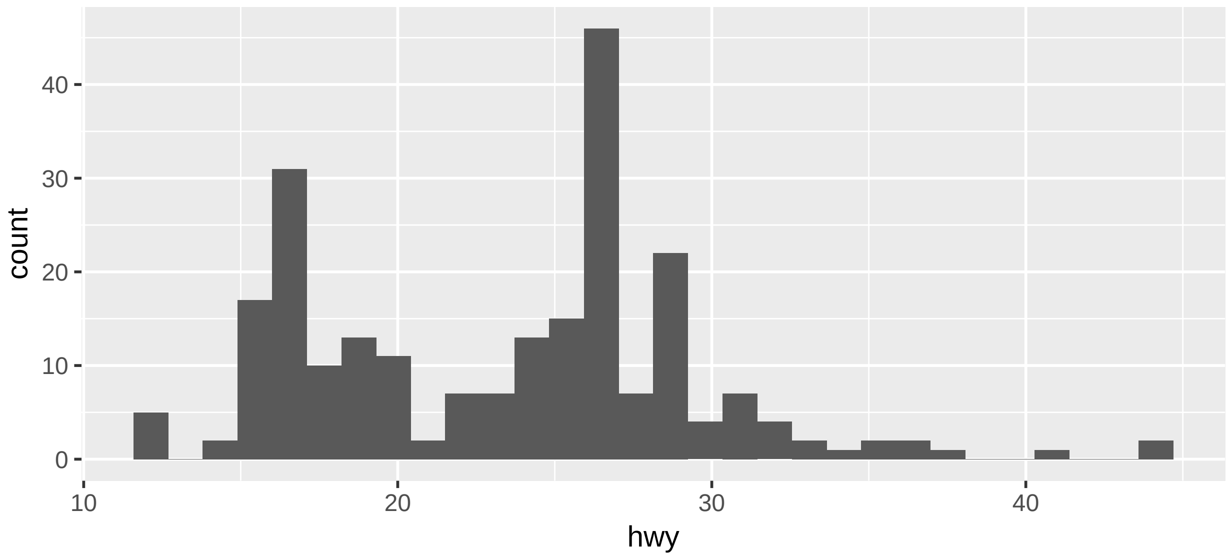
By default, the `geom_bar()` just counts the number of cases and displays how many observations were in each factor level. If I have a data frame that I have already summarized, `geom_col` will allow you to set the height of the bar by a y column.

3.1.2 Histograms

Histograms perform a similar task as a bar graph, but with continuous numerical data. It focuses on a single variable and gives how frequently particular ranges of the data occur.

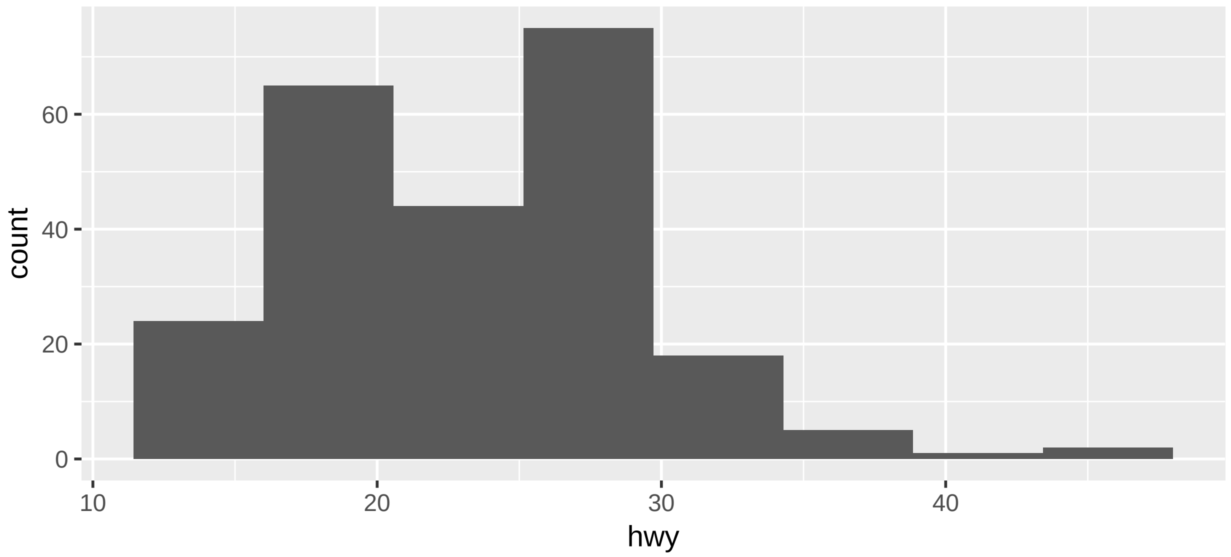
```r
ggplot(mpg, aes(x=hwy)) +
  geom_histogram()
```

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



Just as `geom_bar` by default calculated the number of observations in each level of my factor of interest, `geom_histogram` breaks up the x-axis into distinct bins (by default, 30 bins), and then counts how many observations fall into each bin, and displys the number as a bar. To change the number of bins, we could either tell it the number of bins (e.g. `bins=20`) or the width of each bin (e.g. `binwidth=4`).

```
ggplot(mpg, aes(x=hwy)) +
  geom_histogram(bins=8)       # 8 bins
```

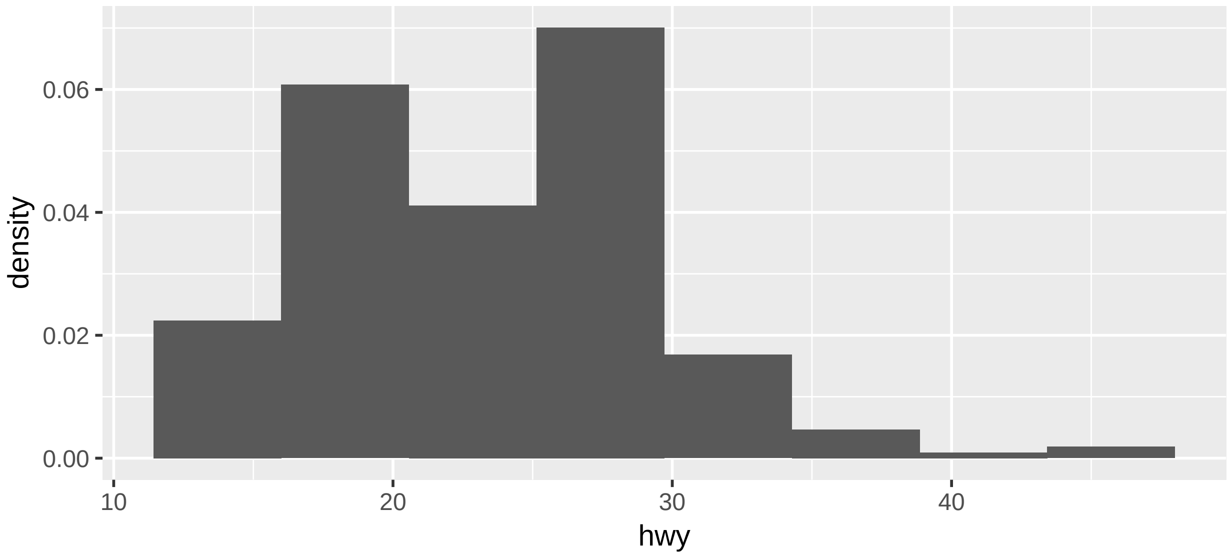


Often we want to rescale the y-axis so that it is in terms of density, which is

$$density = \frac{\#\ observations\ in\ bin}{total\ number\ observations} \cdot \frac{1}{bin\ width}$$

To ask `geom_histogram` to calculate the density instead of counts, we simply add an option to the `aes()` list that specifies that the y-axis should be the density. Notice that this only rescales the y-axis and the shape of the histogram is identical.

```
ggplot(mpg, aes(x=hwy, y=..density..)) +
  geom_histogram(bins=8)       # 8 bins
```



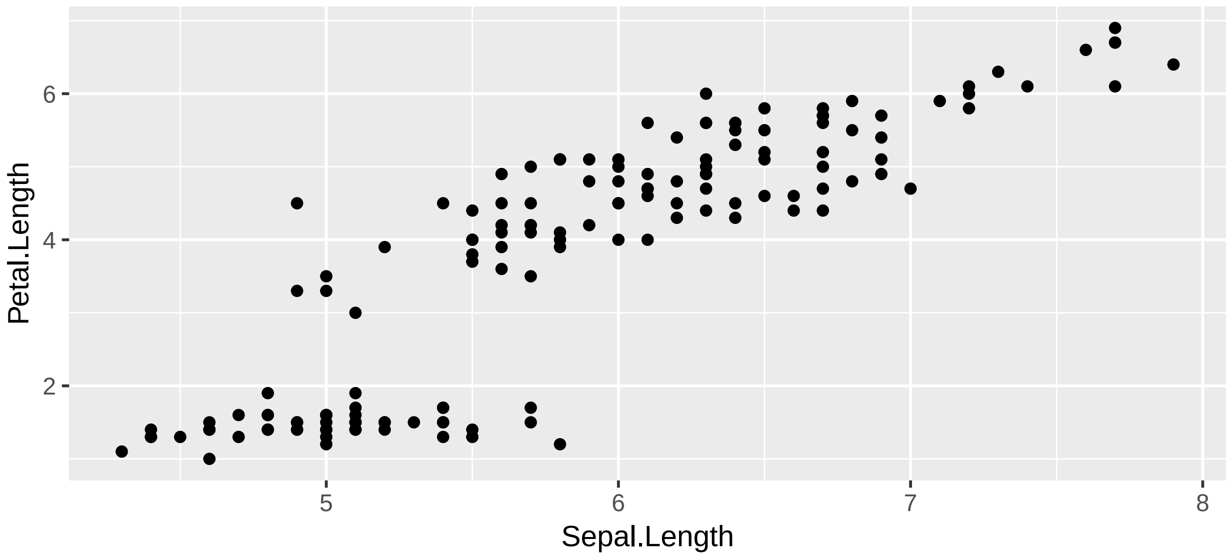
3.1.3 Scatterplots

To start with, we'll make a very simple scatterplot using the `iris` dataset. Recall that the `iris` dataset contains observations on 150 iris plants where we've measured the length and width of the petals and sepals. We will make a scatterplot of `Sepal.Length` versus `Petal.Length`, which are two columns in the dataset.

```
data(iris)  # load the iris dataset that comes with R
str(iris)   # what columns do we have to play with...
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

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



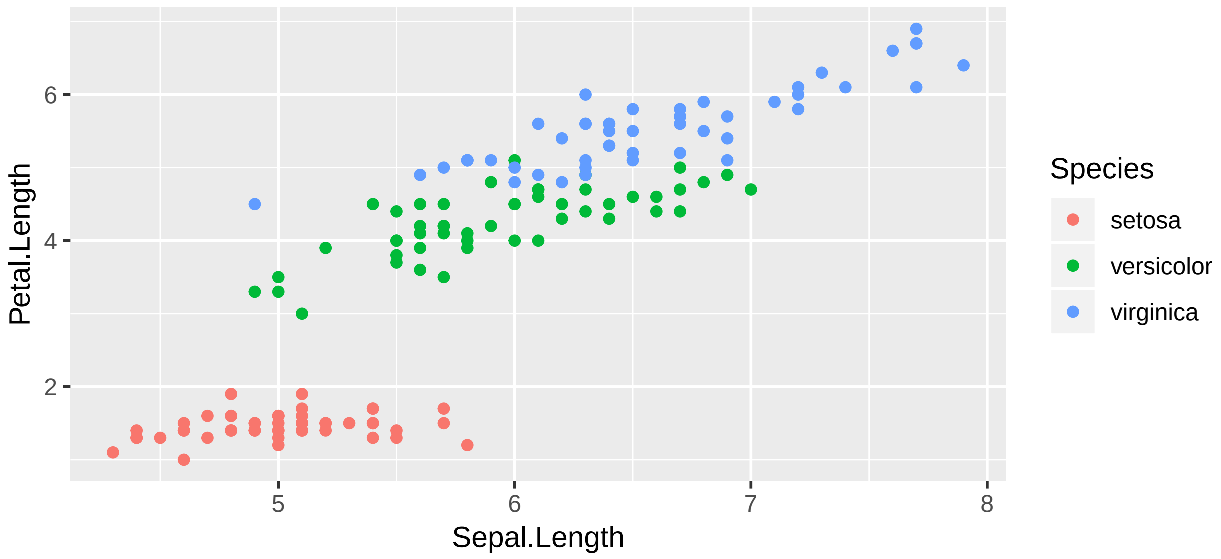
1. The data set we wish to use is specified using `data=iris`.

2. The relationship we want to explore is `x=Sepal.Length` and `y=Petal.Length`. This means the x-axis will be the Sepal Length and the y-axis will be the Petal Length.

3. The way we want to display this relationship is through graphing 1 point for every observation.

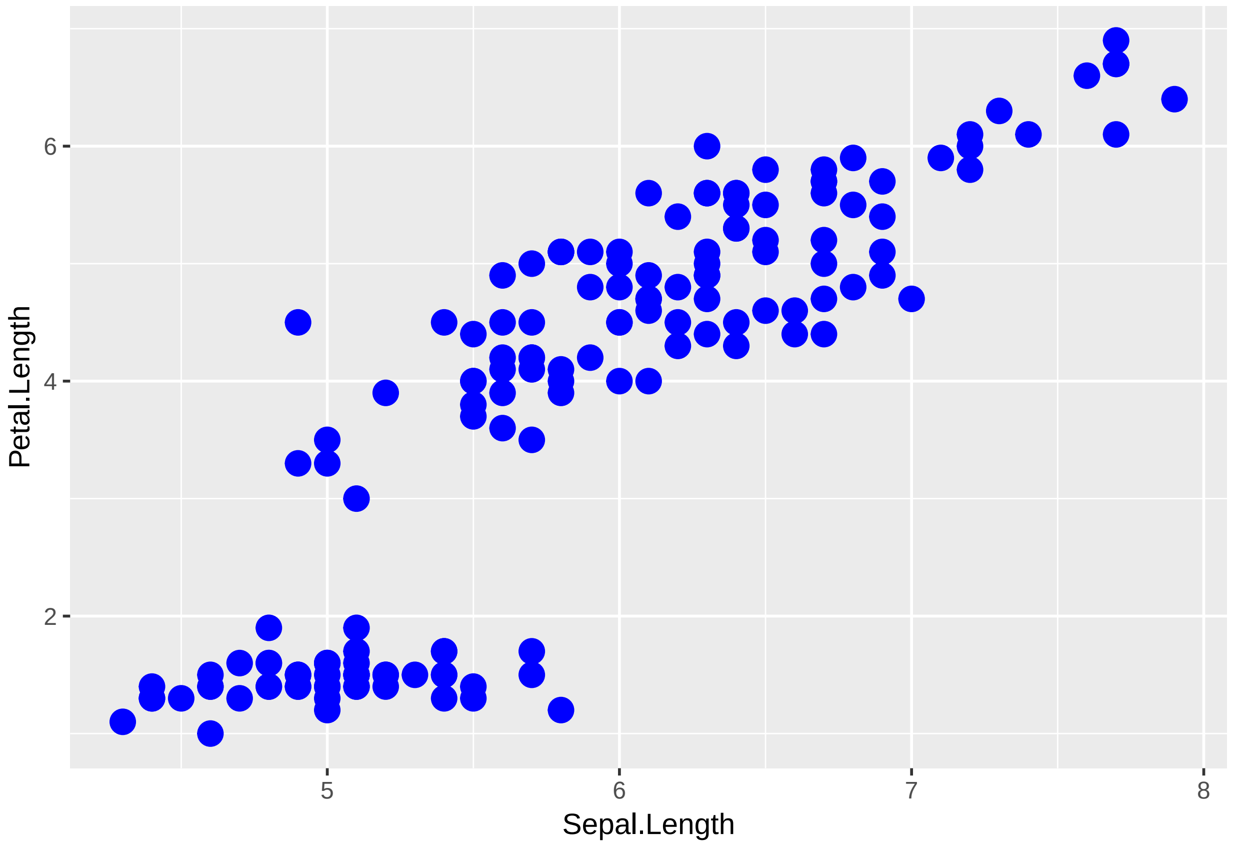
We can define other attributes that might reflect other aspects of the data. For example, we might want for the color of the data point to change dynamically based on the species of iris.

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



The `aes()` command inside the previous section of code is quite mysterious. The way to think about the `aes()` is that it gives you a way to define relationships that are data dependent. In the previous graph, the x-value and y-value for each point was defined dynamically by the data, as was the color. If we just wanted all the data points to be colored blue and larger, then the following code would do that

```r
ggplot( data=iris, aes(x=Sepal.Length, y=Petal.Length) ) +
    geom_point( color='blue', size=4 )
```



The important part isn't that color and size were defined in the `geom_point()` but that they were defined outside of an `aes()` function!

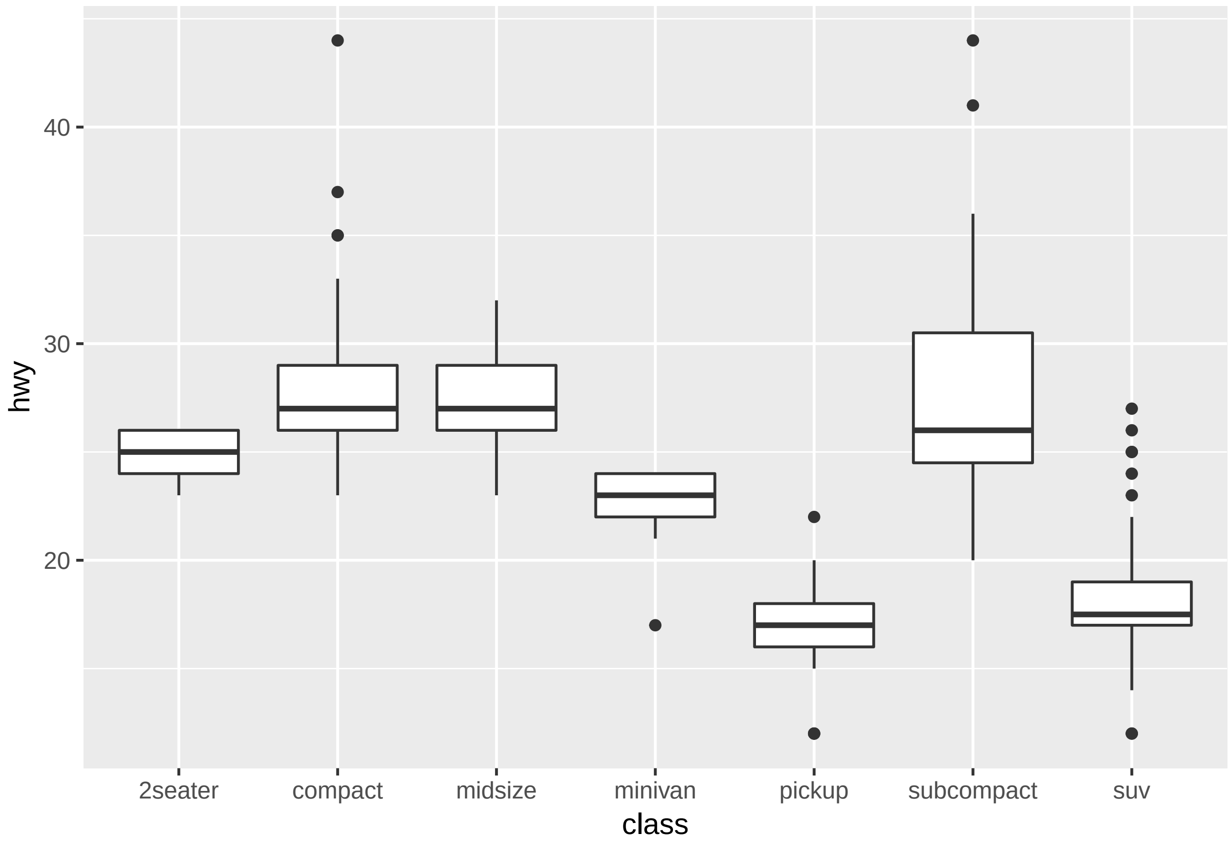
1. Anything set inside an `aes()` command will be of the form `attribute=Column_Name` and will change based on the data.

2. Anything set outside an `aes()` command will be in the form `attribute=value` and will be fixed.

3.1.4 Box Plots

Boxplots are a common way to show a categorical variable on the x-axis and continuous on the y-axis.

```
ggplot(mpg, aes(x=class, y=hwy)) +
  geom_boxplot()
```



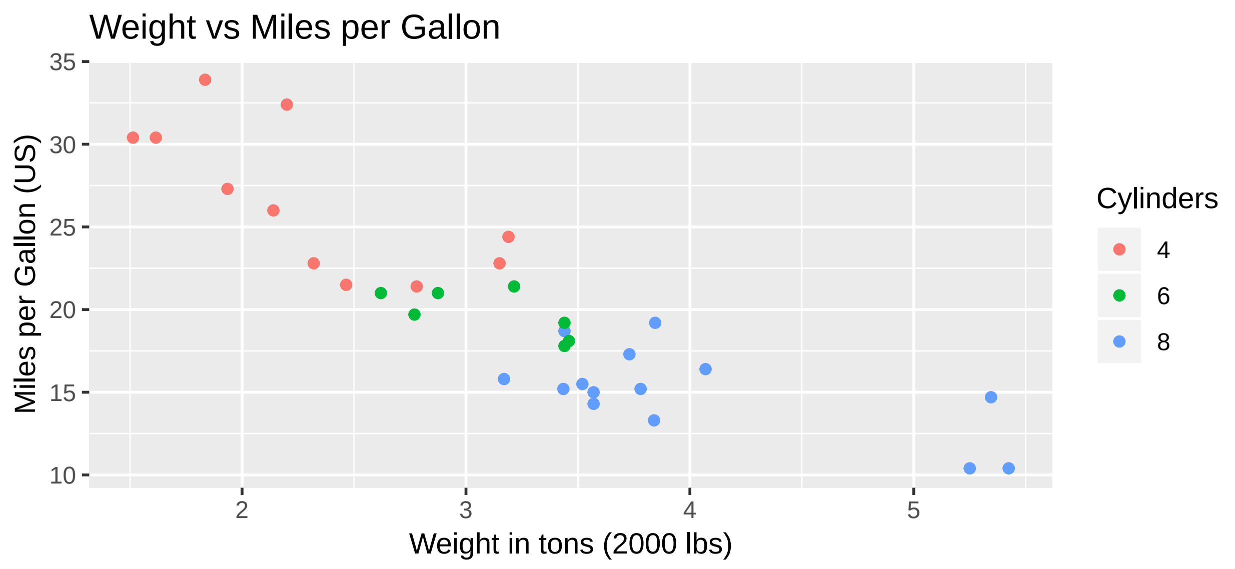
The boxes show the 25th, 50th, and 75th percentile and the lines coming off the box extend to the smallest and largest non-outlier observation.

3.1.5 Labels

To make a graph more understandable, it is necessary to tweak labels for the axes and add a main title and such. Here we'll adjust labels in a graph, including the legend labels.

```
# Treat the number of cylinders in a car as a categorical variable (4,6 or 8)
mtcars$cyl <- factor(mtcars$cyl)

ggplot(mtcars, aes(x=wt, y=mpg, col=cyl)) +
   geom_point() +
  labs( title='Weight vs Miles per Gallon') +
  labs( x="Weight in tons (2000 lbs)", y="Miles per Gallon (US)" ) +
  labs( color="Cylinders")
```



You could either call the `labs()` command repeatedly with each label, or you could provide multiple arguements to just one `labs()` call.

3.1.6 Color Scales

Adjusting the color palette for the color scales is not particularly hard, but it isn't intuitive. You can either set them up using a set of predefined palettes or you can straight up pick the colors. Furthermore we need to recognize that picking colors for a continuous covariate is different than for a factor. In the continuous case, we have to pick a low and high colors and `ggplot` will smoothly transition between the two. In the discrete case with a factor, each factor level gets its own color.

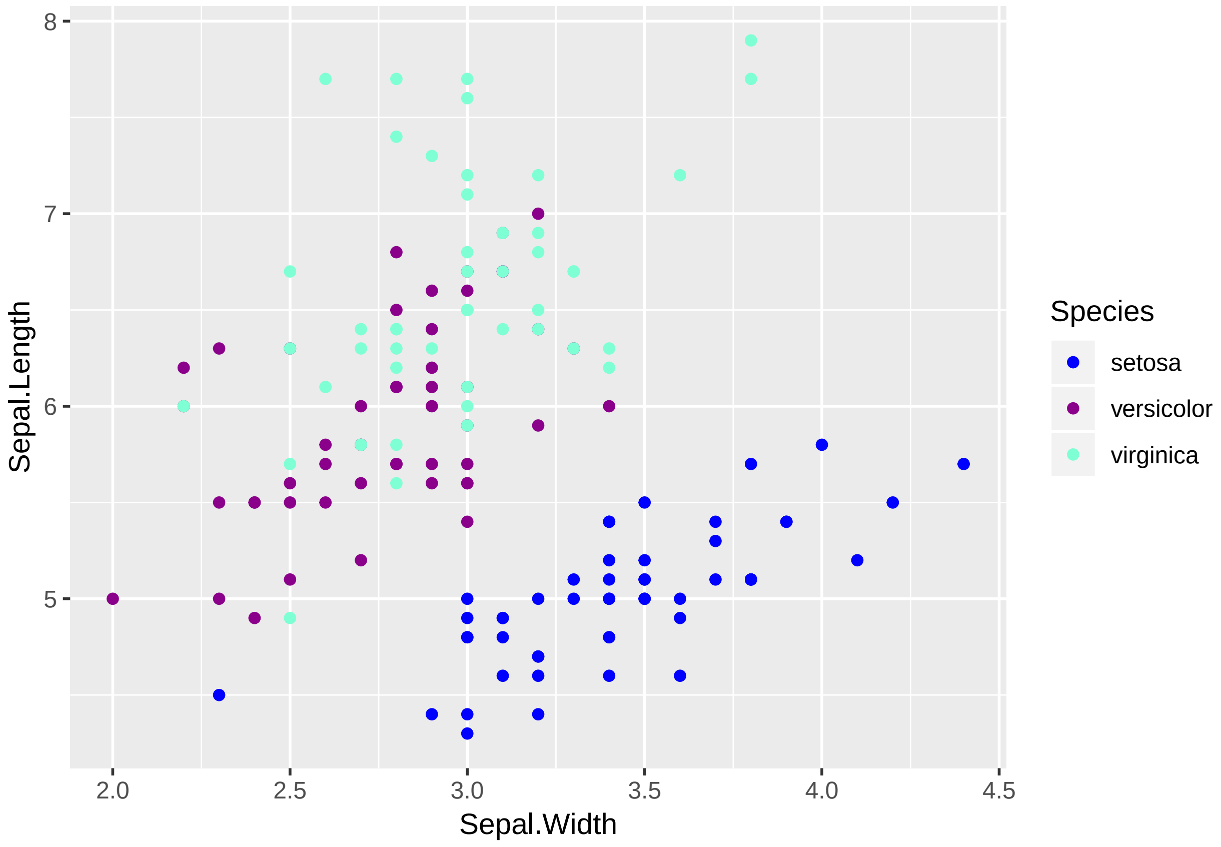
To make these choices, we will use the functions that modify the scales. In particular, if we are modifying the `color` aesthetic, we will use the `scale_color_XXX` functions where the `XXX` gets replaced by something more specific. If we are modifying the `fill` colors, then we will use the `scale_fill_XXX` family of functions.

3.1.6.1 Colors for Factors

We can set the colors manually using the function `scale_color_manual` which expects the name of the colors for each factor level. The order given in the `values` argument corresponds to the order of the levels of the factor.

For a nice list of the named colors you can use, I like to refer to this webpage: https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/colorPaletteCheatsheet.pdf

```r
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length, color=Species)) +
  geom_point() +
  scale_color_manual(values=c('blue', 'darkmagenta', 'aquamarine'))
```



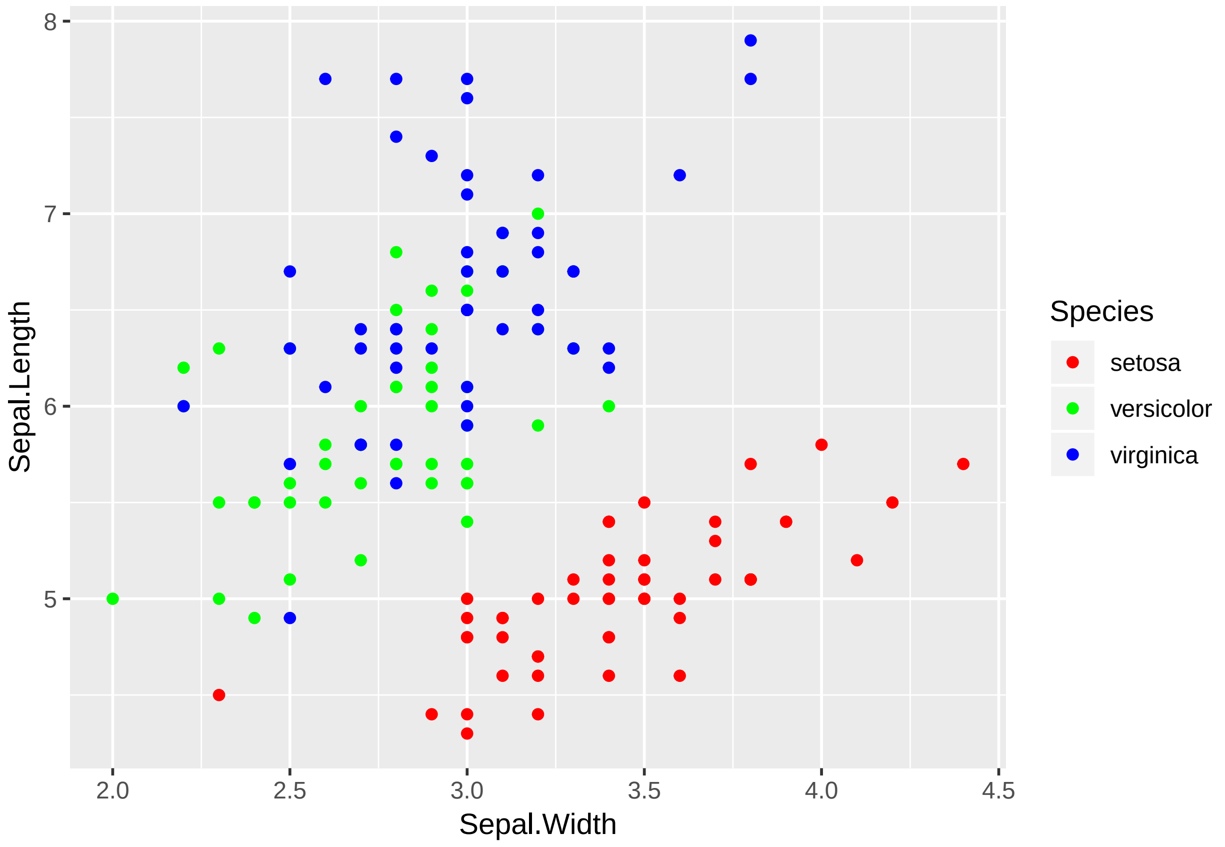
If you want to instead pick a color palette and let the palette pick the colors to be farthest apart based on the number of factor levels, you can use `scale_color_manual` and then have the values chosen by one of the palette functions where you just have to tell it how many levels you have.

```
library(colorspace)    # these two packages have some decent
library(grDevices)     # color palettes functions.

rainbow(6)    # if we have six factor levels, what colors should we use?
```

```
## [1] "#FF0000FF" "#FFFF00FF" "#00FF00FF" "#00FFFFFF" "#0000FFFF" "#FF00FFFF"
```

```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length, color=Species)) +
  geom_point() +
  scale_color_manual(values = rainbow(3))
```



3.1.6.2 Colors for continuous values

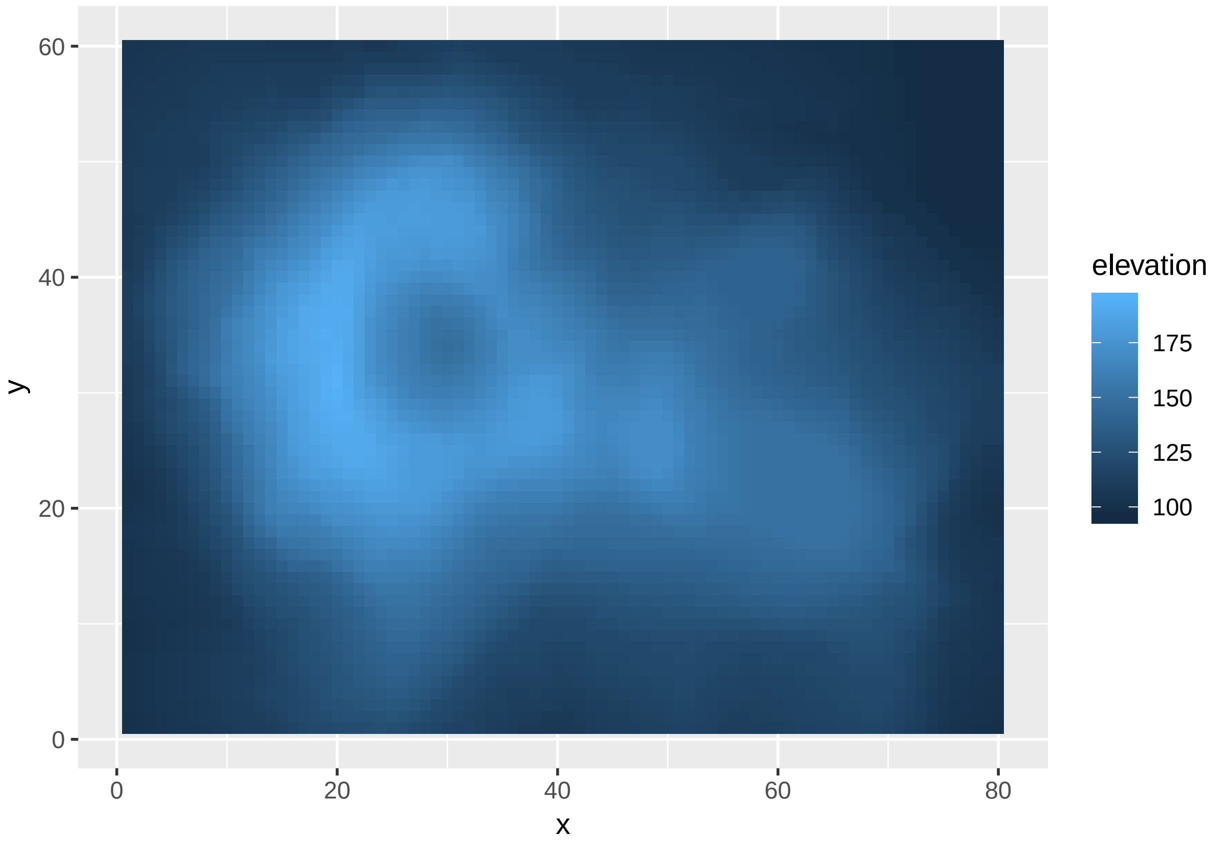
For this example, we will consider an elevation map of the Maunga Whau volcano in New Zealand. This dataset comes built into R as the matrix `volcano`, but I've modified it slightly and saved it to a package I have on github called `dsdata`

```
library(devtools)
install_github('dereksonderegger/dsdata')
```

```
## Skipping install of 'dsData' from a github remote, the SHA1 (43b2f6d8) has not changed since last install.
##   Use `force = TRUE` to force installation
```

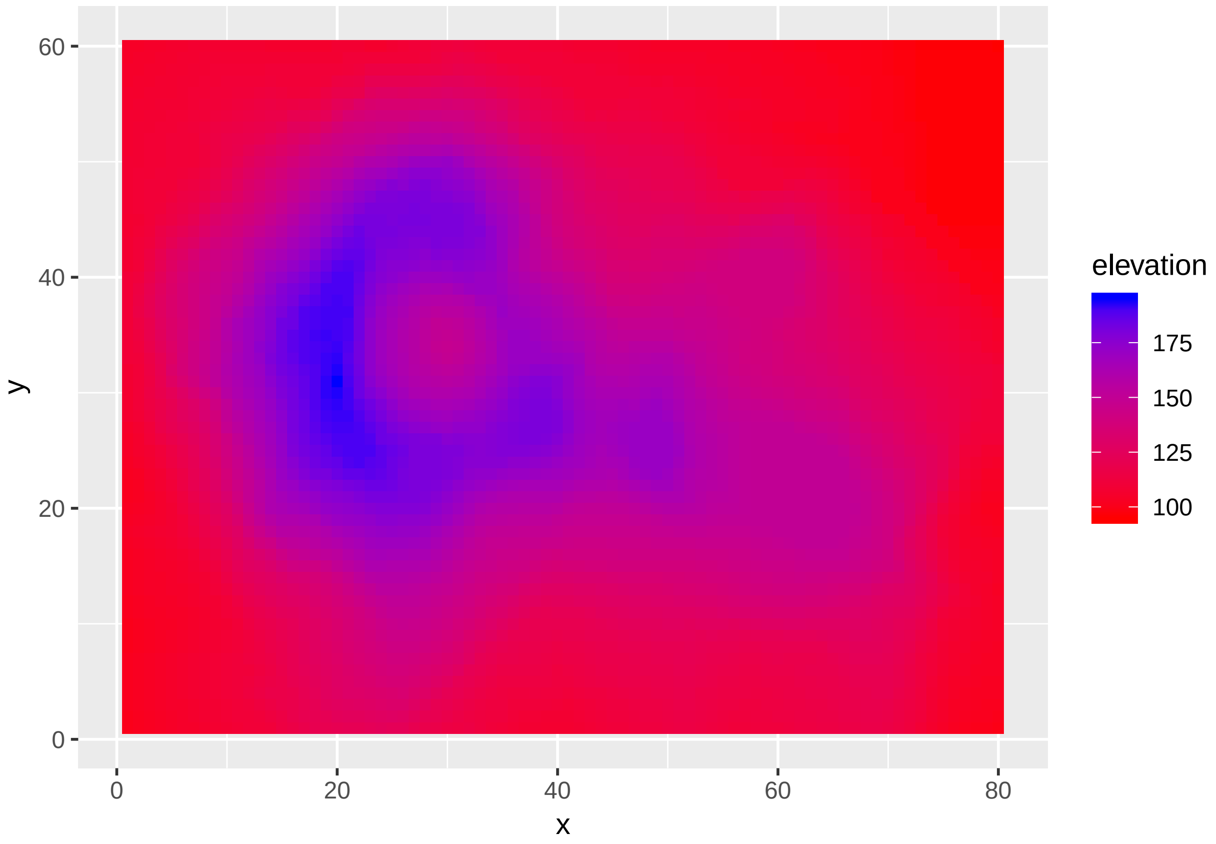
```
data('Eden', package='dsData')
```

```
ggplot( Eden, aes(x=x, y=y, fill=elevation)) +
  geom_raster()
```



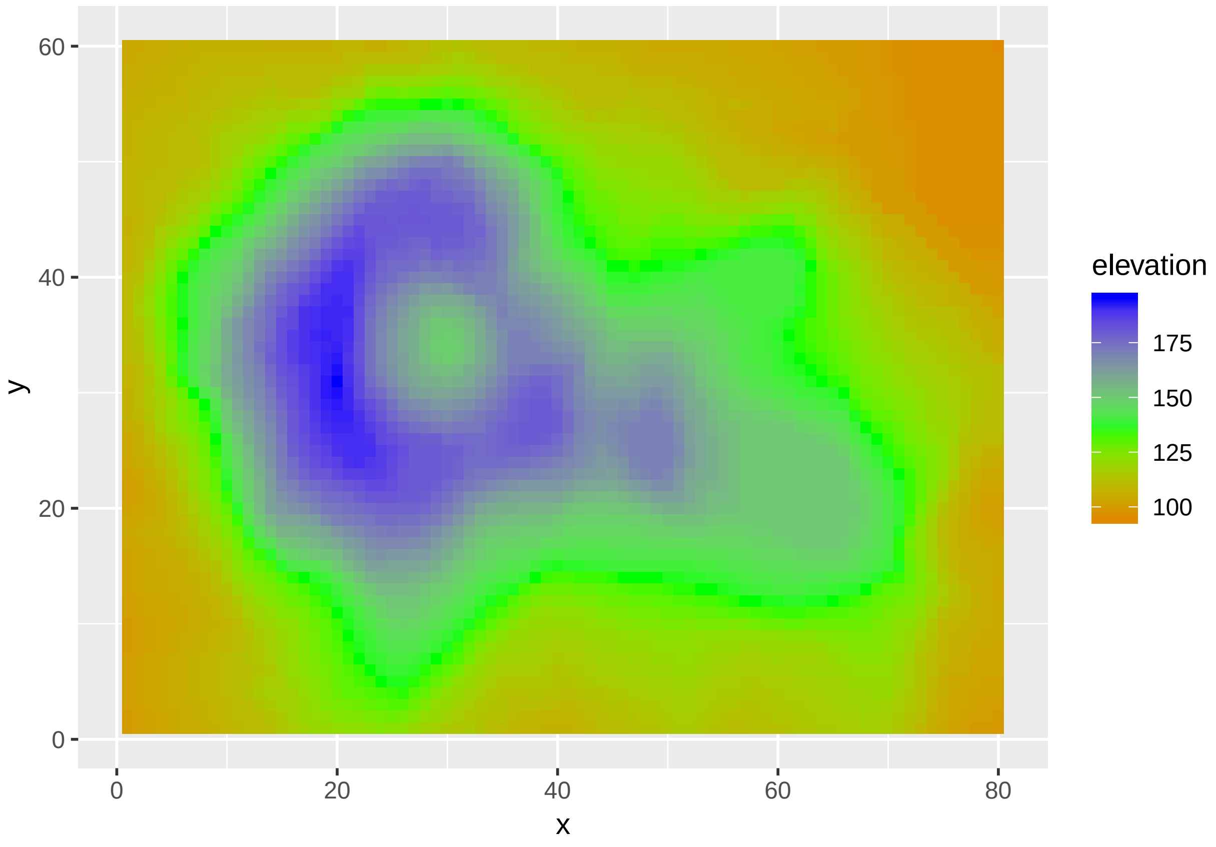
The default gradient isn't too bad, but we might want to manually chose two colors to smoothly scale between. Because I want to effect the colors I've chosen for the `fill` aesthetic, I have to modify this using `scale_fill_XXX`

```
ggplot( Eden, aes(x=x, y=y, fill=elevation)) +
  geom_tile() +
  scale_fill_gradient(low = "red", high = "blue")
```



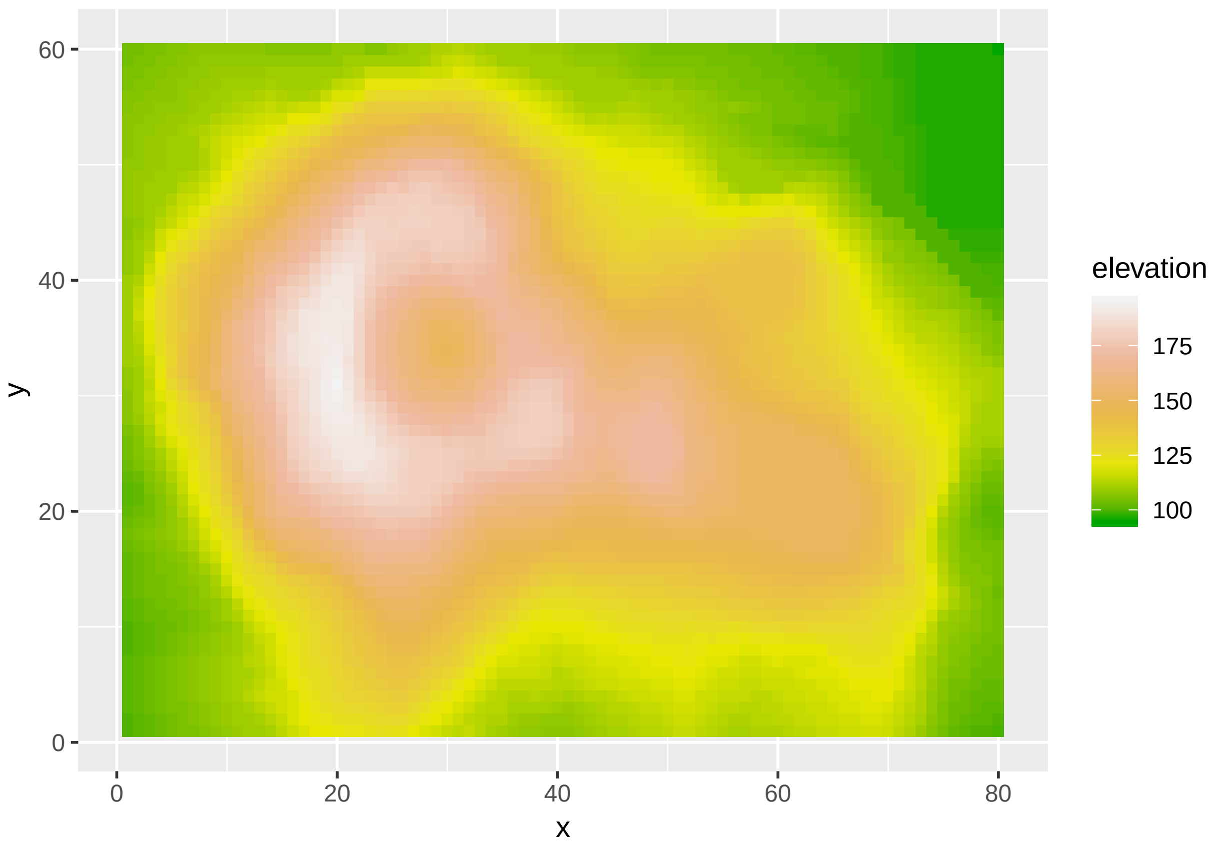
I think we ought to have the blue color come in a little earlier. Also, I want to specify a middle color so that our graph transitions from red to green to blue. To do this, we also have to specify where the middle color should be located along the elevation range.

```
ggplot( Eden, aes(x=x, y=y, fill=elevation)) +
  geom_tile() +
  scale_fill_gradient2(low = "red", mid='green', high = "blue",
                       midpoint=135)
```



If we don't want to specify the colors manually we can, as usual, specify the color palette. The `gradientn` functions allow us to specify a large numbers intermediate colors.

```
ggplot( Eden, aes(x=x, y=y, fill=elevation)) +
  geom_tile() +
  scale_fill_gradientn(colours = terrain.colors(5))
```

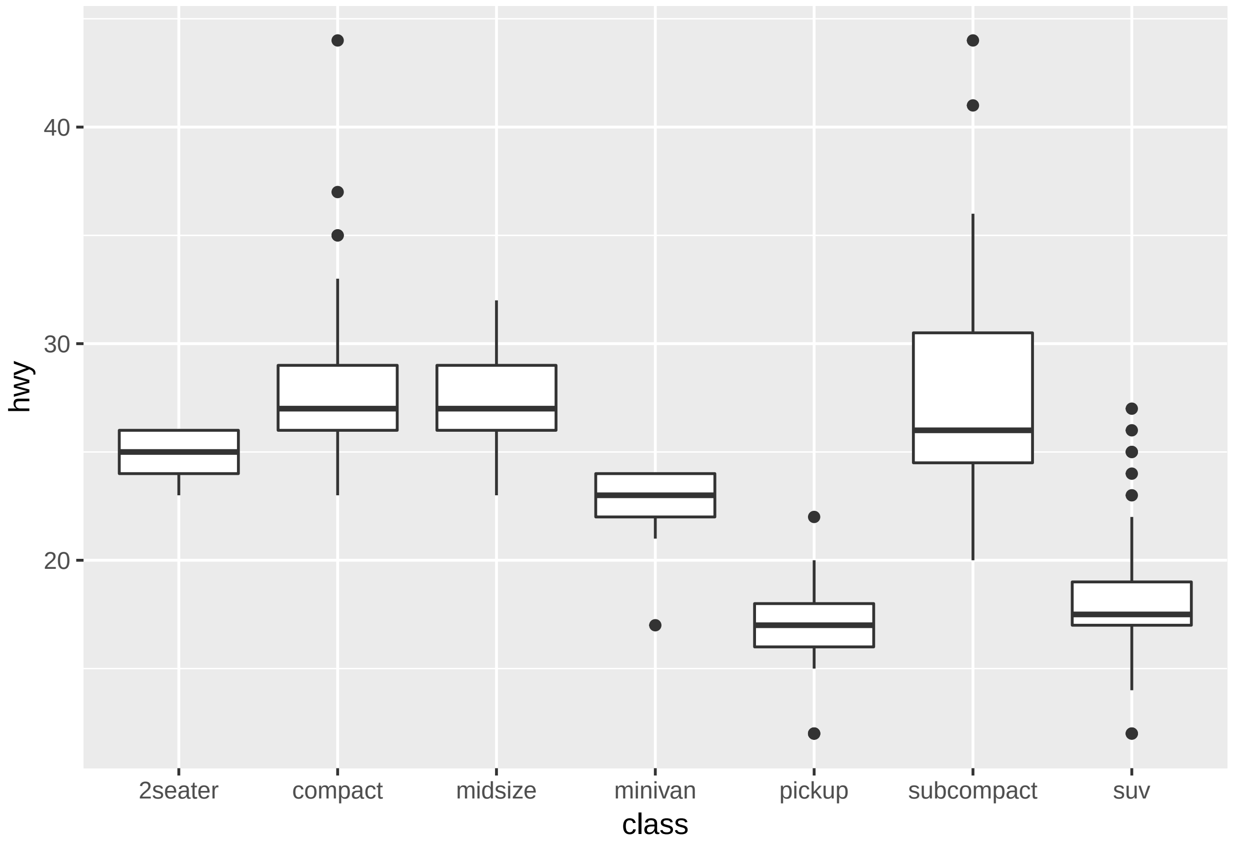


3.1.7 Adjusting axes

3.1.7.1 Setting breakpoints

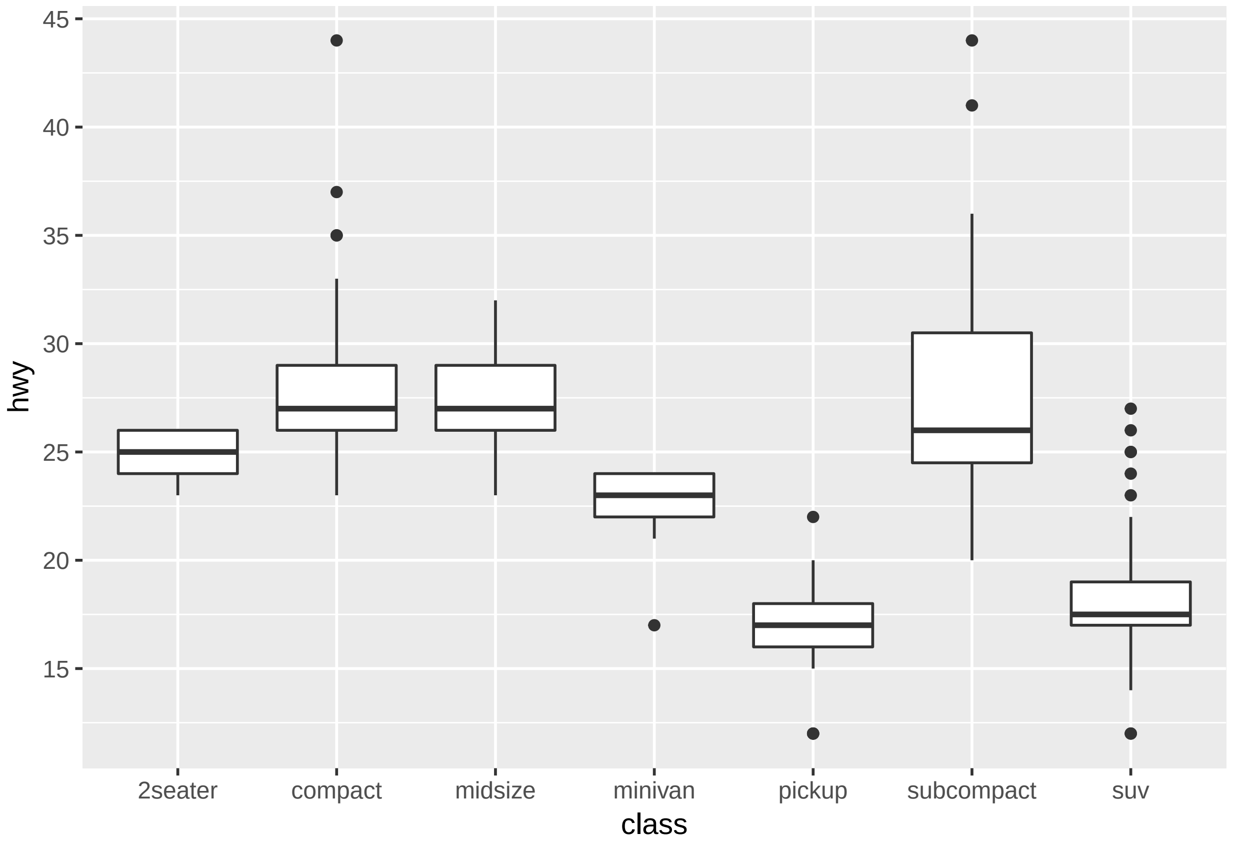
Sometimes the default axis breakpoints aren't quite what I want and I want to add a number or remove a number. To do this, we will modify the x or y scale. Typically I only have a problem when the axis is continuous, so we will concentrate on that case.

```
ggplot(mpg, aes(x=class, y=hwy)) +
  geom_boxplot()
```



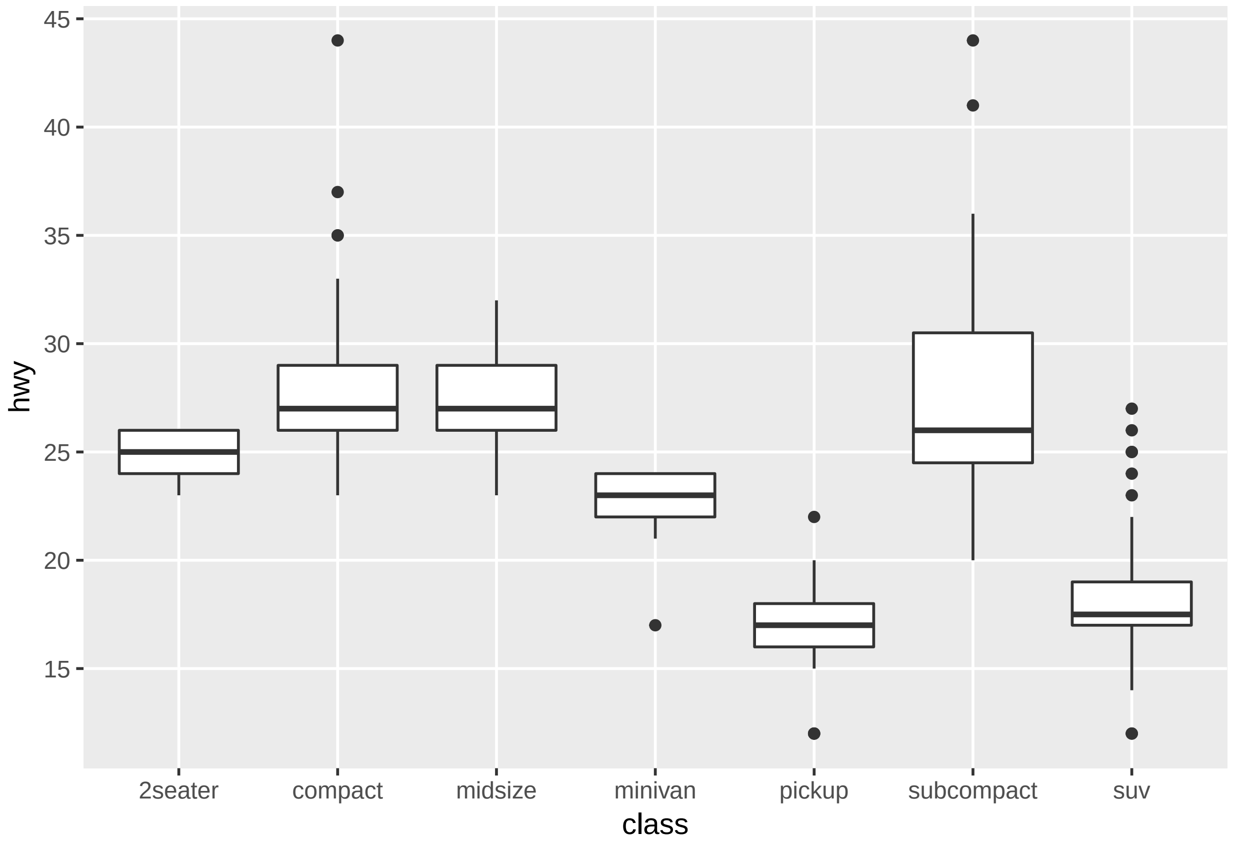
In this case, suppose that we want the major breakpoints (which have labels) to occur every 5 mpg, and the minor breakpoints (which just have a white line) to occur midway between those (so every 2.5 mpg).

```
ggplot(mpg, aes(x=class, y=hwy)) +
  geom_boxplot() +
  scale_y_continuous( breaks = seq(10, 45, by=5) )
```



If we wanted to adjust the minor breaks, we could do that using the `minor_breaks` argument. If we want to remove the minor breaks completely, we could set the minor breaks to be `NULL`

```
ggplot(mpg, aes(x=class, y=hwy)) +
  geom_boxplot() +
  scale_y_continuous( breaks = seq(10, 45, by=5), minor_breaks = NULL )
```

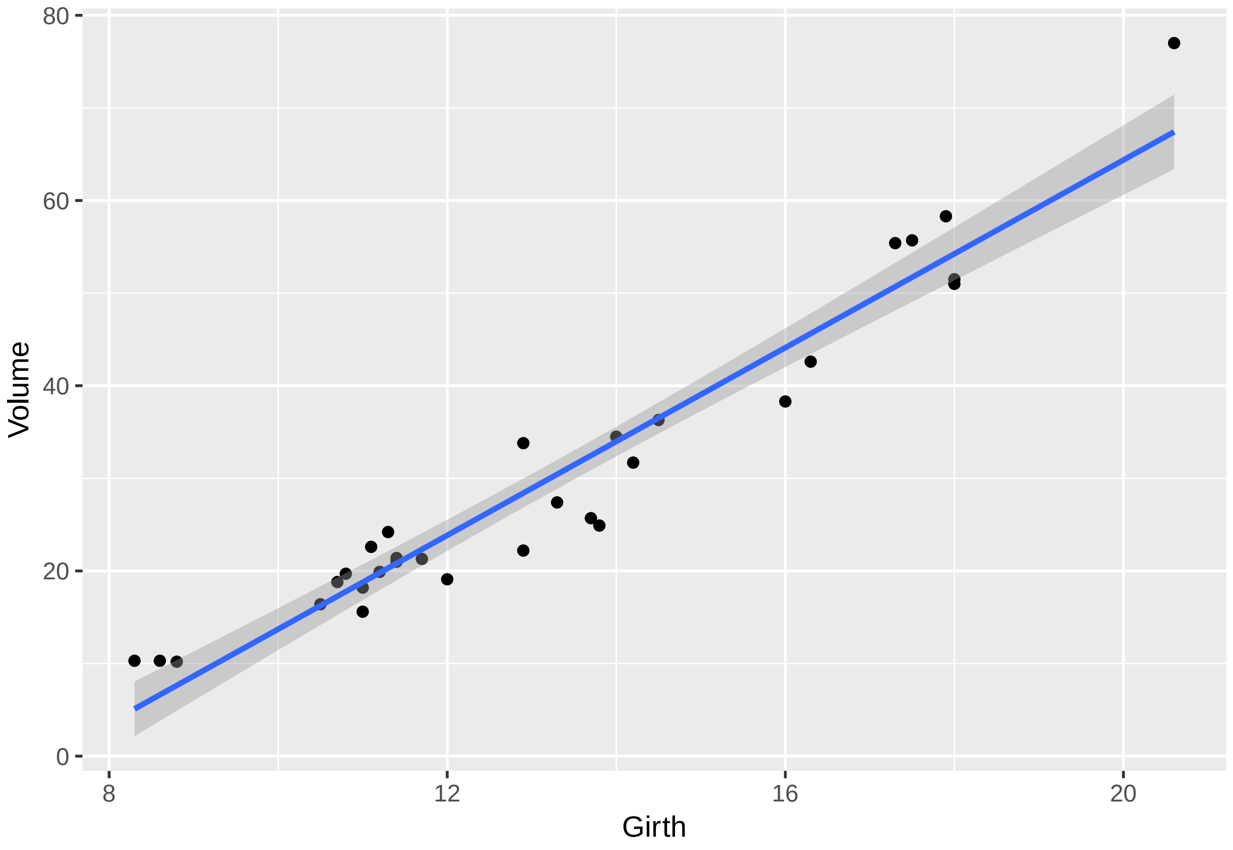


3.1.8 Zooming in/out

It is often important to be able to force the graph to have a particular range in either the x-axis or the y-axis. Given a particular range of interest, there are two ways that we could this:

- Remove all data points that fall outside the range and just plot the reduced dataset. This is accomplished using the `xlim()` and `ylim()` functions, or setting either of those inside another `scale_XXX` function.
- Use all the data to create a graph and just zoom in/out in that graph. This is accomplished using the `coord_cartesian()` function

```
ggplot(trees, aes(x=Girth, y=Volume)) +
  geom_point() +
  geom_smooth(method='lm')
```

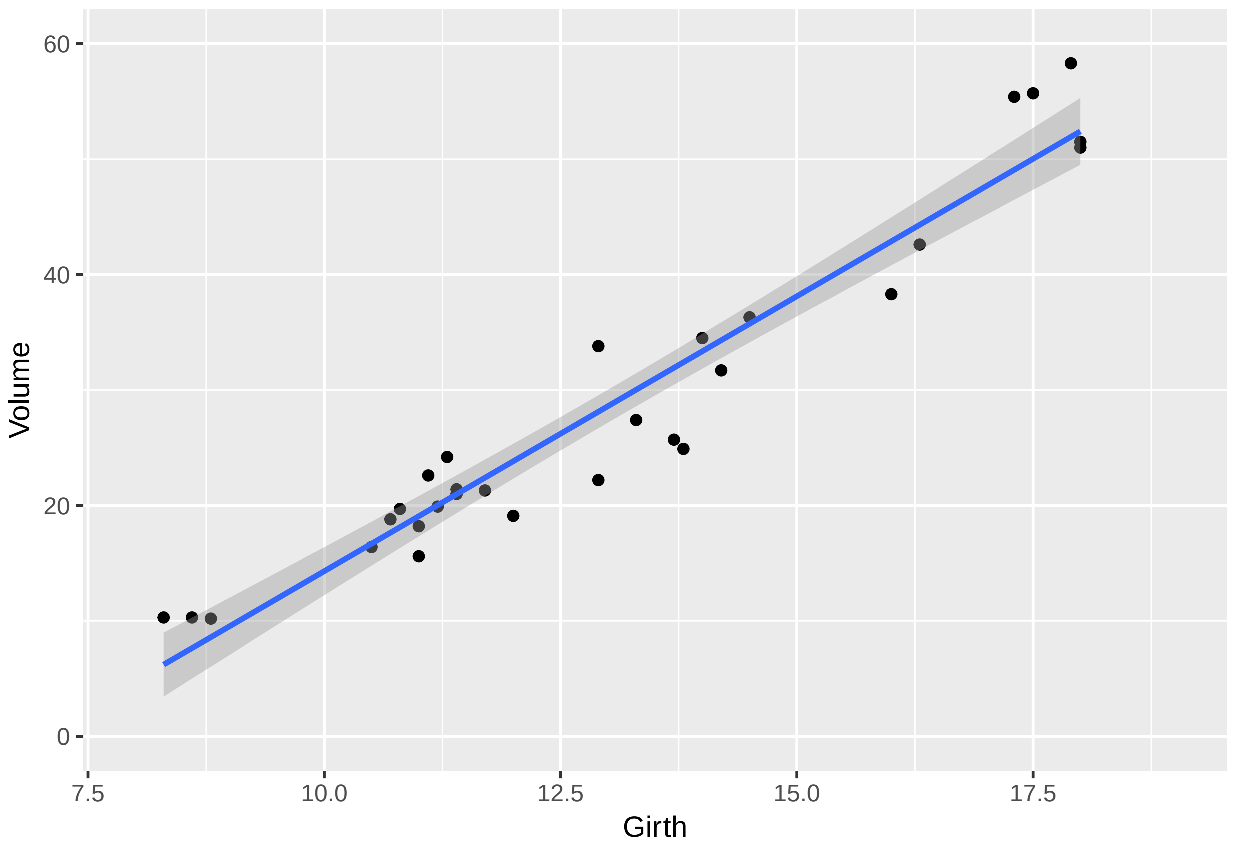


If we want to reset the x-axis to stop at $x = 19$, and $y = 60$, then we could use the `xlim()` and `ylim()` functions, but this will cause the regression line to be chopped off and it won't even use that data point when calculating the regression.

```
# Danger!  This removes the data points first!
ggplot(trees, aes(x=Girth, y=Volume)) +
  geom_point() +
  geom_smooth(method='lm') +
  xlim( 8, 19 ) + ylim(0, 60)
```

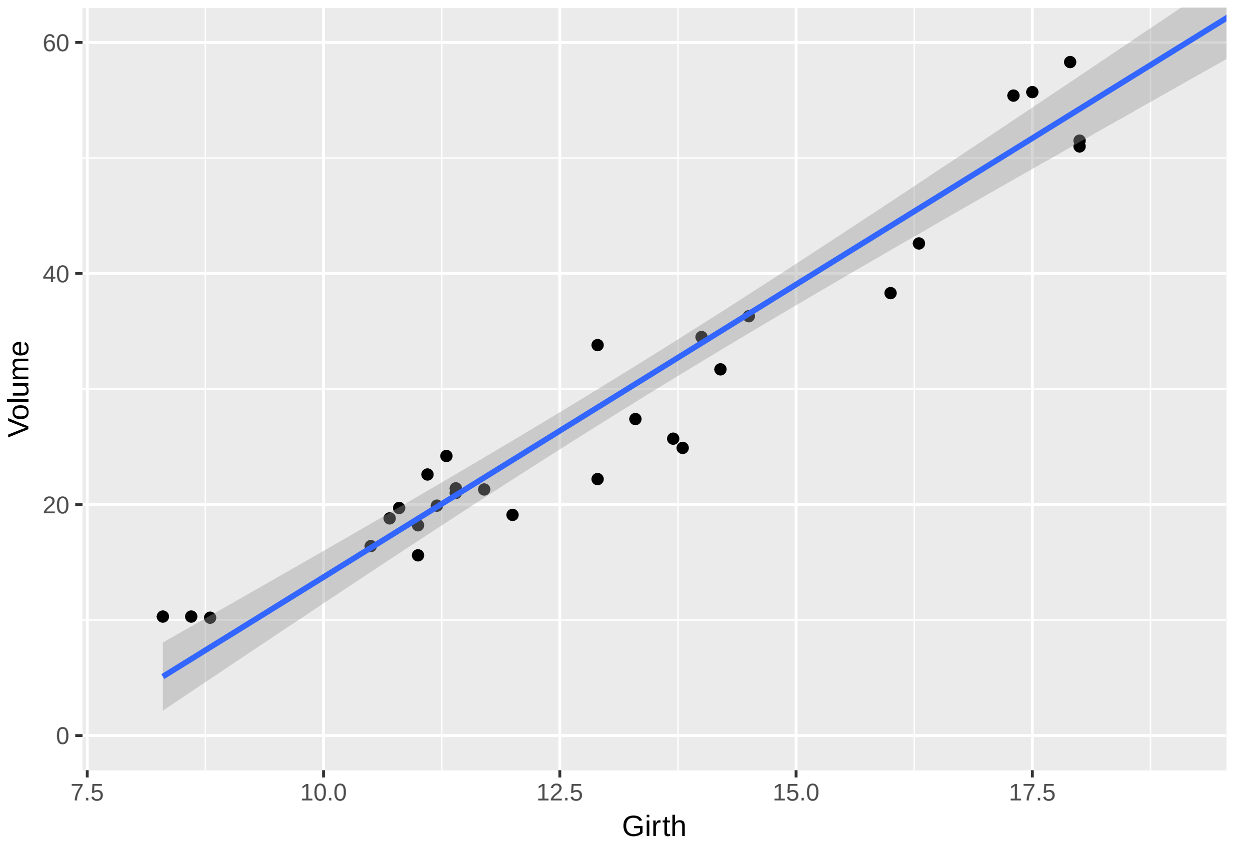
```
## Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



Alternatively, we could use the `coord_cartesion` function to chop the axes _after_everything has been calculated.

```
# Safer!  Create the graph and then just zoom in
ggplot(trees, aes(x=Girth, y=Volume)) +
  geom_point() +
  geom_smooth(method='lm') +
  coord_cartesian( xlim=c(8, 19 ), ylim=c(0, 60))
```



3.2 Cookbook Examples

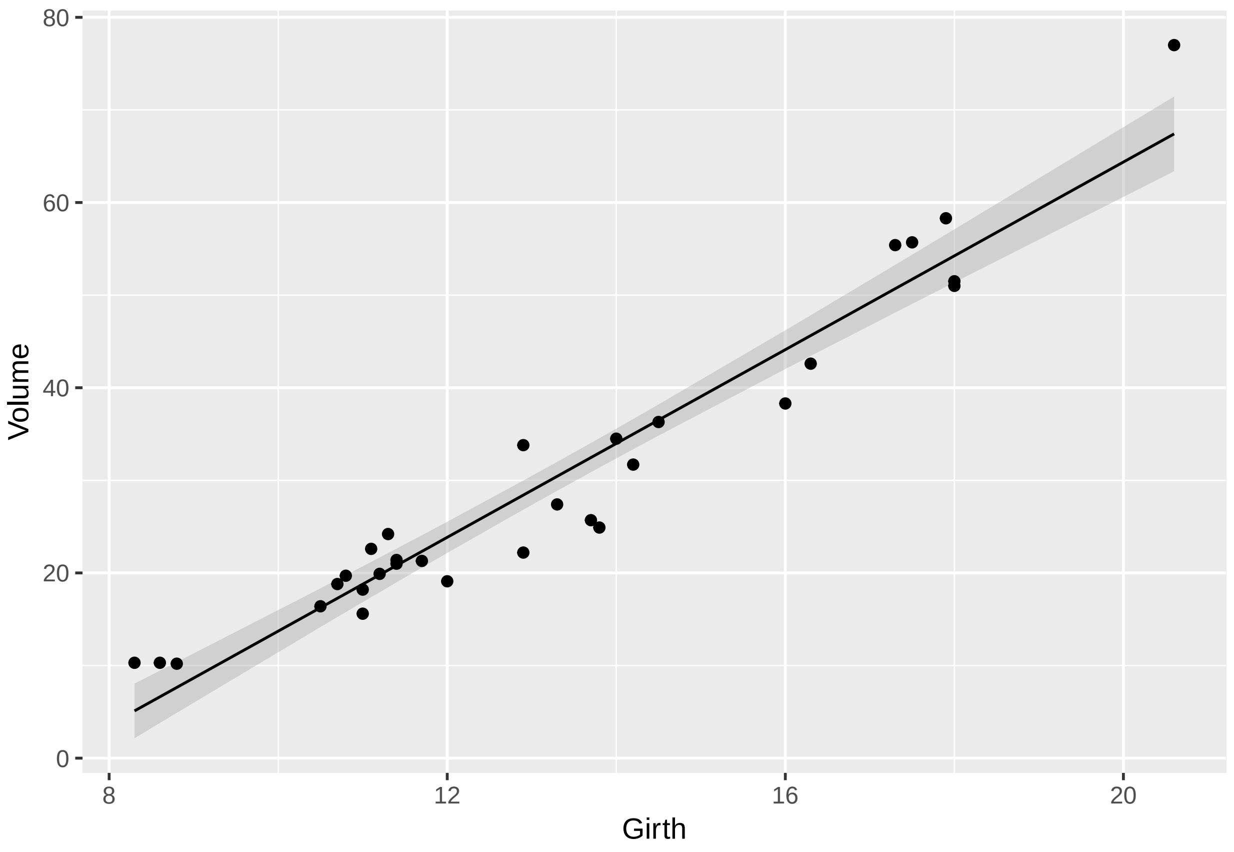
3.2.1 Scatterplot with prediction ribbons

Often I want to create a scatterplot and then graph the predicted values as a ribbon on top. While it is possible to do this automatically using the `geom_smoother()` function, I prefer not to do this because I don't have much control over how the model is created.

```
# fit a linear model to the trees dataset
model <- lm( Volume ~ Girth, data=trees )

# add the fitted values and confidence interval values for each observation
# to the original data frame, and call the augmented dataset trees.aug.
trees.aug <- trees %>% cbind( predict(model, interval='confidence', newdata=.) )


# Plot the augmented data. Alpha is the opacity of the ribbon
ggplot(trees.aug, aes(x=Girth, y=Volume)) +
  geom_ribbon( aes(ymin=lwr, ymax=upr), alpha=.4, fill='darkgrey' ) +
  geom_line( aes(y=fit) ) +
  geom_point( aes( y = Volume ) )
```



3.2.2 Bar Plot

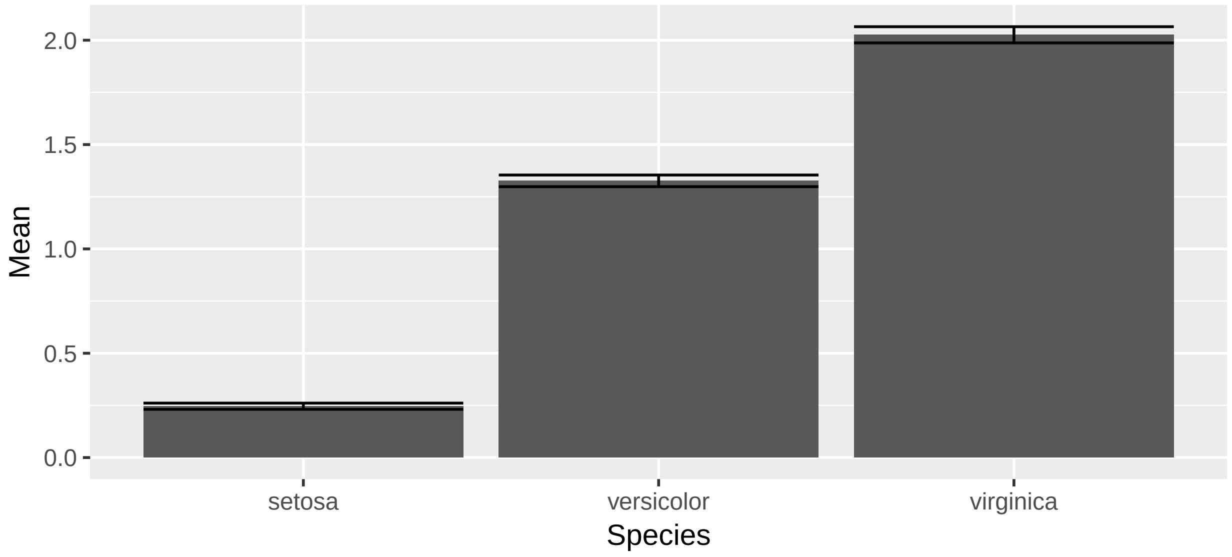
Suppose that you just want make some barplots and add ± S.E. bars. This should be really easy to do, but in the base graphics in R, it is a pain. Fortunately in `ggplot2` this is easy. First, define a data frame with the bar heights you want to graph and the ± values you wish to use.

```
# Calculate the mean and sd of the Petal Widths for each species
stats <- iris %>%
  group_by(Species) %>%
  summarize( Mean = mean(Petal.Width),                    # Mean   = ybar
             StdErr = sd(Petal.Width)/sqrt(n()) ) %>%  # StdErr = s / sqrt(n)
  mutate( lwr = Mean - StdErr,
          upr = Mean + StdErr )
stats
```

```
## # A tibble: 3 x 5
##   Species     Mean StdErr   lwr   upr
##   <fct>      <dbl>  <dbl> <dbl> <dbl>
## 1 setosa     0.246 0.0149 0.231 0.261
## 2 versicolor 1.33  0.0280 1.30  1.35
## 3 virginica  2.03  0.0388 1.99  2.06
```

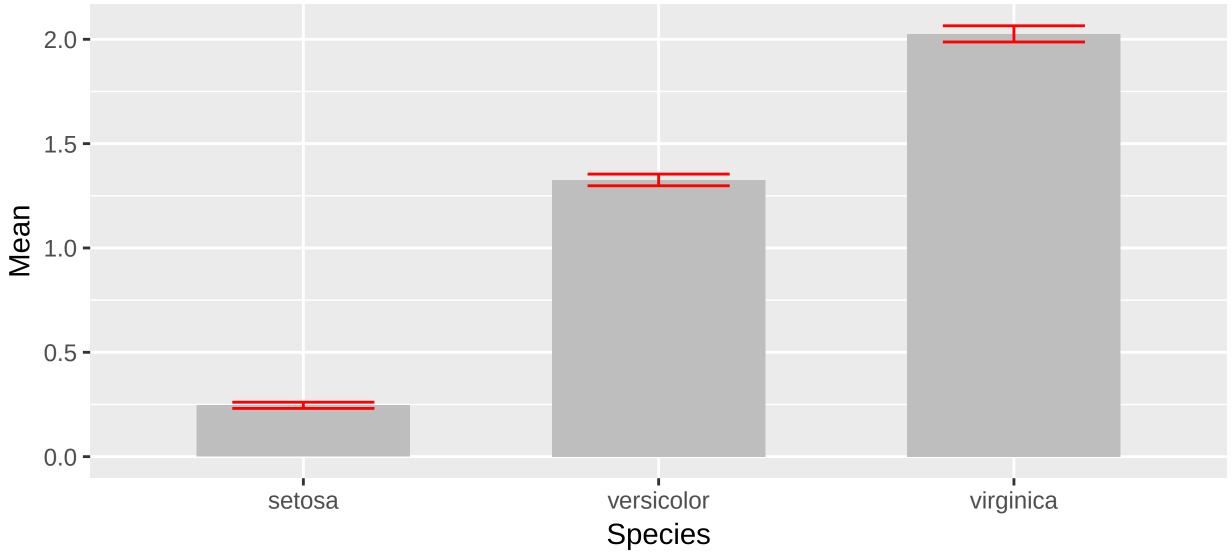
Next we take these summary statistics and define the following graph which makes a bar graph of the means and error bars that are ± 1 estimated standard deviation of the mean (usually referred to as the standard errors of the means). By default, `geom_bar()` tries to draw a bar plot based on how many observations each group has. What I want, though, is to draw bars of the height I specified, so to do that I have to add `stat='identity'` to specify that it should just use the heights I tell it.

```r
ggplot(stats, aes(x=Species)) +
  geom_bar( aes(y=Mean), stat='identity') +
  geom_errorbar( aes(ymin=lwr, ymax=upr) )
```



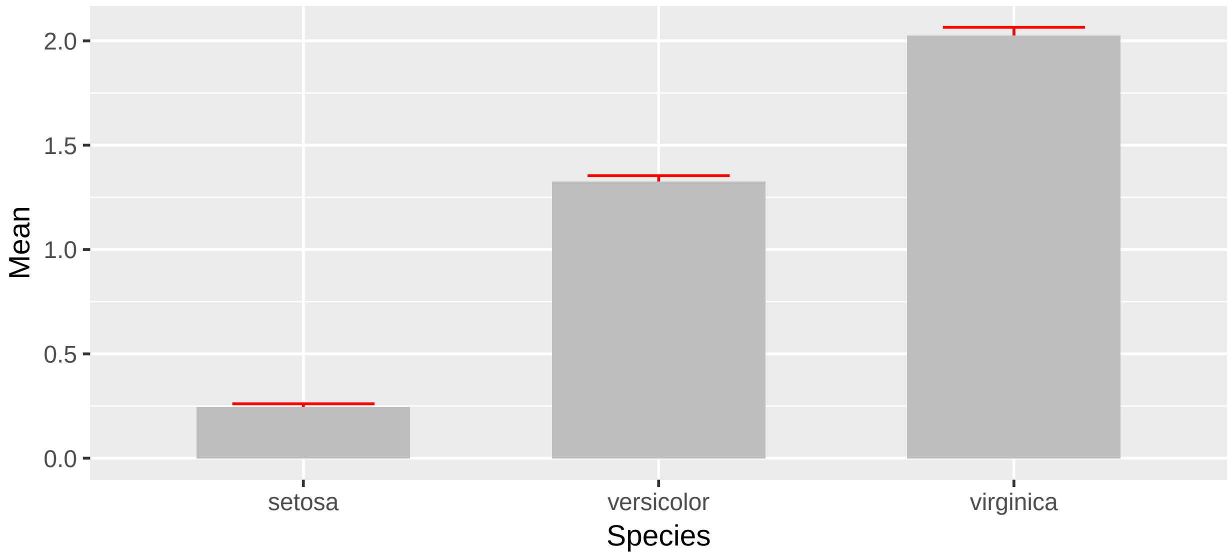
While this isn't too bad, we would like to make this a bit more pleasing to look at. Each of the bars is a little too wide and the error bars should be a tad narrower than then bar. Also, the fill color for the bars is too dark. So I'll change all of these, by setting those attributes *outside of an* `aes()` *command*.

```r
ggplot(stats, aes(x=Species)) +
  geom_bar( aes(y=Mean), stat='identity', fill='grey', width=.6) +
  geom_errorbar( aes(ymin=lwr, ymax=upr), color='red', width=.4 )
```



The last thing to notice is that the *order* in which the different layers matter. This is similar to Photoshop or GIS software where the layers added last can obscure prior layers. In the graph below, the lower part of the error bar is obscured by the grey bar.

```r
ggplot(stats, aes(x=Species)) +
  geom_errorbar( aes(ymin=lwr, ymax=upr), color='red', width=.4 ) +
  geom_bar( aes(y=Mean), stat='identity', fill='grey', width=.6)
```



3.3 Exercises

1. For the dataset trees, which should already be pre-loaded. Look at the help file using `?trees` for more information about this data set. We wish to build a scatterplot that compares the height and girth of these cherry trees to the volume of lumber that was produced.

    a) Create a graph using ggplot2 with Height on the x-axis, Volume on the y-axis, and Girth as the either the size of the data point or the color of the data point. Which do you think is a more intuitive representation?
    b) Add appropriate labels for the main title and the x and y axes.

2. Consider the following small dataset that represents the number of times per day my wife played "Ring around the Rosy" with my daughter relative to the number of days since she has learned this game. The column `yhat` represents the best fitting line through the data, and `lwr` and `upr` represent a 95% confidence interval for the predicted value on that day.

```
Rosy <- data.frame(
  times = c(15, 11, 9, 12, 5, 2, 3),
  day   = 1:7,
  yhat  = c(14.36, 12.29, 10.21, 8.14, 6.07, 4.00,  1.93),
  lwr   = c( 9.54,  8.5,   7.22, 5.47, 3.08, 0.22, -2.89),
  upr   = c(19.18, 16.07, 13.2, 10.82, 9.06, 7.78,  6.75))
```

    a) Using `ggplot()` and `geom_point()`, create a scatterplot with `day` along the x-axis and `times` along the y-axis.

    b) Add a line to the graph where the x-values are the `day` values but now the y-values are the predicted values which we've called `yhat`. Notice that you have to set the aesthetic y=times for the points and y=yhat for the line. Because each `geom_` will accept an `aes()` command, you can specify the `y` attribute to be different for different layers of the graph.

    c) Add a ribbon that represents the confidence region of the regression line. The `geom_ribbon()` function requires an `x`, `ymin`, and `ymax` columns to be defined. For examples of using `geom_ribbon()` see the online documentation: http://docs.ggplot2.org/current/geom_ribbon.html.

```
ggplot(Rosy, aes(x=day)) +
  geom_point(aes(y=times)) +
  geom_line( aes(y=yhat)) +
  geom_ribbon( aes(ymin=lwr, ymax=upr), fill='salmon')
```

d) What happened when you added the ribbon? Did some points get hidden? If so, why?

e) Reorder the statements that created the graph so that the ribbon is on the bottom and the data points are on top and the regression line is visible.

f) The color of the ribbon fill is ugly. Use Google to find a list of named colors available to `ggplot2`. For example, I googled "ggplot2 named colors" and found the following link: http://sape.inf.usi.ch/quick-reference/ggplot2/colour. Choose a color for the fill that is pleasing to you.

g) Add labels for the x-axis and y-axis that are appropriate along with a main title.

3. The R package `babynames` contains a single dataset that lists the number of children registered with Social Security with a particular name along with the proportion out of all children born in a given year. The dataset covers the from 1880 to the present. We want to plot the relative popularity of the names 'Elise' and 'Casey'.

   a) Load the package. If it is not found on your computer, download the package from CRAN.

   ```
   library(babynames)
   data("babynames")
   ```

   b) Read the help file for the data set `babynames` to get a sense of the columns

   c) Create a small dataset that only has the names 'Elise' and 'Casey'.

   d) Make a plot where the x-axis is the year and the y-axis is the proportion of babies given the names. Use a line to display this relationship and distinguish the two names by color. Notice this graph is a bit ugly because there is a lot of year-to-year variability that we should smooth over.

   e) We'll use `dplyr` to collapse the individual years into decades using the following code:

   ```
   small <- babynames %>%
     filter( name=='Elise' | name=='Casey') %>%
     mutate( decade = cut(year, breaks = seq(1869,2019,by=10) )) %>%
     group_by(name, decade) %>%
     summarise( prop = mean(prop),
                year = min(year))
   ```

   f) Now draw the same graph you had in part (d).

   g) Next we'll create an area plot where the height is the total proportion of the both names and the colors split up the proportion.

```
ggplot(small, aes(x=year, y=prop, fill=name)) +
   geom_area()
```

This is a pretty neat graph as it show the relative popularity of the name over time and can easily be expanded to many many names. In fact, there is a wonderful website that takes this same data and allows you select the names quite nicely: http://www.babynamewizard.com/voyager. My wife and I used this a lot while figuring out what to name our children. Notice that this site really uses the same graph type we just built but there are a few extra neat interactivity tricks.

Chapter 4

Introduction to Data Wrangling using `dplyr`

4.1 `mutate`

4.2 `group_by`

4.3 `summarize`

Chapter 5

Introduction to using Model Objects

5.1 formula notation

5.2 Building models

5.2.1 t-tests

5.2.2 lm objects

5.3 Accessor function using traditional functions

5.4 Accessing model results using `broom`

Chapter 6

Graphing data and model results

6.1 Process

6.2 Augmenting data with model output

6.3 Plotting using `ggplot2`

Deeper Details

Chapter 7

Factors

7.1 How factors are stored

7.2 Using `forcats`

Chapter 8

Data Frames, tibbles, oh my!

8.1 Why extend the data frame

Chapter 9

Data Import

Chapter 10

Data frame Manipulation using dplyr

Chapter 11

Data frame

11.1 reshaping

11.2 merges

Chapter 12

More ggplot2

Chapter 13

Flow Control

Chapter 14

Functions

Chapter 15

Strings

Chapter 16

Dates

Chapter 17

Performance Issues